Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Machine Learning on Microcontrollers in the Automotive Industry

Maschinelles Lernen auf Mikrocontrollern in der Automobilindustrie

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Florian Walter, M.Sc.

Author Florian Donhauser

Date November 16, 2020 in Garching

Disclaimer

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, November 16, 2020

(Florian Donhauser)

Abstract

This thesis investigates how microcontrollers can be used for machine learning in the automotive industry. Car development and production, driver experience, fault diagnosis and predictive maintenance, and mobility services are identified to be the main areas of use cases for machine learning in this industry. Three example programs are deployed on an STM32F746 microcontroller using the TensorFlow Lite for Microcontrollers library. The commonly used MobileNet architecture is shown to be slow and not applicable to many real-time tasks. The UPM Vehicle Image Database is used for binary image classification of cars in low-resolution images using dense neural networks and convolutional neural networks, a task that might be used for advanced driver-assistance systems. Only very small convolutional layers achieve a low enough inference time. The best dense neural network achieved an accuracy of 97.0% with an inference time of 8.6 ms while inference took 11.73 ms for a convolutional neural network with an accuracy of 94.8%. The promising area of fault diagnosis is tested by identifying seven different fault types of an internal combustion engine using different dense neural networks. The system achieved an accuracy of up to 96.8% while keeping the inference time at only 1.7 ms. These two successful proof-of-concepts show possible applications for the trending topic of edge machine learning.

Zusammenfassung

Diese Bachelorarbeit untersucht, wie Mikrocontroller für maschinelles Lernen in der Automobilindustrie eingesetzt werden können. Autoentwicklung und -produktion, Fahrerlebnis, Fehlerdiagnose und vorausschauende Wartung, sowie Mobilitätsdienste werden als Hauptgruppen von Anwendungsfällen für maschinelles Lernen in dieser Branche identifiziert. Drei Beispielprogramme werden auf einem STM32F746 Mikrocontroller unter Verwendung der TensorFlow Lite for Microcontrollers Bibliothek getestet. Die häufig verwendete MobileNet-Architektur ist langsam und für viele Echtzeitaufgaben nicht anwendbar. Die UPM Vehilce Image Database wird zur binären Klassifizierung von Autos in Bildern mit niedriger Auflösung unter Verwendung dichter neuronaler Netze und konvolutioneller neuronaler Netze verwendet, eine Aufgabe, die bei Fahrerassistenzsystemen auftreten kann. Nur sehr kleine Faltungsschichten erreichen eine ausreichend niedrige Laufzeit. Das beste dichte neuronale Netzwerk erreichte eine Genauigkeit von 97,0% mit einer Inferenzzeit von 8,6 ms, während die Inferenz für ein konvolutionelles neuronales Netzwerk mit einer Genauigkeit von 94,8% 11,73 ms dauerte. Der vielversprechende Bereich der Fehlerdiagnose wird getestet, indem sieben unterschiedliche Fehlertypen eines Verbrennungsmotors mit verschiedenen neuronalen Netzen klassifiziert werden. Das System erreichte eine Genauigkeit von bis zu 96,8% bei einer Inferenzzeit von nur 1,7 ms. Diese beiden erfolgreichen Machbarkeitsnachweise zeigen mögliche Anwendungen für das Trendthema Edge Machine Learning.

Contents

Chapter 1

Introduction

Machine Learning is one of the most trending topics in computer science right now and might revolutionize many fields, including the automotive industry. Here, producing a smart self-driving car is the goal of many automakers like Tesla[Tes], Daimler[Dai], Ford[For] or BMW[BMW]. In general, big datasets are often needed for machine learning and some applications require constant internet access to offer all their features, e.g. personal voice assistants like Amazon Alexa[Joh] or image recognition apps like Google Lens[Goo]. If the calculations are done in the cloud, the data is shared with the provider of this service raising the question of data privacy. Furthermore, constant and fast internet access is not always possible since e.g. Germany only had a 4G coverage of 76.9% in 2019 according to Opensignal[Boy19].
The alternative to sending all data to the cloud is to deploy some form of edge computing which keeps the data and computation closer to the user's device. Machine learning can even work completely offline on the user's device without an internet connection. The advantages of running machine learning algorithms on edge devices instead of in the cloud have been highlighted recently by George Plastiras et al.[Pla+18], Shuiguang Deng et al.[Den+20] and Dianlei Xu et al.[Xu+20]. It reduces latency, which is important for time-critical applications and many real-time solutions do not have the time to communicate with a cloud server before taking action[Pla+18]. Furthermore, edge machine learning can increase the privacy of the user's data[Pla+18], reduce communication costs[Den+20], and helps with scalability[Xu+20].
Edge machine learning is especially interesting for the automotive industry because machine learning applications in the vehicle that require an online connection can be challenging while the vehicle is moving since it might enter an area with a slow or even no internet connection at all. Additionally, the data might be highly sensitive if it e.g. contains voice recordings or a location history of the car, so the added privacy is very beneficial[Pla+18]. Real-time applications like driver assistance systems also require a low latency, which might not be feasible when the data needs to be sent to a server first. Since modern cars already have many embedded microcontrollers, about 50 per car was an estimate from 2011[Fle11], the hardware for edge machine learning is already there. Additionally, the automotive industry uses microcontrollers in other areas like production[Hom18], where the usage of offline machine learning could be beneficial as well.

This thesis will investigate how inexpensive microcontrollers could be used for machine learning in the automotive industry. Since this thesis will deal with different machine learning algorithms, the relevant ones will be briefly described at the beginning to provide a common knowledge basis. Afterwards, this work will look at which use cases can be run on a microcontroller as well as the restrictions this hardware-choice creates. The existing applications of machine learning on embedded devices and microcontrollers will be described in

the related work section. Since new applications in the automotive industry are of interest, the general use of artificial intelligence in the automotive industry is also analyzed and evaluated for possible machine learning solutions running on a microcontroller. This use case analysis will result in candidates for machine learning on microcontrollers. Additionally, to test the feasibility of these use cases, this thesis will look at three applications running on an STM32F746 microcontroller using the TensorFlow Lite for Microcontrollers library. The workflow of using this library is described as well as the hardware before three examples are deployed. They include object detection using a MobileNet, a vehicle image dataset classified using different neural networks, and a possible application for fault diagnosis and predictive maintenance. Each of the three examples contains Python code that is used to train a machine learning model in TensorFlow and to evaluate its accuracy. Afterwards, the developed model is deployed using a C++ program on the STM microcontroller and its runtime is measured, showing its applicability for real-time problems. Hence, these examples can be proof-of-concepts for real-world implementations.
The combination of the literature search in the related work section and the use case analysis, combined with the practical part, containing the three examples, aims to answer the original question of how microcontrollers can be used for machine learning in the automotive industry.

Chapter 2

Machine Learning Algorithms

This thesis will deal with different Machine Learning (ML) algorithms which are used in related work, different use cases in the automotive industry, and in the three example programs of this thesis. To provide a common knowledge basis, the most important ML Algorithms for this thesis will be briefly described starting with Decision Tree (DT), followed by K-Nearest Neighbors (KNN), Support Vector Machine (SVM), and different forms of Neural Networks (NN), namely Dense Neural Network, Recurrent Neural Network (RNN), and Convolutional Neural Network (CNN). The focus will be on NNs since they are the ML algorithm being used for examples in this thesis.
The different ML abbreviations used in this thesis are also noted in table 2.1.

Abbreviation	Meaning
AI	Artificial Intelligence
CNN	Convolutional Neural Network
DT	Decision Tree
KNN	K-Nearest Neighbors
ML	Machine Learning
MLP	Multilayer Perceptron
NN	Neural Network
RNN	Recurrent Neural Network
SVM	Support Vector Machine

Table 2.1: Machine Learning Abbreviations

All these ML algorithms are so-called supervised learning algorithms which means they require a set of training data consisting of a set of inputs X and their matching outputs Y. According to Eaton[Eat17], the goal of a supervised learning algorithm is to come up with a function that maps the input to the correct output as best as possible, meaning the ML algorithm tries to find a function that approximates $f(X)=Y$. If the output Y is a category, also called a label, the process is called classification[Eat17]. One example would be pictures that contain different animals and for each picture, the animal needs to be identified. If the output Y is a numerical value, it is called regression. One example would be a ML algorithm that automatically appraises cars, the output of which is a number for the approximated price and not one of several categories. The phase of using the already labeled data to come up with a function is called the training phase. Afterwards, this function can be used to give guesses for outputs of new input data, which is called inference. To test how well a ML algorithm generalizes for new data, the available labeled data is usually split up into training and testing sets. The training set is used to calculate an approximation of the function between

input X and output Y during training, the testing set is not looked at in this step. Afterwards, the input values of the test set are used for inference and the inferred results are compared to the real output values of the test set. The testing allows the real accuracy and quality of the ML algorithm to be evaluated since the goal is to be able to deal with new data. The ML algorithm should perform well with the data it was trained with and also equally good with new data, which is evaluated with the test set. If an algorithm only memorizes the training set but does not understand the underlying relationships between input features and output, the algorithm will not generalize well on new data, which is called overfitting ans should be avoided[Lia16].

2.1 Decision Tree

A Decision Tree (DT) has a tree structure, in which internal nodes contain decisions based on input values, meaning a test for a specific range or feature, the edges contain value ranges based on the decisions, and leaves contain output values. The structure can also be seen in figure 2.1. Fürnkranz[Für08] describes the functioning of this ML algorithm. Starting from the root, the tree is traversed from the top and each decision determines which path is taken next until a leaf is reached determining the inference result. To create a decision tree, the input values of the training dataset are used to split the data up into the output categories as unevenly as possible in each step to separate the categories as much as possible. This is repeated until a certain size, depth or complexity maximum is reached and the final decision tree is determined[Für08].

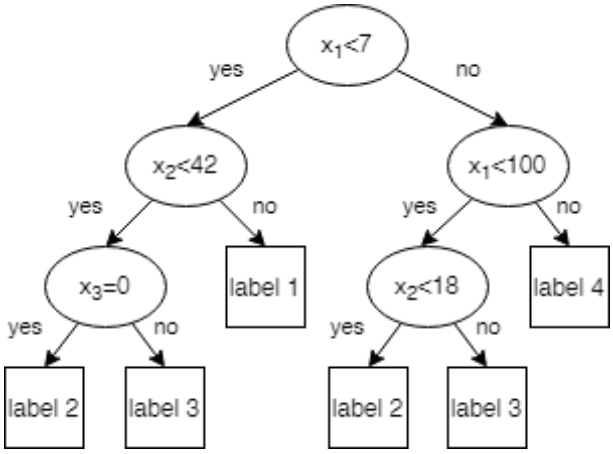

Figure 2.1: Decision tree with 4 output labels

2.2 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is based on the idea that a new input observation which is most similar to a specific prior input observation x is likely to also have a similar output as the output of x, which is an idea so natural that it can not easily be traced back to one specific inventor according to the lecture slides of Cosma Shalizi[Sha]. Normalization of the input values can help with making the distance function more meaningful. Later, different variations of this nearest neighbor idea were developed, one of them being KNN for which

not the single most similar input is used but rather the closest K data points. According to Shaliza[Sha], for classification, the most common output value for the K input values is used, for regression, the average counts. The vote can be based on the distance or a weighted average can be used. The closer the distance to the new observation, the more meaning a previous observation has. This approach of using K nearest neighbors and possibly using weights was already proposed by Cover and Hart in 1967[CH67]. For a KNN algorithm, there is no training phase, but all training data needs to be stored for inference. An example of KNN for binary classification of two-dimensional data without weights can be seen in figure 2.2. The new input is marked with a cross.

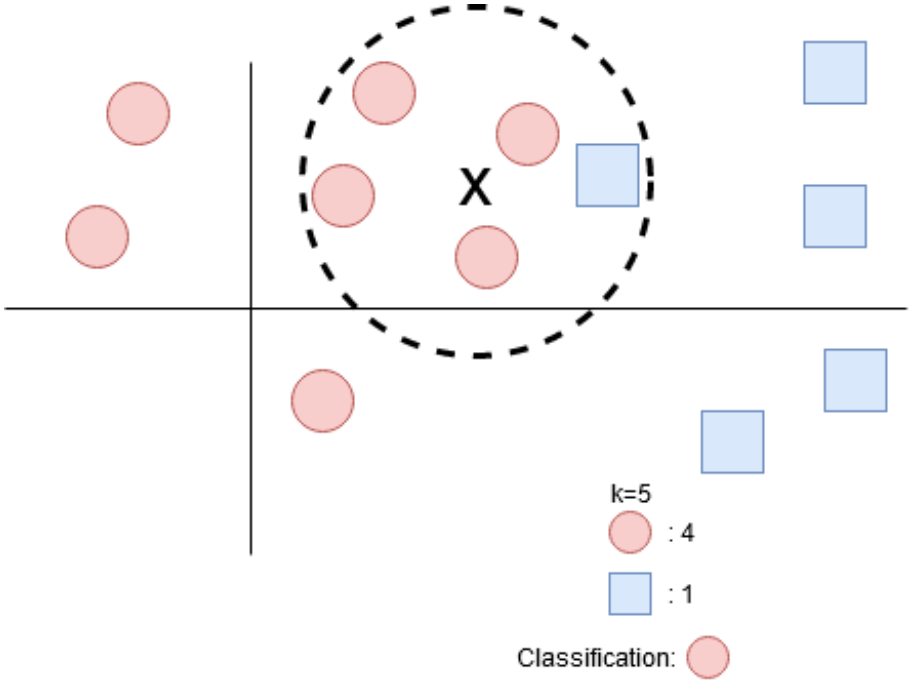

Figure 2.2: K-Nearest Neighbor algorithm with k=5 (based on [Sri18])

2.3 Support Vector Machine

Support Vector Machines(SVM) are designed for binary classification and were developed by Vapnik and colleagues at the AT&T Bell Laboratories in the 1990s[CV95]. For a set of training data with *N* input dimensions, a hyperplane in the N-dimensional space is calculated which acts as a boundary between the two classes for future inferences[CV95]. An example for two-dimensional data can be seen in figure 2.3 where the hyperplane and the maximized margin are shown. The power of an SVM can be further increased by using the "kernel trick" which uses M functions to map the N input values to a different M-dimensional feature space, M can be bigger than N[Ber03]. Since these functions can be non-linear, non-linear problems can be tackled whereas without the kernel trick only linear classification is possible[Ber03]. To use SVM for more than 2 categories, multiple SVMs are trained and all their results combined are used to find the right label[Mar20].

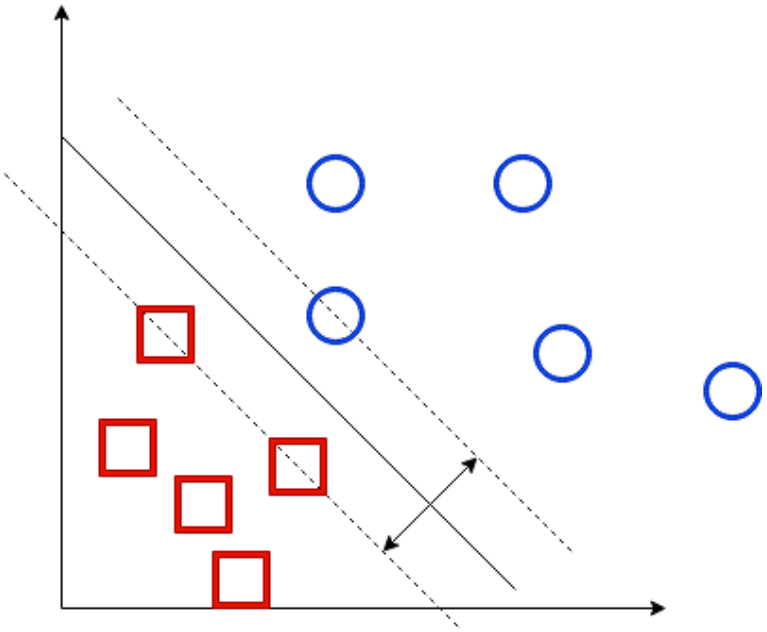

Figure 2.3: SVM for binary classification of two-dimensional data (based on [Bha])

2.4 Neural Network

Instead of thinking about how a machine could be able to learn from data, one can also think about how we as humans are able to learn from data. Our brain gives us the ability to think and is made up of about 10^{11} neurons[RN09]. To understand and mimic the behavior of a brain using a machine, the psychologist Frank Rosenblatt developed the perceptron model in 1958[Ros58]. A single perceptron can be used for binary classification. It has inputs that are multiplied by weights deciding their influence on the output. The sum of these weighted inputs minus a bias weight is then used for an activation function. For binary classification, a step function can be used that assigns one label for values below a certain threshold and the other label for values above the threshold[RN09]. Rosenblatt[Ros58] also described how to adjust the weights to improve the classification result. First, they are initialized with imperfect weights, often even random weights. If an error occurs, the weights are changed based on the extent they are to blame for the error which decreases future errors for the same input[Ros58]. The structure of a perceptron with three inputs is also shown in figure 2.4.
The power of a single perceptron is limited since it is only able to weigh the input through multiplication and without further logic, which e.g. makes an XOR-function impossible and limits the applicability to nonlinear problems[RN09]. As described, our brain consists of many neurons so a logical next step was to try to connect multiple perceptrons together in order to create multilayer perceptrons (MLP). Connecting them is relatively easy, the output of one perceptron is simply used as the input for the next one. The perceptrons form multiple layers starting with the input layer taking in the real input vales, followed by optional multiple hidden layers and ending in the output layer, whose output is the real output of the MLP[RN09]. The real challenge is how to adjust the many weights of the multiple perceptrons in the layers based on a single error value. The solution that is still widely used today was the development of the backpropagation algorithm. During the 1960s the general ideas were developed and finally applied to the MLP by Paul Werbos in 1974[Wer90].
Since the connection of perceptrons has some resemblance to our human brain, MLPs are also called artificial neural networks (ANN) or just neural networks (NN). The individual

perceptrons are also called neurons or nodes. After the development of the MLP, other forms of NN were also invented which makes the MLP just one case of neural networks. The case of an NN that has all possible connections between the neurons, like a MLP, is now often called a fully-connected or dense NN. If a NN has many layers this can also be called deep learning (DL)[Mic20].
In this thesis, dense NN will sometimes be described by the number of neurons in their layers. The first number represents the input layer and is equal to the number of input variables. This input layer has no real neurons and no activation functions. The following numbers represent the number of neurons in the hidden layers. The last number is the number of neurons in the output layer. An example would be "42-16-8-4" which is a dense NN with 42 input values, a first hidden layer with 16 neurons, a second hidden layer with 8 neurons, and an output layer with 4 neurons. The same notation was also used by Wu et al.[Wu+04]. An example NN with the structure 3-4-4-1 is also depicted in figure 2.5
Apart from dense NNs, two other types are also relevant for this thesis, convolutional neural networks and recurrent neural networks, and will be presented as well.

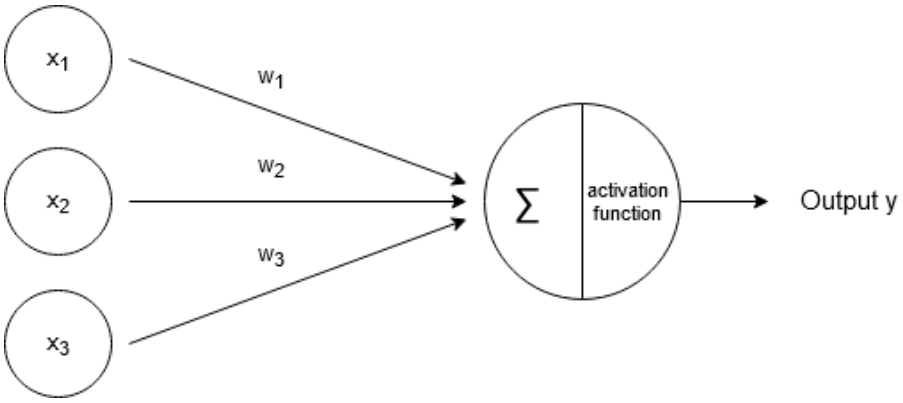

Figure 2.4: Perceptron with three inputs

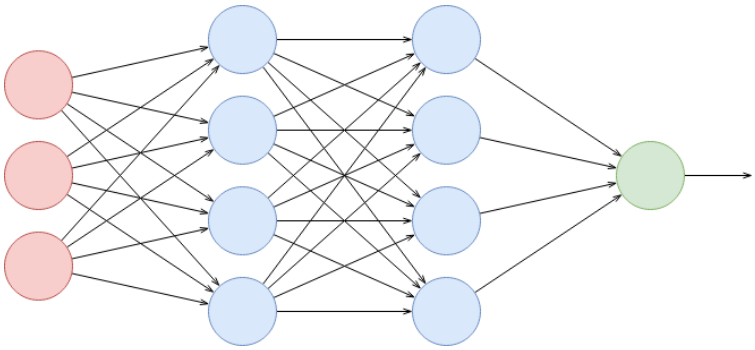

Figure 2.5: NN with the structure 3-4-4-1

2.4.1 Convolutional Neural Network

The input data for a NN can have a context between the input values, e.g. in a time series a value is a successor to one value and a predecessor to another one. A further example is images, where pixels can be neighbors. A dense NN does not know about these connections and has the possibility to come up with dependencies between any combination of inputs. In contrast, Hubel and Wiesel already showed that the brain of animals has neurons that

respond to regions in the vision in 1959[HW59]. If this connection between input values would be respected in the structure of the NN, the other connections between less related input parameters, e.g. pixels that are far away from each other in an image, could be partially left out. If unnecessary weights are left out, the chance of overfitting could also be reduced[Lia16]. Kunihiko Fukushima was inspired by the findings of Hubel and Wisel and tried to incorporate a similar structure into a neural network in 1980[Fuk80]. His "Neocognitron" network has two different kinds of layers which he calls S-cells and C-cells. The S-cells use a local part of the input and find certain features of this part that are relevant for the task. They only have connections to these input values and not to all values from the previous layer, as would be the case with dense NN. The step of looking for the same feature in different local parts of the image can be implemented as a convolution operation. They also gave name to NNs featuring this idea which are called convolutional neural networks (CNN) which feature these operations inside of convolutional layers[Bat17]. A kernel for the convolution contains the learned weights that are established during training[Bat17]. The next layer in the Neocognitron is a layer of C-cells which try to deal with shifts of the input values, meaning a shift of the image[Fuk80]. If the task is to recognize an object in an image, the object should be recognized no matter where it is shifted to. To do so, the C-cells have fixed inputs from the previous S-cells which detected a certain feature. In this way, the results of multiple S-cells from different parts of the image which detect the same feature are combined which makes it possible to detect this feature even when it is shifted. Such a layer is called a downsampling layer[Fuk80]. Fukushima used spatial averaging which is nowadays replaced by pooling, max-pooling being the most common one, which just outputs the maximal result for each regional part of the previous layer, e.g. each 4*4 area[Bat17]. With each time the two layers alternate, the observed area is increased until features are detected globally and the whole image is taken into account for the final output[Fuk80].
Convolutional neural networks were later trained with backpropagation like dense NN so global instead of local optimization is possible. A first application of CNN for a big dataset was hand-writing recognition using a CNN designed by LeCun et al.[LeC+90] in 1989 which contained convolutional layers whose neurons job is "equivalent to a convolution with a small size kernel, followed by a squashing function" and downsampling layers which perform "local averaging and a subsampling"[LeC+90].
The popularity of CNNs greatly increased when, starting in 2006, GPUs instead of only CPUs were used to accelerate the performance[CPS06]. Today, CNNs are a very common NN architecture for ML tasks with images or videos as input data and they have been shown to perform well in areas like face recognition, pose estimation, activity recognition, or different forms of image classification and object recognition[Kha+20].

2.4.2 Recurrent Neural Network

Dense Neural Networks and Convolutional Neural Networks both work on an input X based on which they try to calculate an approximation of the output Y. The output of one neuron is sent to the next layer, always in the same direction and data is never flowing backward. Hence, NNs working in this way are called feed-forward networks[RN09]. For many ML tasks, the input is a sequence of values X_t, e.g. a series of pictures for a video input of a self-driving car. In such a scenario, if a red car is detected to be driving in front for the last couple of frames, a blurry red object in the next frame is also likely to still be the same red car. This relationship between the inputs and also the output values of a series of inferences is not modeled by a normal feed-forward NN that would only take in the new camera picture of the blurry red car. A Recurrent Neural Networks (RNN) incorporates this relationship into its NN structure by allowing back connections that pass information from one inference cycle

to the next, enabling to model a system with state. The structure of the RNN can be cyclical instead of only acyclical like a feed-forward NN[RN09].
The first RNNs were developed in the 1980s, one of them being the Hopfield Network which forms an associative memory and can recognize patterns it has seen before, even when only part of the pattern is visible or parts of the input data are corrupted in comparison to the training data[Hop84]. Later in 1990s, the Long Short-Term Memory (LSTM) architecture was developed by Sepp Hochreiter and Jürgen Schmidhuber[HS97]. The state passed on from iteration to iteration in a RNN can have problems with situations where a context mentioned many iterations ago is important since weights $w<1$ lead to a decrease in the absolute value of the state[Gee20]. The LSTM architecture has ways to forget unimportant parts of the state and add new information to it[Gee20]. If both are not needed, the information is kept as it is and the long-term memory is not altered[Gee20]. RNNs like the LSTM are commonly used today for natural language tasks, ML applications with video input or audio data[Gee20].

Chapter 3

Related Work

This chapter analyzes previous work related to the topic of this thesis which combines three elements.
Firstly, the topic of machine learning (ML) which is a subset of artificial intelligence (AI). Since related work does not need to be completely identical, also work on AI in general will be considered.
The second topic of this thesis is the usage of microcontrollers. The term describes a vast group of small integrated circuits used in embedded devices. A microcontroller contains a microprocessor, memory and other associated circuits. It often performs one specific task for one specific application and in order to do so controls some or all of the functions of an electronic device. Embedded devices in general are specialized devices used for some specific purposes and which are usually embedded within another object or larger system, hence the naming[Tecb]. Since some sources only use the term "embedded" even when the embedded system contains a microcontroller, the search for related work includes both "microcontroller" and "embedded" as keywords.
The third topic is the automotive industry. Finding related work for this topic can require multiple keywords since sources might talk about cars which themselves are the main product of the automotive industry and hence very much related. After reading review papers about the automotive industry, especially the work of Borrego-Carazo et al.[Bor+20], the keywords used for this third topic were selected as being "automotive", "vehicle", and "car".

3.1 Search methodology

Since the focus of the search for related work is a combination of all three topics, the search terms are also a combination of three described keywords. Since "Artificial Intelligence" led to few relevant results, it was not used in combination with embedded. All used search terms are:

- Machine learning microcontroller automotive
- Machine learning microcontroller vehicle
- Machine learning microcontroller car
- Artificial intelligence microcontroller automotive
- Artificial intelligence microcontroller vehicle
- Artificial intelligence microcontroller car
- Machine learning embedded automotive
- Machine learning embedded vehicle
- Machine learning embedded car

To find related work, these search terms were used in popular search engines with a focus on academic work. The selected ones are "Google search", "Google scholar", "Microsoft Academic", "OPAC", and "Bielefeld Academic Search Engine (BASE)". For each search term and search engine, the first 20 results were analyzed for possibly related work.

The results can be grouped into matching search topics regarding the three described groups.
A number of results deal with microcontrollers and some form of AI but are not directly related to the automotive industry but rather about the combination of those two things in general. These results are still somewhat relevant since using AI on microcontrollers is a relatively new topic and results in other fields may still include relevant findings for automotive applications e.g. regarding the limitations of microcontrollers or helpful ML libraries.
Other examples deal with general applications of AI in the automotive industry but do not use microcontrollers. These findings are less related to the work of this thesis and will not be included in the section of related work. However, a topic that has not been tried yet on a microcontroller could possibly be implemented on such a device. Therefore, AI in the automotive industry in general will be discussed in the following chapter called "Use Case Analysis".
Some sources are about microcontrollers in the automotive industry but do not incorporate any AI techniques. Since this is deemed to be of little relevance to this thesis, these results will not be included.
Of course, the most relevant findings are the ones matching all three topics. These sources can differ widely in the used hardware. They range from implementations on expensive Nvidia Drive or Jetson development boards which cost thousands of dollars to slow single-core microcontrollers that are available for just a few dollars. Therefore, these sources will also be grouped by used hardware to get an understanding of the required computing power for different automotive applications.

3.2 ML on microcontrollers in general

Machine Learning on microcontrollers and embedded systems has been an area of research for more than ten years based on the found sources. While some work specifically targets applications in the automotive industry, many sources talk about the idea of ML on these platforms in general. These sources are still very valuable for this thesis, since they might use similar hardware, describe helpful workflows and ideas, or their examples might be related use cases in the automotive world.
Before starting to implement a solution using a microcontroller, it is good to understand the benefits and drawbacks of such an approach. Articles from the last few years show that ML on microcontrollers is gaining popularity and the tools to implement it are improving and increasing in number. A review on Edge Machine Learning by Massimo Merenda et al. shows that the general interest in edge computing has been increasing during the last years, e.g. the Google trends interest score rose from below 10 in 2016 to more than 100 in 2020[MPI20]. Their work is focused on the internet of things and not the automotive industry.
There are also multiple relatively short (web) articles describing possible reasons for increased interest in ML on embedded systems, e.g. by Jacob Bingo of Design News[Ben18], Sebastian Gerstl of Intelligent Mobility Xperience[Ger19], or Andreas Burkert for ATZelectronics worldwide[Bur19]. All mentioned articles talk about reasons to use ML on microcontrollers, some possible options to do so but do not describe real examples in contrast to this

thesis.
To implement ML on a microcontroller, a good ML library supporting these devices is very helpful. An option available since 2018 is CMSIS-NN and ARM has published an overview of compatible devices and their performance with different NNs[SL19]. A similar paper is the work of Fouad Sakr et al.[Sak+20] who have analyzed different hardware options for ML on microcontrollers. They evaluated the performance of six STM32 boards for different ML algorithms, namely NN, SVM, DT, and KNN. Their work uses the X-Cube AI package from STM to deploy these algorithms. This thesis will also use an STM32 microcontroller but will additionally focus on applications in the automotive industry instead of picking datasets from multiple areas[Sak+20].
Since microcontrollers are restrictive in computational power and memory size, researchers are also looking at ways to reduce the computation cost and memory size requirements for ML algorithms. The smallest NN used in the paper published by ARM with CMSIS-NN is 80 KB[SL19], while Sakr et al. deployed SVMs, DT and KNN on a microcontroller with only 32 KB of memory[Sak+20]. However, other papers go even further, e.g. down to 2 KB in the work by Ashish Kumar et al.[KGV17]. They test different ML algorithms on an Arduino Uno with only 2 KB of available SRAM and develop a decision-tree algorithm called Bonsai which creates a "single, shallow, sparse tree" that also involves non-linear predictions in all nodes. This showcases that shrinking down ML models can make them fit into the RAM of inexpensive microcontrollers[KGV17].

3.3 Applications of ML on embedded devices in the automotive industry

As described, machine learning on embedded devices has been rising in popularity. Therefore, it is not surprising that there are already some examples for problems in the automotive industry. They vary in the used hardware, its computational power and consequently the computational difficulty of the problems it can tackle. First, papers that try to reduce the computational complexity or discuss different approaches without testing them on a specific embedded device will be discussed. Afterwards, examples deployed on real embedded hardware will be analyzed and ranked based on the computational resources used.

3.3.1 Related work not deployed on embedded hardware

Before ML could be deployed on embedded devices for applications in the automotive industry, papers looked at the computational costs of ML algorithms and how to possibly reduce them. One early example is the work by Davide Anguita et al.[Ang+07] who already looked at ways to minimize the computational cost of support vector machines in 2007. Unlike this thesis, they did not actually test a real-world example on an embedded platform but rather just looked at ways to make it possible while still performing the test on a PC. The tackled problem is a dataset by Daimler-Chrysler for binary classification of gray-scale images with a resolution of 36*18 pixels. One half of the images contains a pedestrian crossing the road while the other half does not and just contains an empty road. To use a SVM, the values for the kernel and coefficients are needed. Anguita et al. tested how the accuracy changes when the number of bits representing these values is reduced to decrease the total size of the SVM's weights[Ang+07].
A more recent paper by Jelena Kocić et al.[KJD19] proposes a NN architecture that they call "J-Net" that aims to be lightweight so it can be deployed on embedded automotive platforms.

They use a self-driving car simulator to test the NN performance and compare it to two other NN architectures that were previously developed and applied for self-driving. Like the previous paper, this evaluation was done on a PC and not on an embedded device. All three NN were able to steer the car and "differences between autonomous driving using different models were notable, but not large"[KJD19]. In size and computational cost, however, there was a large difference. Compared to the second-best option, J-Net needs less than half as many arithmetic operations and also less than half the amount of memory, 1.8 MB[KJD19].

3.3.2 Related work deployed on embedded hardware

Related work deploying ML algorithms for tasks in the automotive industry is most related to this thesis. Such papers will be compared and the results will also be summarized using the table 3.1 at the end of this chapter. As was shown in the related work which was not focused on automotive applications, some ML algorithms can be run on very limited hardware, e.g. with less than 2 KB of memory required[KGV17]. However, for more complex computer vision problems with large NNs, faster embedded systems are necessary. These embedded systems can be grouped based on their performance. First, papers using systems which include fast GPUs will be presented, e.g. Nvidia Jetson boards. Afterwards, problems solved with devices running multiple CPU cores like Raspberry Pi will be analyzed. Finally, microcontrollers with just a single and usually slow core will be presented. The results will also be summarized in the table 3.1 at the end of this chapter.

Nvidia Systems with GPU The fastest system used in a related paper is the Nvidia Drive PX2 board in the research of Shinpei Kato et al.[Kat+18] who developed "Autoware on Board". The original Autoware is an open-source set "of software packages and libraries required for autonomous vehicles"[Kat+18] which is extended so it can be run on ARM-based devices and the embedded GPUs of the Nvidia Drive PX2. This Nvidia product contains 12 CPU cores and a very powerful Nvidia Pascal based GPU with a total theoretical performance of 8 TFLOPS which is significantly faster than most notebooks[Har16]. This performance also comes at a big price, initial prices for automotive companies were about $15000 per unit, and a big power consumption of up to 250W[Lam16]. The Autoware system includes multiple ML components including Caffe as a deep learning framework for object detection. The performance of the Nvidia embedded system was compared to a high-end laptop computer which was running a Nvidia GTX 980 GPU and the Intel i7-6700K CPU for different tasks of the Autoware application. "The worst execution time observed on the DRIVE PX2 platform was about three times as much as that on the laptop computer"[Kat+18] which was deemed acceptable for real autonomous vehicles. This shows that some real-time problems require significant computing power which can be delivered by power-hungry hardware including fast GPUs.
Nvidia also offers cheaper embedded boards in the Jetson series which also combine CPU cores with GPU ones. The first board called Jetson TK1 featuring the Tegra K1 SoC was released in 2014 and was followed by the Jetson TX1 and Jetson TX2.
These Nvidia Jetson boards were also used in related work for machine learning applications in the automotive industry. Dendaluce Jahnke et al. used the Nvidia Jetson TK1 for torque vectoring optimization of a multi-motor electric vehicle in 2018[Den+19]. A NN was used to control the four torque points, one for each wheel. 1024 control options had to be evaluated so the best one could be chosen with a total maximum cycle time of 5 ms, hence only

4.88 μs for each of the 1024 inferences. They evaluated two dense NN with different number of inputs, but the same internal structure (16-32-16-8-4 and 24-32-16-8-4), resulting in 1284/1556 parameters and 2468/2996 arithmetic operations. The slightly larger of the two relatively small NN "still was successful even under the most challenging situations in what respects to the dynamical behavior of the predicted variables"[Den+19] and the inference only took 0.040 μs per iteration, which is far lower than the 4.88 μs requirement[Den+19].
Guido Borghi et al.[Bor+17] also used the Nvidia Jetson TK1 and furthermore compared it to its predecessor, the Jetson TX1 in 2017. The task was to estimate the driver's head pose which helps with identifying the driver's behavior, especially to monitor their attention. They used a RNN incorporating 5 convolutional layers with the dataset containing 64*64 pixel depth frames and achieved "good accuracy"[Bor+17] with angle errors of up to $7.5 \pm 6.3°$. Since a lack of attention should be recognized quickly, the time per cycle including inference is also important which was 53.3ms for the Jetson TK1 and 32.9ms for the Jetson TX1 which is also deemed a "good" result[Bor+17].
A third paper using the Nvidia Jetson board which is related to this thesis was published by Nashwan Adnan Othman et al.[OAK19] in 2019. The task is to detect when a driver has closed his eyes for longer than typical which can be a sign of drowsiness. The system has two steps. First, the face is detected using a SVM, and in a second step, the coordinates of different parts of the eyes are used to calculate whether they are open or closed. The researchers think that the system has a "high accuracy"[OAK19] since it passed all of their 10 test demonstrations without errors. Using the Nvidia Jetson TX2, they were able to perform 16-18 cycles per second, meaning that one cycle took about 55-63 ms[OAK19].

Multi-core CPU devices Another option for embedded devices are Raspberry Pi boards which are relatively inexpensive at prices of about $35-$40. The first board was released in 2013 and currently, Raspberry Pi 4 is the newest model. These boards feature an ARM CPU with up to 4 cores and a Broadcom GPU with a total power draw of up to 6.25W for the newest Raspberry Pi 4 [Rasa] with a theoretical performance of 13.5 GFLOPs[The]. In the last described paper, Othman et al.[OAK19] also compared the performance of the Nvidia Jetson TX2 to the one of a Raspberry Pi 3 which was only able to achieve 2-3 cycles per second which is too slow for the task.
However, other less complex tasks in the automotive industry can still be solved using these embedded devices. One example is the work of Fakhar et al.[Fak+19] who deployed a KNN algorithm on a Raspberry Pi 3 for automatic number plate recognition. In the testing, the system achieved an 85% accuracy and the recognition took 2 to 3 seconds. Depending on the task this speed can still be enough, e.g. to recognize which car is parked in a parking lot where there is plenty of time to recognize the number plate. In other situations, e.g. to identify cars on a busy road, the performance of the Raspberry Pi would not be sufficient.
A similar device to the Raspberry Pi boards is the Freescale Cortex A9 LMX6Q which also has a quad-core ARM processor and with performance similar to the one of a Raspberry Pi 2[Rasb]. This device was used in the work of Ju-Seok Shin et al.[Shi+17] to detect vehicles. The system uses small decision trees to find possible cars and in the next step, a NN is used to decide whether the candidates actually contain vehicles or not. The NN works on 32*32px grayscale images of the candidates with a structure of 512 nodes in the first layer, 32 in the second, and 2 in the last one. Accuracy depending on the test sequence is 85%-99% with an average speed of 14.89FPS, meaning about 67 ms per iteration[Shi+17].

Single-core microcontrollers While Raspberry Pi boards are already pretty affordable, single-core microcontrollers can be even cheaper and also smaller in size. Additionally, they usually need less energy. Such a device will also be used for examples in this thesis, an STM32F746 with a single core running at up to 216 MHz. With these small boards there come big limitations regarding the memory size and CPU performance. However, they can still run ML algorithms to perform tasks in the automotive context. One example is the work by Umar Farooq et al.[Far+10] who use an AT89C52, which is a low-power 8-bit microcontroller running at up to 24 MHz. The task is to navigate an autonomous vehicle to a goal location. The job of the AT89C52 microcontroller is to detect obstacles using four ultrasonic sensors in which case going around the obstacle instead of directly to the goal takes priority. The microcontroller runs a very small NN with 3 neurons in the input layer, 6 in the hidden layer, and 4 in the output one. To reduce the computation cost even further, the sigmoid activation function of the neurons is approximated through piecewise linear functions[Far+10].
Another related work using a small NN on a microcontroller has been developed by Raj Kumar et al.[KK12] who try to calculate the state of charge of a car battery using a PIC 16F877 microcontroller which has a single core running at up to 20 MHz. The NN has five neurons in the input layer for the five measured values ("terminal voltage, current drawn, internal resistance, temperature and time"[KK12]), 11 neurons each in two hidden layers, and one neuron for the output. The researchers think that the system achieves reasonable accuracy, but it needs 4.8 seconds to calculate the state of charge[KK12]. Depending on the usage this can still be sufficient if e.g. the last value is always stored and just updated every 4.8 seconds. During this time-frame, the state of charge of a car battery usually does not undergo big changes.
Small decision trees with shallow depths are also a candidate for inexpensive single-core microcontrollers. Such an approach is employed by Kyle Ying et al.[Yin+15] and implemented on a Texas Instruments MSP430 microcontroller with a single core with a maximum speed of only 16 MHz. They show a proof-of-concept for recognizing different kinds of vehicles driving on a track based on data from magnetometer sensors placed in or next to the road. To demonstrate the feasibility of such an approach, they use seven different RC cars instead of real passenger vehicles. The sensors provide twelve features out of which the three most important ones were selected to reduce computational cost. The resulting decision tree has a maximum depth of only 6 and in total only 10 split conditions based directly on the 3 selected values showcasing the relatively low complexity of the problem. The accuracy was about 98% and features are extracted and evaluated at 75 Hz, meaning about 13 ms per cycle[Yin+15].

Related work deployed on embedded hardware				
Source	Hardware	Problem description	ML Algorithm	Time per iteration
Nvidia Systems with GPU				
[Kat+18]	Nvidia Drive PX2	Control of autonomous vehicle	Multiple, including NN	Not specified
[Den+19]	Nvidia Jetson TK1	Torque vectoring optimization of a multi-motor electric vehicle	NN	4.88 μs requirement, 0.040 μs achieved
[Bor+17]	Nvidia Jetson TK1 and TX1	Head pose estimation	RNN	53.3 ms for TK1 and 32.9 ms for TX1
[OAK19]	Nvidia Jetson TX2 and Raspberry Pi 3	Drowsiness detection based on closed eyes	SVM	~55-63 ms (Nvidia Jetson) and ~333-500 ms (Raspberry Pi, insufficient)
Multi-core CPU devices				
[Fak+19]	Raspberry Pi 3	Number plate recognition	KNN	2-3s
[Shi+17]	Freescale Cortex A9 LMX6Q	Vehicle detection	DT and NN	67 ms
Single-core microcontrollers				
[Far+10]	AT89C52 microcontroller	Obstacle detection for autonomous navigation	NN	not specified
[KK12]	PIC 16F877 microcontroller	State of charge of car battery	NN	4.8 s
[Yin+15]	Texas Instruments MSP430	Vehicle identification using magnetometers	DT	13 ms

Table 3.1: Related work deployed on embedded hardware

Chapter 4

Use Case Analysis

Before implementing machine learning on a microcontroller, a use case needs to be found. Since the performance of a microcontroller is limited, choosing the right problem to be solved is essential for a successful project. This chapter describes the different ways machine learning can be used in the automotive industry and whether the usage of a microcontroller might be feasible. Later chapters will then use the scenarios with sensible use cases for machine learning on a microcontroller for the demonstration implementations.

4.1 Use case groups of AI in the automotive industry

Artificial Intelligence has been used in the automotive industry for many years and there are already many known use cases. Multiple papers have described them and this section will try to summarize their work and group similar applications into use case categories. Each category will then be evaluated for specific scenarios which might benefit from the usage of AI on microcontrollers. Since the performance is limited, not all problems can be solved with this hardware.
The paper "Intelligent systems in the automotive industry: applications and trends" [GRF07] and the report "Accelerating automotive's AI transformation: How driving AI enterprise-wide can turbo-charge organizational value" [Win+19] give a good overview of AI in the automotive industry. Several areas of use-cases can be identified.
AI is frequently used in **car development and production** e.g. in assembly process planning, vehicle production sequencing, development of new parts, engine manufacturing, machine diagnostics or supply chain management[GRF07]. The finished car then might use AI for speed control[GRF07], emission control, different levels of autonomous driving and assisted driving features, or speech control for the infotainment system[Win+19] which this paper will group together as **driver experience**.
Since the finished product can always have problems, **fault diagnosis** is important and can benefit from the usage of AI[GRF07]. Even better than detecting a problem would be to predict the occurrence which is called **predictive maintenance** [Win+19]. A similar idea is the use of AI for warranty claims "to prevent future warranty issues and recalls"[GRF07].
Apart from producing and selling cars, many automotive companies also offer **mobility services** where e.g. AI projects developed at Michelin can be used for improved fleet management or car sharing services who use AI to optimize their service[Win+19]. The different use case groups and the described examples for them are also depicted in figure 4.1

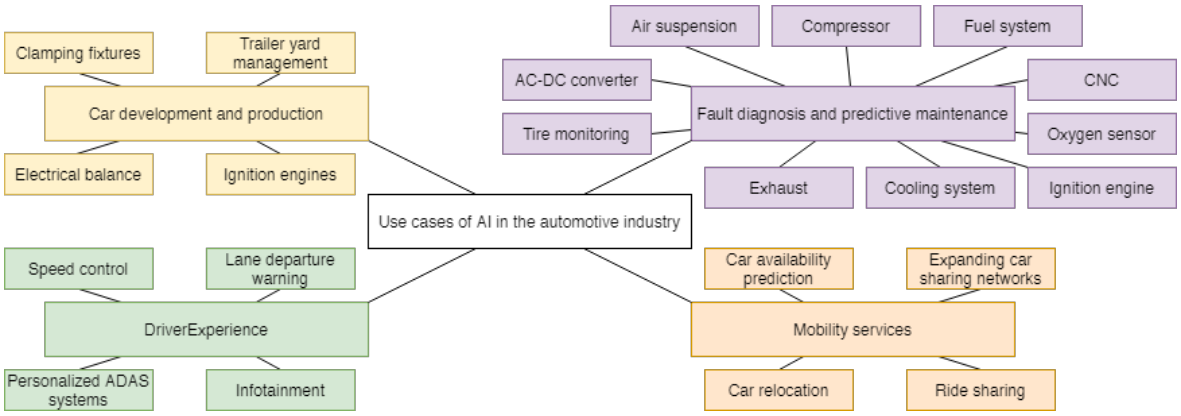

Figure 4.1: Use cases of AI in the automotive industry

4.2 AI for car development and production

Already during the development of new cars and their components, AI is being used. Systems can be modeled with the help of AI instead of only relying on computing intensive simulations. This can make the development of new parts quicker and was recently used to e.g. reason about the vehicle's electrical balance, meaning the balance between power supply and consumption in a car[HYC18].
The developed components can then be produced and AI can also be beneficial in this step. Fraunhofer IWU is researching how AI can help develop more flexible clamping fixtures which hold elements in place during production which would make it possible to reuse them for multiple different car models[FRP18].
Calibrating components can also use AI techniques, spark-ignition engines are one possible example. These components have gained more and more degrees of freedom in the last years since the number of input and output values rises to e.g. comply with more restrictive emission standards[Tur+16]. With the traditional approach of look-up tables and trying out every possible combination, this task would be "virtually impossible"[Tur+16]. The relationship can often be non-linear and therefore NNs are also a good option for modeling these systems.
This technique was for example tried by Wu et al.[Wu+04] in 2004 who used a NN to analyze the airflow rate of a 2.4 liter engine. The degrees of freedom were lower than common today, the system had just 4 input values and the only output calculated is the required fuel injection amount. The researchers evaluated different dense models with varying amounts of hidden layers and neurons. They found that a model with the four inputs, two hidden layers of eight neurons each and the one output value (4-8-8-1) worked best[Wu+04]. Small neural networks consisting of few dense layers and neurons will be used in the examples of this thesis and are a good fit for machine learning on microcontrollers. This shows that valve controls with few degrees of freedom would also be a possible application for microcontrollers.
During production, the automotive industry has many other applications of AI. The logistics of production can benefit from automation. One such example is the use of cameras to automatically detect and identify trailers in a trailer yard management system as described in "Artificial Intelligence and Deep Learning Applications for Automotive Manufacturing" by Luckow et al.[Luc+18]. The developed system has two phases. First, a bar-code is detected using a NN model which does not differentiate between different bar-codes. Afterwards, the part of the image containing the code is sent to a server where the code is identified with a larger second model. The first stage can be run on mobile devices. e.g an iPhone 7 with 23 hz, since the NN model is not as demanding and complex. Therefore, the examples of this

thesis will take a look at small computer vision problems which might also be feasible with a microcontroller. The second phase, however, required significantly more computing and thus was only deployed on high end GPUs like Nvidia Tesla K40, P100 or V100[Luc+18]. Complex computer vision problems similar to the second phase thus will not be feasible on the STM32F746 microcontroller used in this thesis since the performance of a modern GPU and a single-core microcontroller are in two entirely different ballparks.

4.3 AI for driver experience

When interacting with the car, the driver will notice the usage of AI in the automotive industry most directly. Many modern cars have advanced driver assistance systems (ADAS) which can incorporate AI. They utilize sensor data to help the driver by detecting driving errors so the driver can be warned or the system can intervene itself. ADAS can also automate certain tasks altogether, e.g. the control of the car's lights. Currently, ADAS help the driver, but they have to act according to specific rules. These models and functions between sensor values and ADAS behavior can be developed by human experts. An alternative would be to approximate these functions based on data from a real driver who sort of acts as a role model for how to behave in different road scenarios. One such example is the work of Zhang et al.[Zha+18] who implement a deep learning algorithm for autonomous speed control. Their system includes a three layer fully connected NN, which is rather small for deep learning, but the exact parameters were not included in their paper.
Machine learning algorithms require input values to perform inference. ADAS can benefit from data about the road condition or the vehicle state (e.g. speed, acceleration, suspension compression, direction, or slip). Unfortunately, not all cars have sensors for all these metrics. Kim et al.[Kim+17] developed a system that augments missing sensor information. The car includes sensors to measure "the longitudinal and lateral accelerations, the yaw rate, and the wheel speed,[...] the steering wheel angle and the throttle position"[Kim+17]. Based on these available values, other values are then augmented including the road condition (classified as non-slippery or slippery), "lateral velocity, the side slip angle, the lateral tire force, the roll rate, the suspension spring compression, and the heading direction"[Kim+17]. The system is good at detecting slippery roads, correctly classifying 51 out of the 52 test cases. The other augmented metrics also achieved a "high level of accuracy"[Kim+17] according to the researchers. In a second step, the real sensor values and the augmented data are used to predict whether the driver wants to change lanes or is leaving the lane accidentally, a common ADAS problem needed e.g. for lane departure warning systems. Compared to a system only using the sensor data without augmentation, "the driver's intention for lane change can be detected more accurately using both measured and estimated data"[Kim+17].

As described, ADAS currently often relies on one specific model which is independent of the preferences of the driver. An experienced and fit driver might find them too intrusive and would like activation thresholds to be higher. An older and less safe driver might, however, prefer more active systems since they aren't trusting their own actions as much. Unfortunately, most current ADAS try to model the average driver and base activation thresholds on this average behavior[HW17]. Recently, some researchers have tried to personalize ADAS. One such example is the work of Vallon et al.[Val+17] who have tried to develop a personalized ADAS for lane changes. They use a SVM to determine whether a lane change should be performed to overtake a slower vehicle driving in front. This result is then used for a motion planner that either tries to stay in the lane or perform the lane change maneuver. To add personalization, 25 lane changes of two different drivers were recorded. A SVM was then set up

for each driver to match their driving behavior more closely. If the ADAS should behave more like the actual driver of the car, this approach is beneficial since the matching SVMs have "average normalized errors of 6% and 7% in test sets for Drivers A and B"[HW17], whereas applying the SVMs to the wrong driver (e.g. SVM A to driver B) makes them perform "only half as good at correctly classifying the testing data"[HW17]. This shows that personalization of ADAS makes it possible to understand the driver better.

Personalization can also be applied to a car's infotainment system. Here, AI is already used for navigation which might also include traffic prediction[Teca]. With personalization, the infotainment system can learn from the driver's behavior and make suggestions for possible trips they might want to make. These suggestions can also be made for other infotainment features like radio stations or music from streaming services.
Gaffar and Kouchak tested the benefits of such a system that includes recommendations for the infotainment UI[GK16]. Their goal was to minimize the time the driver needs to interact with the UI since this can be a distraction which should be avoided when driving a vehicle because distractions are one of the main causes of accidents. With a 95% confidence level, the researchers can statistically "say that response time with customized user interface is better"[GK16] and "the number of errors [...] is less"[GK16]. Therefore, such a personalization can be beneficial and make the infotainment system less distracting and add to the driver experience.
In the future, the driver might even become a passenger since the car might completely drive itself using the help of AI.

4.4 AI for fault diagnosis and predictive maintenance

During its lifetime, a vehicle can have problems which can lead to worse or no system performance, less stability or might even be a risk for the driver which makes fault detection an important topic in the automotive industry[Yu+12]. Subsequently analyzing the fault size or its location is called fault diagnosis[Yu+12]. Accurate and early fault detection and diagnosis can prevent further damage to the system and therefore reduce the severity of the damage and subsequently caused faults[XIE+18]. It increases safety and up-time while also reducing repair time and cost[Jun19].
Predictive maintenance, also called prognostics, goes one step further and tries to stop a fault from happening altogether. Based on the current condition of the vehicle, the system tries to predict components that are likely to fail[Sha+18].
Since vehicles are moving machines, continuous remote monitoring via a network connection is difficult[Pir+19]. The cost of wireless communication, bandwidth limitations, low network coverage or traveling between countries can lead to problems when trying to perform fault diagnosis or predictive maintenance in the cloud[Pry+15]. Hence it makes sense to use microcontrollers for these use cases, if possible since computation can be performed directly in the vehicle which removes the need for a network connection.
AI is often used for fault diagnosis and predictive maintenance for which some examples will now be discussed, none of which deploy the approach of using AI on a microcontroller.
Pirasteh et al.[Pir+19] analyze diagnostic trouble codes from 30.000 Volvo trucks to infer a future failure of an air suspension component and a powertrain component within a prediction window (predictive maintenance). The paper uses a random forest algorithm with a maximum tree depth of 10 and 100 trees.
A CNC machine of Fiat Power Train Technologies is used in another paper by Einabadi et

al.[EBE19] and the remaining useful life of 24 parts is predicted using a NN. With these predictions, the maintenance cost is minimized and big downtime costs are avoided which is an example of predictive maintenance[EBE19].
Researchers from the Polytechnic of Turin developed a system that identifies degradation of oxygen sensors in the exhaust system of General Motors cars[Gio+18]. Predictions are made on a dedicated server running a NN instead of using a microcontroller directly in the vehicle[Gio+18].
Data on vehicle compressors failures in 65000 Volvo trucks is used in a paper from 2015 to showcase the benefits of a predictive maintenance solution based on a random forest classifier[Pry+15]. Using this solution results in a net profit but the calculations depend on estimated costs for repairs with "large uncertainties"[Pry+15].
Fault detection based on vibration data of a gasoline engine using a NN is described by Ahmed et al.[Ahm+15] and an accuracy of 97% is achieved. Their work proposes to use such a system at a dealership or assembly plant but does not mention the possibility to deploy it inside the car with a microcontroller.
A NN is trained to detect valve damage in an internal combustion engine based on acoustic data by Jafari et al.[JMB14] and achieves a 92% accuracy. The used NN is very small, consisting of only 17 densely connected nodes.
The performance of a random forest and a support vector machine were compared by Daniel Jung in 2019 regarding faults in a Volvo internal combustion engine that are to be detected. The inference is done on a computer running Matlab[Jun19].
Fault diagnosis for an AC-DC Converter of an electric vehicle's drive train is implemented by Moosavi et al.[Moo+16] using a NN. The network is small consisting of only three layers and the training time was a mere 39 seconds. Performance is deemed to be "acceptable to implement the trained ANN in experimental work"[Moo+16].
Xie et al.[XIE+18] evaluated a fault diagnosis system for 5 fault types of a Hyundai car's engine. They use a NN for this task with only 32 densely connected neurons implemented in Matlab and achieves an accuracy of 98%.
The data of 70 Toyota Corolla cars is used by Shafi et al.[Sha+18] for vehicle monitoring and predictive maintenance for the ignition system, the exhaust system, the fuel system, and the cooling system. The paper compares the machine learning techniques decision tree, support vector machine, k nearest neighbors, and random forest all running on a central server the system needs to communicate with. Accuracies of >96% are achieved using the Support vector machine for each of the 4 systems[Sha+18].
To research the possibility of using a microcontroller instead of a PC or a central server, the authors of these 10 papers were contacted and asked these questions:

- Is it possible to get the dataset you used for training and testing? Alternatively, do you know of any similar and publicly available datasets for fault detection and predictive maintenance, preferably in the automotive industry?
- Is it possible to get the code you used?
- Since resources are very limited on microcontrollers, small size is essential. What is the approximate total size of the weights/parameters of your AI algorithm?
- What is the longest time inference might take for the use case to still make sense?
- Do you think this is doable using a microcontroller with a single core, 216Mhz and 340 Kbytes of RAM?

Unfortunately, out of the 10 papers only 2 authors responded. Sławomir Nowaczyk could neither share the code nor the dataset since both contain "a lot of sensitive information". Regarding the practicability of the ideas, he responds with "both memory and computational power

of a microcontroller should be enough to do reasoning" as described in the paper[Pir+19]. Daniel Jung responded with a lot of helpful information. He sent the Matlab code of his paper and the used dataset. Regarding the idea of implementing inference on a microcontroller he thinks: "I have not considered on-line computational aspects of my code in a microcontroller. The main problem that I considered in the paper is fault isolation which is not always necessary to perform on-line in the vehicle but could be done off-line, for example, when visiting the workshop, or remotely if the vehicle is connected and streaming data to a server." Since a car is not constantly in a workshop and, as previously described, network coverage can be spotty, switching to a microcontroller for fault detection would be an improvement over the original idea of this paper[Jun19].
The data used in the paper by Daniel Jung[Jun19] follows the same approach as the previous work "Residual Selection for Consistency Based Diagnosis Using Machine Learning Models"[FK18] in which a link to their data can also be found. The data provided in the email response and used in Daniel Jung's paper are 20276 sets of 42 residuals and the corresponding labels which are seven fault types and one normal state. The residuals are the deviations of the observed output of the Volvo engine from a model of the engine and were generated by a toolbox described in previous work from Frisk et al.[FKJ17] and which is available on fault-diagnosistoolbox.github.io. Daniel Jung mentions in his email that "The residuals themselves have been generated in C++ and compiled and could run on a standard laptop approximately 500x faster than real time. However, from a computational aspect it is maybe not necessary to evaluate every sample but perhaps every 10th sample". Therefore, this residual generation could also be done on a microcontroller. This dataset proved to be a good use case for microcontrollers and the implementation is discussed in a later chapter which showcases how a NN on a microcontroller could be used for real-time fault diagnosis.

4.5 AI for mobility services

The automotive industry not only sells cars but nowadays also offers other mobility services. These mobility services can also benefit from the usage of AI. Some applications are connected to the previously discussed topics of predictive maintenance and ADAS but this field also has new optimization problems e.g. regarding the positioning of vehicles to increase availability, profit, and customer satisfaction.

As described, predictive maintenance can increase reliability and prevent more expensive damage. This can of course also be applied to fleets for mobility services. One such example is the work of Michelin who lease tires to truck fleets and offer a monitoring program for performance analysis and predictive maintenance[Mic].
Cars offered as a mobility service also benefit from ADAS as was discussed above for normal cars. Since many such systems use vision data, Keyu Lu et al.[Lu+18] developed a new object detection system for Car Sharing Services which uses camera images as input data. Their system incorporates Haar filters into a CNN meaning that some weights are not determined through training but rather constrained to the factors of a Haar filter "which has a strong representation in feature extraction"[Lu+18]. They found that such a system can use less memory compared to one without such a filter. However, even their smallest system with an error rate of 9,5% in classification tasks requires more than 900 KB of memory, significantly more than is available on the microcontroller for this thesis. This shows once more, that difficult classification tasks based on complex image data are not the best match for this hardware.

Mobility services are often some form of either ride sharing or car sharing. Ride sharing means using a vehicle together with another person that wants to travel in a similar direction, in other words carpooling with usually previously unknown people. Examples for this service are UberPool[Ube] or BlaBlaCar[Bla]. Car sharing means using a car that you do not own for a short trip up to an extended amount of time. This car can also be used by different people in the times that you do not use it yourself. In both cases, there are optimization problems that can be solved with AI techniques.
During ride sharing you have to sit in a car with another often unknown passenger. To increase customer satisfaction it might be beneficial to not match passengers solely on their start and destination points but rather to also include other factors like shared interests. Lasmar Junior et al.[JRR18] questioned 296 volunteers about preferences for ride sharing and factors that make for a good ride share experience. They found that users younger than 40 years mostly preferred to share a ride with people of similar hobbies. Based on the survey results they developed a machine learning model that recommends ride share passengers based on data extracted from their Facebook profiles including age, gender and hobbies. The correlation of this system was higher than for a system that does not take hobbies into account showing that the additional data combined with the AI algorithms results in better ride share passenger matchings[JRR18].

For car sharing the system does not need to match passengers but there are multiple other optimization problems to solve. According to Brendel et al.[BRK17], there are three groups of car sharing services that can be differentiated. For two-way car sharing, the driver needs to drive both ways of his trip with the same car and bring it back to the original station. One-way car sharing gives you the option to leave your car at a different station than the one you picked your car up at. No stations exist for free-floating car sharing where users can pick up and leave vehicles anywhere within the operation area of the car sharing provider[BRK17].
To solve optimization problems and to use AI techniques, data for car sharing usage is very beneficial. If not enough real data is available, Brendel et al.[BRK17] show how additional rental data could be generated.
For one-way and free-floating car sharing there can be a relocation problem since cars might be parked in different spots than where they are needed next. The car sharing provider might need to manually relocate cars again so users have a vehicle ready nearby when needed which can result in more paid trips and consequently more profit[Alf+17]. Since manual relocations require work and therefore cause costs, it makes sense to only perform them when necessary based on accurate usage predictions. Alfian et al.[Alf+17] compare different machine learning techniques for forecasting based on real-world one-way car sharing data from HanCar in Seoul, Korea. They found that a dense NN captures "the daily number of reservation patterns for 28 days [...] quite well"[Alf+17]. Based on the NN predictions and the suggested relocations, they created a car sharing simulator and compared it to a system with no relocations. The NN system outperformed the alternatives "in terms of the utilization ratio and acceptance ratio"[Alf+17] showcasing the benefits of ML techniques.
Daraio et al.[Dar+20] also looked at ways to predict car availability, in their case for free floating car sharing in Berlin. Their system not only looks at the time and spatial position data, but also takes meteorological data into account because the weather could affect the usage of car sharing since it is an alternative to e.g. cycling or public transportation. The city map was divided into cells for which contextual data was added to describe them further e.g. as a city center, suburb or airport. They then not only looked at ML algorithms but also compared their performance to approaches without ML algorithms.
The ML algorithms performed better in most cases, especially during the more busy weekdays which also include high variances in the data, and the results were also checked for statis-

tical significance. "The outcomes of the statistical validation confirm that the performance of ML-based approaches are superior to those of baseline methods in the most challenging functional and temporal contexts (i.e., when the variance of the occupancy levels is fairly high)"[Dar+20].

While relocating can improve the satisfaction of current users and the availability at current stations, some researchers are also looking at ways to use ML for best expanding car sharing networks and user groups.
Luo et al.[Luo+19] try to predict the demand for cars in a car sharing system not only for existing stations but also during expansion for new stations. Their data is from a car sharing company in Shanghai in the year 2017 during which it expanded its network from 1705 to 3127 stations. Their system using a NN "can effectively model the complex temporal and spatial dependencies within the evolving station network"[Luo+19] and can therefore help during expansion phases as well.
When expanding a network, a car sharing provider can offer services to new people that are interested in car sharing. However, it can also be beneficial to look at reasons why some people are not using car sharing and are not interested in the current offerings. Chicco et al.[CPC20] analyze the perception of car sharing based on data from a survey performed in 2018 and focus on the 2394 Italian respondents. They found that 29% of men were users of car sharing but only 21% of women showing that there is a gender gap in the usage of car sharing. They then use AI clustering algorithms to find different groups of users with shared characteristics. Groups with high usage of car sharing tended to be younger, have a high level of education, and often live in individual single-person households in cities with more than 500k inhabitants[CPC20].

Chapter 5

Workflow

This chapter will look at the workflow of this thesis. First, the choice of the TensorFlow Lite for Microcontrollers library will be explained and its usage will be highlighted. Afterwards, the hardware running this library will be discussed, the STM32F746 microcontroller.

5.1 TensorFlow Lite for Microcontrollers

There are many ways to get machine learning running on a microcontroller. In general, the process can be split up into two parts. First, the training data is used to generate a model. This step is called learning and can require significant computing power, especially for large datasets and large neural networks. Secondly, the model is used to get insights about new data which is called inference. Since the computing power of a microcontroller is very limited, common solutions perform the learning part on a more powerful PC and only the inference on the microcontroller device.
When using machine learning without a microcontroller, two frameworks are very common. According to a small survey with 3000 developers involved in machine learning or data science about 38% of them use TensorFlow and 11% use PyTorch[Dev19]. Since the learning part of the process is usually still performed on a PC when developing for a microcontroller, it would be great if existing code could be reused.

The developers of TensorFlow are trying to achieve this using their relatively new project TensorFlow Lite for Microcontrollers which will also be the framework used for this bachelor thesis. It was chosen over the X-Cube AI package from STM because of its support for microcontrollers from multiple manufacturers instead of only one.
This chapter will describe the general idea of it and the workflow of its usage for this thesis.
Pete Warden presented TensorFlow Lite for Microcontrollers at the TensorFlow Developer Summit in March 2019[War19]. The main selling point is the project being part of TensorFlow Lite which means that " it uses the same APIs, file formats, and conversion tools, so it is well integrated into the whole TensorFlow ecosystem, making it easier to use"[War19]. The project is still very new and TensorFlow's own website calls it an "experimental port"[Tenj].
TensorFlow Lite for Microcontrollers is written in C++ 11 and requires a microcontroller with a 32-bit platform. It does not require any C or C++ standard libraries, dynamic memory allocation or support for a file system by the microcontroller. At the time of writing, the library has a size of about 16KB in RAM[Tenj]. The developers of the library have tested examples for Arduino, ARM Cortex-M Series based microcontrollers and ESP32 boards and provide these programs as well.

Deploying a machine learning model on a microcontroller using TensorFlow Lite for Mi-

crocontrollers is a multistep process.
First, a suitable model needs to be developed and trained. This step does not differ vastly from a normal TensorFlow project but the restrictions of a microcontroller have to be kept in mind. This step will be performed in Python for this paper as it is the most popular choice for developing a TensorFlow model and the default language for TensorFlow's own tutorials and install pages. Depending on the familiarity with TensorFlow, this step might not differ much from current machine learning workflows. As usual, a model to solve the problem needs to be decided on. With TensorFlow Lite for Microcontrollers, there are currently some limitations for the supported operations. At the time of writing, the supported operations can be found in the *all_ops_resolver.cc* file. They include support for *FullyConnected* neural networks, different activation functions like *Softmax* or *Swish*, padding and pooling, convolutional layers for CNN with *Conv2D* or *DepthwiseConv2D*, and more [Tena].
Furthermore, the size of the model is limited, since it needs to run on a microcontroller later on and fit into its memory. It therefore makes sense to try out some examples with smaller models to get an understanding of the number of layers and nodes that can work on the microcontroller being used. Training the model with the training data and evaluation of the model's performance work as usual with TensorFlow.

After a suitable model is found, it needs to be converted for later use with the microcontroller. First, it is converted into FlatBuffer as it would normally be used with TensorFlow Lite and it is saved to a file. This is done with a Python code similar to this:

```
# Convert the model to the TensorFlow Lite format
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
# Save the model to disk
open("converted_model.tflite", "wb").write(tflite_model)
```

In this step, TensorFlow can also perform a quantization. By default, the weights of the NN are stored as 32 bit floating point values. To reduce model site, a different data type with fewer bits can be used. The weights can either be changes to 16 bit floats or to 8 bit integers[Tenh]. If all weights are quantized to use integers, the model size, hence, can be reduced by up to a factor of four. Depending on the hardware choice, integer arithmetic can be faster than float arithmetic which means quantization can also result in a speed improvement. Since TensorFlow Lite for Microcontrollers does not support all TensorFlow operations and layers, the success of a quantized model is not guaranteed and at the time of writing there are open issues for quantization related problems on the TensorFlow Lite for Microcontrollers github[Tenb] [Tene] [Tenc].
The converted model needs to be included in the microcontroller code later on. The suggested way to do this is "to include it as a C array"[Tenk] which can then be compiled as part of the program deployed on the microcontroller. One way to do so, also recommended by the creators of the library, is to use the Linux *xxd* command. The command creates a hexdump from the *.tflite* file that was previously exported. By using the *-i* option the output is written in C as a complete static array which is then saved to a file that will be used by the microcontroller[Wei]. The full command would look similar to this:

```
$ xxd -i converted_model.tflite > model_data.cc
```

The last step is to use the TensorFlow Lite for microcontrollers C++ library together with the converted model to perform inference on the device. The input to the model needs to have the same format and if used for the training data, also the same data preprocessing needs to be applied.

The inference on the microcontroller usually follows certain steps which are described in

the example programs of the library[Tenl]:

1. Include the required library headers meaning an interpreter which loads and runs the models, a resolver that provides the operations needed by the interpreter, the schema for the FlatBuffer file format used for the saved model, the version information and a way for TensorFlow to output errors and debug information. Also, include the previously generated model. The code could look like this:

```
// The resolver
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
// The error reporter used for error and debug messages
#include "tensorflow/lite/micro/micro_error_reporter.h"
// The interpreter
#include "tensorflow/lite/micro/micro_interpreter.h"
// The schema for FlatBuffer
#include "tensorflow/lite/schema/schema_generated.h"
// The version information for the library
#include "tensorflow/lite/version.h"
// The header file for the converted model
#include "model_data.h"
```

2. Set up the logging for error and debug messages by creating a *tflite::ErrorReporter*. The code could look like this:

```
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

3. Load the model and instantiate the resolver and interpreter so the model can be used for inference. The code could look like this:

```
tflite::Model *model = tflite::GetModel(model_tflite);
if (model->version() != TFLITE_SCHEMA_VERSION) {
    TF_LITE_REPORT_ERROR(error_reporter,
        "Model␣provided␣is␣schema␣version␣%d␣not␣equal␣"
            "to␣supported␣version␣%d.",
            model->version(),
            TFLITE_SCHEMA_VERSION);
        return;
}
static tflite::ops::micro::AllOpsResolver resolver;
static tflite::MicroInterpreter static_interpreter(model,
    resolver, tensor_arena, kTensorArenaSize, error_reporter);
tflite::MicroInterpreter *interpreter = &static_interpreter;
```

4. Set up the tensors for the input and output before providing input values based on new data. Depending on the data type, the code could look like this if e.g. floats are used:

```
interpreter.AllocateTensors();
TfLiteTensor *input = interpreter->input_tensor(0);
TfLiteTensor *output = interpreter->output_tensor(0);
float *input_floats = input->data.f;
// Load the data into the input tensor in some way
// The source of the data will vary depending on the problem
```

```
for (int i = 0; i < input_size; i++) {
    input_floats[i] = data[i];
}
```

5. Run the model by using *Invoke()* and obtain the output. Depending on the meaning of the output, the code can look a bit different. If e.g. a classification problem needs to be solved, the argmax of the output values might be used as the prediction. In this case, the code could look similar to this, provided the C++ standard library is available:

```
TfLiteStatus invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(error_reporter, "Invoke␣failed\n");
}
int prediction = std::distance(output->data.f,
    std::max_element(output->data.f,
    output->data.f + output_size));
```

5.2 The Used Microcontroller: STM32F746

To perform machine learning on a microcontroller, a device that can do so is needed. It would make sense to choose a device that has support for existing machine learning libraries so the whole program does not need to be written from scratch. In this case, support for TensorFlow Lite for Microcontrollers would be beneficial as the motivation to use this library was just highlighted. Prof. Knoll generously provided an STM32F746 Discovery kit for usage in this thesis which can be seen in figure 5.1. The board is one of the supported platforms for the TensorFlow Lite for Microcontrollers library and examples for this specific board are also available which makes it a great fit [Tenj].
The board features a 32-bit ARM Cortex M7 core with a maximum frequency of 216 MHz and an included Floating-point unit (FPU). The ARM Cortex-M Series are ARM's processors for microcontrollers out of which the ARM Cortex M7 is "the highest performance member"[Arm]. The ARM core has a maximum current consumption without peripherals of 141 mA which results in a power consumption of up to 508 mW[STMc]. The board has 320 KB of SRAM which is enough to run many small machine learning models since the TensorFlow Lite for Microcontrollers library only needs about 16 KB itself at the time of writing[Tenj]. The integrated ST-LINK/V2-1 debugger will be used for debugging and error messages in the machine learning examples of this thesis. The board also contains other components like a touchscreen, USB ports, an Ethernet port, and microphones[STMa]. The screen is used for one of the examples to display input images and the inference output.

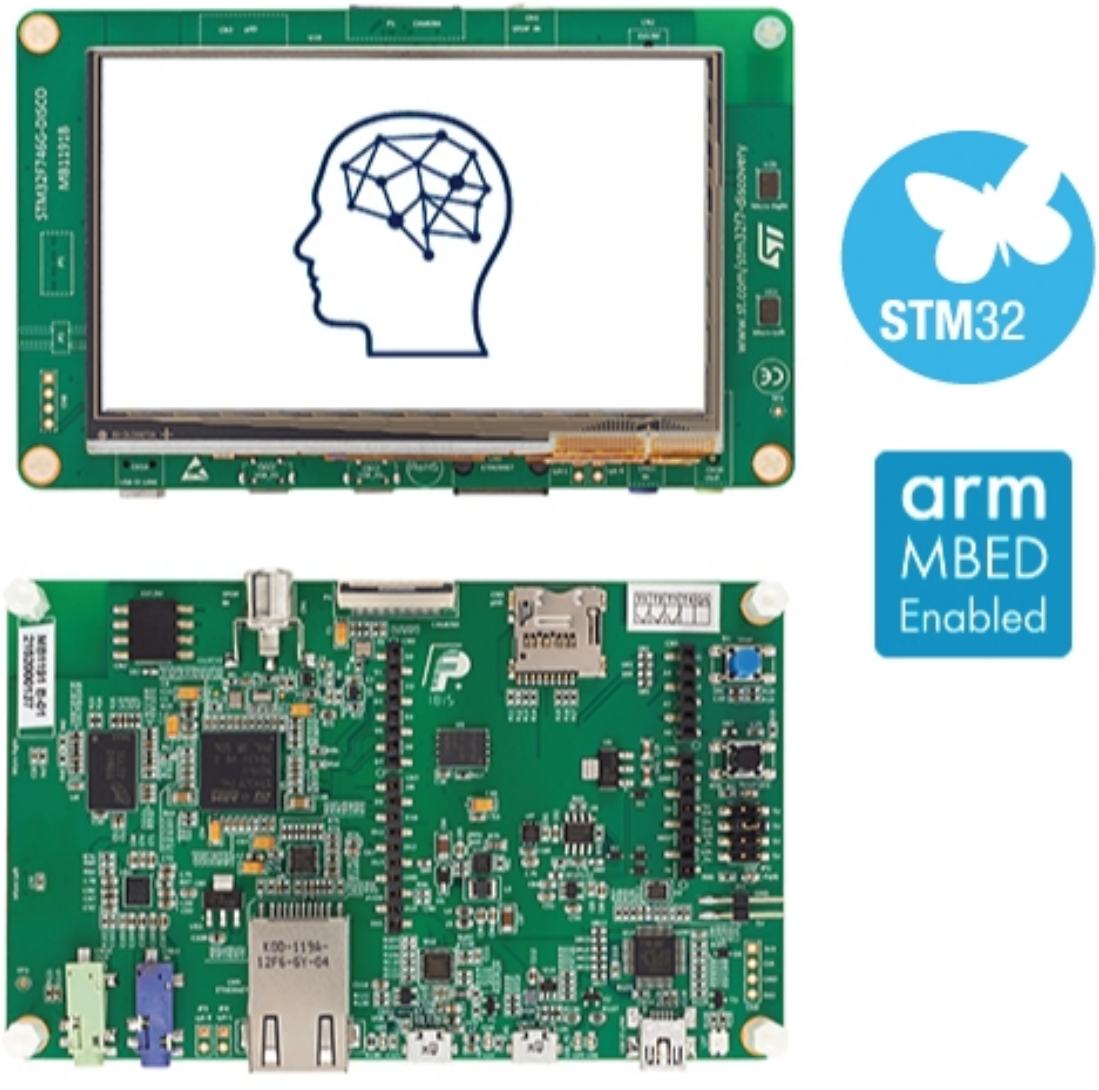

Figure 5.1: The STM32F746 Discovery kit (source: [STMa])

Chapter 6

Example Programs for Using a Microcontroller in the Automotive Industry

To demonstrate how machine learning can be deployed on microcontrollers for tasks in the automotive industry, this thesis evaluates the performance for three example datasets. First, an example from the TensorFlow Lite for Microcontrollers library is used to get a better understanding of the workflow and the performance of the STM32F746. The dataset uses 96*96 pixel images and a MobileNet for classification. The runtime for the inference is shown to be relatively high which shows the limited computational power of the microcontroller.
Next, a dataset with smaller images is used which contains images with one car or images without a car. This binary classification is attempted using dense NNs and also CNNs with the goal of finding a NN structure that achieves a satisfying accuracy while keeping a short inference time that is suitable for real-time tasks.
Last, the topic of fault diagnosis and predictive maintenance described in the use case analysis is evaluated using the dataset shared by Daniel Jung. Several dense NNs are evaluated and showcase how similar tasks could be solved using ML on microcontrollers.
A detailed user documentation for running the three examples can be found in the appendix.

6.1 Object Recognition in Images with MobileNet: Person Detection Example

Trying out an "experimental port"[Tenj] of a library can be challenging and therefore the first example will be based on one of the demo programs of the TensorFlow developers themselves. The example is called person detection and is part of the four examples provided by TensorFlow to showcase their experimental port for microcontrollers at the time of writing[Pet19]. As described in the use case analysis, object recognition with vision data is used widely in the automotive industry. This program will showcase that object recognition can be implemented on a microcontroller but also the restrictions of the limited hardware. The program in itself is not intended for use in the automotive industry but the results about the classification of objects in image data are applicable to other tasks there as well.

The dataset and model The example trains a neural network to classify images by recognizing whether a person is present in them or not. It uses a subset of the COCO dataset which normally contains more than 330k images with bounding boxes for 91 object categories e.g. people, bicycles, different animals, furniture items and more[Lin+14]. The target label is not a bounding box but just the binary classification regarding the presence of a person which

is one of the 91 object categories[Pet19]. This new image collection is called Visual Wake Word Dataset and was developed by researchers at Google in 2019. It only contains people that take up more than 0.5% of the image area and contains 115k images with 47% of them labeled as "person", the dataset hence is nearly balanced[Cho+19].
A typical approach for object recognition in images using neural networks on limited resources is the usage of a MobileNet which is a class of efficient CNN models that were developed by Howard et al. in 2017 [How+17]. According to the original paper[How+17], MobileNets of version 1 have two parameters used to decide on the trade-off between low latency and high accuracy. The width multiplier α is used on the layers to reduce the number of inputs and outputs by multiplying them with this factor $0 \leq \alpha \leq 1$. The second modifier is the Resolution multiplier which is applied to the input image and the internal representations in the layers compared to the default input resolution of 224*224[How+17].
This example will also use a version 1 MobileNet with a width multiplier of just 0.25 and an image input size of 96*96 pixels[Pet19]. The model will fit in 250 KB of flash memory[Pet19] which together with the TensorFlow library and the rest of the program will come relatively close to the 320 KB of SRAM of the used microcontroller for this thesis, even though the width multiplier of 0.25 is as low as the smallest example in the original MobileNet paper and the resolution of 96*96 is smaller than the originally tested smallest resolution of 128*128[How+17].
This highlights that models that are considered "efficient models [...] for mobile and embedded vision applications"[How+17] are already pushing microcontrollers to their limits and the choice of the model really needs to factor in the available memory. The model achieves an accuracy of about 84% on the described Visual Wake Word Dataset in the described binary classification[Pet19].

Training in Python The code for the example can be found on the official TensorFlow github[Tenf] and is not original work of this thesis. As usual with TensorFlow Lite for Microcontrollers, a model is created in Python. The visual wake word dataset is part of slim which itself is part of the TensorFlow models repository[Tend]. The corresponding scripts of slim can be run to download the Visual Wake words dataset and use the right labels[Teng]. Afterwards, use the slim script "train_image_classifier.py" as described in the person detection example to train the MobileNet. The person detection example recommends settings that require significant computing power. It "will take a couple of days on a single-GPU v100 instance to complete"[Teng] and it will reach an accuracy of about 84% and the finished model then can be exported to TensorFlow Lite. For this example, weights are also converted from floats to eight bit values using *TFLiteConverter*. Finally, the model is converted to a C source file by using *xxd* as was described in previous chapters.

The C++ code At this point, the Python side of the project is done and the work continues in C++ to get the inference running on the microcontroller. Embedded machine learning applications often have 5 common steps according to Warden[Pet19]:

1. Get input values
2. Preprocess the input to obtain features for the machine learning model
3. Run inference with these features and receive an output from the model
4. Postprocess the output, e.g. by finding the argmax
5. Act depending on this information

In this case the person detection example works as follows[Tenf]:

1. Take a picture using a camera of a test image stored as an array in the program

2. Crop the image to 96*96 pixels to fit the input size of the neural network
3. Run inference with the new image using the MobileNet
4. Decide if the score for the presence of a person or for the absence of a person is higher
5. Output the result using the serial debug connection

For compatibility reasons with the Arduino programming language, the program has two central functions in *main_functions.cc* called *setup* and *loop*. On Arduino, *setup* is called when the program starts and is used to initialize the program. Afterwards, the *loop* function is executed consecutively[Ard]. If the program is not deployed on Arduino, as in this case, this behavior is achieved using the simple main function in *main.cc*:

```
int main(int argc, char* argv[]) {
  setup();
  while (true) {
    loop();
  }
}
```

The *setup* function initializes the *tflite::MicroErrorReporter* for debugging, loads the model, creates the resolver, and the interpreter. It also allocates the required memory and sets the input variable to the right location of the model. This is the normal approach for TensorFlow Lite for Microcontrollers as was described in a previous chapter. Afterwards, the *loop* function runs in an infinite loop. First, it calls *GetImage* defined in *image_provider.cc* to load data into the input of the model. Then, it invokes the inference and reads the output of the TensorFlow model. Finally, this data is used to call *RespondToDetection* defined in *detection_responder.cc*. The implementation of the Image Provider and Detection Responder is dependent on the used microcontroller and available hardware. For Arduino devices, there is an implementation using the Arducam which takes real images which are then cropped to 96*96 pixels. Similar implementations also exist for the SparkFun Apollo 3 board and ESP32 microcontrollers with compatible cameras. If no camera is present, the default implementation fills the input with 0 values, but the example also includes sample image data that could be used instead. In this thesis, the code is slightly altered, so the sample images stored in *person_image_data* and *no_person_image_data* as arrays are used in an alternating fashion.

Results The example was deployed on the STM32F746 microcontroller using Mbed as this is the described way to do it in the TensorFlow examples and Mbed is compatible with boards from multiple vendors. To make use of the included display of the STM32F746 board, a function to display the image and the predictions on the screen was added. To evaluate the performance of this example, the time for each inference was measured. The time does not include loading the data into the model, just the inference alone happening when *Invoke()* is called. The time was measured using the Mbed *Timer* class which measures the elapsed time in seconds. The time was consistent for the first two decimals and 4.76 seconds for each inference. This long runtime rules out many possible applications for a NN of this size. Recognizing objects outside of the car for driver assistance needs to be significantly faster, e.g. to recognize a car in front of the vehicle. It could, however, be used to detect drowsiness, recognize a specific passenger, or for number plate recognition of parked cars in a parking garage similar to the work of Fakhar et al.[Fak+19].
An approach of just taking common NN architectures like the MobileNet and immediately deploying them on the microcontroller expecting quick performance for any task is not possible with such restricted hardware. For many real-time tasks, the inference just takes too long. The task needs to be evaluated for its complexity and not all challenging ML problems are solvable with such hardware. Furthermore, the NN also needs to be scaled down appropri-

ately. The next two examples will look at such an approach and problems that can be solved more easily with a microcontroller.

6.2 Object Recognition in Smaller Images with Dense NN and CNN: UPM Vehicle Image Database

The first example showcased that CNNs can be deployed on the STM32F746 for image classification but the inference time of multiple seconds is too slow for many applications in the automotive industry. One example is the identification of cars driving in front of the vehicle for advanced driver assistance systems where quick inference is essential. As was shown in the related work section, autonomous driving usually requires significant computing power, Nvidia GPUs are a common hardware choice. The microcontroller just can not compete with such devices, the task would need to be simplified in some way. One option is to only detect vehicles driving directly in front of the vehicle and which are relatively close meaning that they would take up nearly the whole image of a forward-pointed camera. Such a system might not be able to provide a full self-driving experience for a self-driving car but could be used for adaptive cruise control or forward collision warning. The images also need to be relatively low resolution since 96*96 pixels in the previous example combined with the MobileNet already amounted to significantly too long inference times.

The dataset Datasets containing images with these characteristics exist and to demonstrate a proof of concept, the UPM Vehicle Image Database from the Universidad Politécnica de Madrid is used[Gru]. It was selected since it contains images with a low resolution of just 64*64 pixels and the images either contain no vehicle or one vehicle that is relatively centered and fills out nearly the whole frame. Instead of more complex bounding boxes being calculated, this dataset is used for binary classification. Compared to more complex options, this dataset makes the classification easier and a successful approach using the STM32F746 more likely. The dataset was created by Jon Arróspide et al.[ASN12] whose system has a two step process. First, a likelihood model is used to find areas of the image that are likely to contain a vehicle. In a second step, a SVM is used only on these cropped regions with the potential vehicle candidates for the final result. The dataset with the 64*64 pixel images is used as an input for the SVM of the second step. It contains 4000 images with vehicles and 4000 images from the vehicle-mounted forward-looking camera without vehicles. The dataset is "open for use" and can be downloaded from the institute's website[Gru]. The images contain images "at least 50% of the vehicle rear" and "different viewpoints of the vehicle rear"[ASN12]. The "vehicle appearance, pose, and acquisition conditions (e.g., weather conditions, lighting)"[ASN12] vary to make the dataset more applicable to different scenarios.
The SVM used by Arróspide et al. was evaluated with different parameters and a trade-off between high accuracy and low complexity/low inference time was made, the final system has an "average correct detection rate of 90.51%"[ASN12]. The system with the two step process was "able to operate near real-time at 10 fps on average over an Intel(R) Core(TM) i5 processor running at 2.67 GHz"[ASN12] meaning each iteration took about 100 ms. The goal for this example will be to deploy a NN on the microcontroller to classify the 64*64 pixel images of the second step. It should achieve a similar accuracy while keeping the inference time low.
As is needed with the TensorFlow Lite for Microcontrollers workflow, the deployment of the NN consists of two main parts, the Python program creating and training the NN and the

C++ code deployed on the STM32F746 using the created model for inference. The structure of the NN needed to be relatively small for the inference time to be significantly lower than in the first example. One approach taken in related work is the usage of small dense NNs as was the approach taken by Farooq et al.[Far+10] or Kumar et al.[KK12]. Another option is the usage of a CNN since they are known to perform relatively well for image classification and convolutional layers were e.g. deployed in the related work of Borghi et al.[Bor+17].

Training in Python For both options, dense NN and CNN, multiple variants with a varying number of neurons, layers and different parameters were trained in Python. First, the *extractImages.py* script is used to split up the 8000 images into a training, validate, and test set with an 80%, 10%, and 10% split while keeping the balance of the two classes of images. Afterwards, *CreateModel.py* is used to create, train, validate, and export the models. Since the dataset contains relatively little images for an image dataset, overfitting can be a big problem. To combat this, augmentation was used. The *ImageDataGenerator* from the TensorFlow Keras library was deployed as it provides different forms of augmentation that can be turned on and off easily. To reduce the complexity of the classification, a grey-scale version of the 64*64 images is used. The augmentation is only performed for the training data, not the validation and testing one. Since cars can be left and right of the vehicle, *horizontal_flip* augmentation makes sense, but *vertical_flip* does not since cars usually do not occur up-side-down on the road. A zoom of up to 20% and a brightness reduction of up to 20% help with making the images more varying. These augmentations reduced the overfitting which was relatively large without them.

After the augmentation, the images are further reduced in size to 32*32 pixels using averaging to reduce the inference times and the number of weights. The NN, hence, has 1024 inputs. The usage of a convolution with a set kernel for edge detection was found to be slow and the approach of integrating the convolution into the NN structure, as is the case for a CNN, was chosen instead. This also allows the weights of the convolution matrix to be chosen during training instead of using fixed values that might not be optimal.
The model is then trained using the Adam optimizer with binary cross-entropy as loss and the accuracy is also tracked for the training and validation sets for each epoch. The accuracy was found to not improve anymore for many iterations before 600 epochs so this value was chosen as the number of training epochs. After the weights are determined, the models are converted to TensorFlow Lite, once directly and once with quantization by using *tf.lite.Optimize.OPTIMIZE_FOR_SIZE*. Unfortunately, for this example, quantization led to hybrid models which are not supported by TensorFlow Lite for Microcontrollers yet. Hence, this example will only include results for the normal tflite models. Afterwards, xxd is run to convert the models to C files for later use with the microcontroller. Finally, the test data is used to evaluate the two models through the TensorFlow inference API. Each evaluation includes the confusion matrix, f1 score, precision, recall, and accuracy. All calculations are performed using "binary" for averaging since this is a binary classification problem. Since the models and their weights are not changed afterwards, this evaluation has the same inference results, no matter if it is run using the TensorFlow inference API in the Python Script on a PC or TensorFlow Lite for Microcontrollers on the STM32F746 board. The evaluation on the microcontroller will be performed to analyze the inference times, the accuracy already having been determined on the PC. All results can also be found on the USB stick that is part of this thesis.

The C++ code Once the work of the Python script is done, the model can be used in C++ using the TensorFlow Lite for Microcontrollers Library. The mbed approach taken in the previous example from TensorFlow was not ideal, since it does not include a debugger that works with the STM23F746 microcontroller. Therefore, this example will use a different IDE, STM32Cube, that provides this and more additional features. Since this example was not based on previous C++ code, an empty STM32Project was first created using the STM32CubeIDE. Next, the TensorFlow Lite for Microcontrollers library was included. To see the output of this library, the DebugLog method needs to work but the default one only does when using mbed. It was altered to use *ITM_SendChar* to send each char instead. Afterwards, the main program was written which can be found in *main_functions.cc*.
The structure of the program follows the typical TensorFlow Lite for Microcontrollers structure that was previously described. The setup part did not need to be changed significantly from the TensorFlow Lite for Microcontroller examples. The loop part uses multiple example inputs for inference which are loaded in alternating order by copying the values into the float input tensor of the NN. The inference result is decided by checking if the output is closer to 0 or to 1, the two labels. The result can be printed out on the debug output if needed and in this way also compared to the real label of the test data. The runtime was measured using *HAL_GetTick()* which returns the number of SysTick elapsed which is equal to the number of milliseconds elapsed. The inference is run 100 times with changing inputs to increase the accuracy of the time measurement. The debug output was turned off during measurement to not affect the time measurement. Afterwards, the time is printed in ms and divided by 100 for the results in table 6.1. To change the used model, the model array from the corresponding cc file is copied into *model.cc* and the name of the array is changed to *model_tflite* as this is the reference name used in *main_functions.cc*.

Results Now that both programs are finished, different NN structures can be easily tested by defining them in Python, running the script to train the NN and evaluate its accuracy, and copying the NN's C++ array into *model.cc* to test the runtime on the STM32F746. The training with an image dataset containing only 8000 pictures showcased problems regarding overfitting without the augmentation with training accuracies close or even equal to 100% and a clear gap to the validation and training accuracy. The added augmentation through zoom, brightness changes and horizontal flipping helped with the overfitting. Unfortunately, the validation accuracy is still somewhat unstable and fluctuates around the training accuracy. The loss also has the same problem. To deploy a similar system in the real world, using a larger dataset is advisable. Since the validation accuracy fluctuates, the measured test accuracy is likely to be different depending on when exactly the training is stopped and should be seen as including some deviation. The accuracy was rounded to three digits.

For the models, there are multiple parameters that can be changed. For dense NNs, the last layer needs to have one neuron since the output does not use one-hot-encoding. The previous layers can have a changing amount of neurons and a changing amount of layers. Three dense NN architectures were tested, a smaller 1024-8-4-1 structure with an average accuracy of 93.5%, a larger 1024-16-16-16-1 structure with an accuracy of 97.0%, and a large 1024-32-32-16-4-1 structure with an accuracy of 96.7%, all performing better than the SVM used by Arróspide et al.[ASN12]. The execution times on the NN were also quick with 4.25 ms, 8.60 ms, and 17.07 ms. This shows that dense NNs are one possible solution that can be deployed on microcontrollers for real-time image classification. Especially the dense NN with the 1024-16-16-16-1 structure is shown to perform accurately and with a low inference time that is applicable to real-time tasks. Even larger dense NNs were not tested, since they

would not fit into the RAM of the STM32F746, the largest dense NN leads to a program that requires 92.04% of the available RAM according to the STM32Cube IDE. Larger dense NNs are also not likely to perform better since overfitting might become a bigger problem and the largest dense NN also does not perform better than the second largest one. The structure of the best performing dense NN is also depicted in figure 6.1

Since CNNs are known to perform even better than dense NNs in image classification tasks, the addition of convolution and max-pooling layers was also tested.
The first CNN has the same structure as the first dense NN and adds one convolutional layer with two convolution filters of size 3*3 and stride one at the front.
The second CNN showcases the focus on convolutional operations by using two convolutional layers with four filters each and two max-pooling operations of size two in between. These layers are followed by two dense layers.
The fourth CNN tests the results of shrinking this structure by reducing the number of convolutional filters to two, increasing the max-pooling size to four and reducing the dense layers to only include four and one neurons.
The convolutions of the CNNs were found to drastically increase the inference times on the microcontroller. While the small dense NN with a 1024-8-4-1 structure results in an acceptable execution time of 4.25 ms on the microcontroller, the addition of the convolutions in the first CNN increased the runtime to 81.02 ms on the STM32F746. However, adding a convolution layer also increased the accuracy from 93.5% to 96.6% showcasing the trade-off between high accuracy and quick execution time. Depending on the use case, this added accuracy might outweigh the added time penalty and 81.02 ms could still be sufficient in some scenarios, especially since the original system used with this dataset had a cycle time of 100 ms, albeit including the first phase with larger images.
The larger second CNN architectures performed even better and achieved an accuracy of 96.9% accuracy on the test set but with a very long execution time of 264.16 ms on the STM32F476. The other two options (CNN3 and CNN4) were found to also have too high runtimes on the microcontroller of 168.56 ms and 78.47 ms while achieving accuracies of 97.3% and 94.1%.

While the accuracies of the first four CNNs were sufficient and sometimes even outperformed the results of the dense NNs and especially the SVM in the paper by Arróspide et al.[ASN12], the runtime of the CNNs needed to be reduced. To do so, four more CNN structures were tested.
Since the dense NNs have a low runtime even with many layers, a slightly bigger dense structure with sixteen, sixteen and one neurons was coupled with a single convolution filter at the front for CNN5. While the runtime was lower than the previous CNNs, 45.25 ms was still not satisfactory.
Next, the work of the dense layers was reduced by adding max-pooling of size four in between and reducing the dense structure to only eight and one neurons. This structure of the sixth CNN only helped with the runtime slightly reducing it to 40.65 ms but dramatically reduced accuracy to only about 88.4%. CNN6 shows overfitting, which unfortunately is also the case for the last two structures, CNN7 and CNN8. The overfitting can be seen in figure 6.3.

Since one normal convolution with a filter of size 3*3 already required too much computational power, another trick was needed. The stride parameter was found to be very helpful. Instead of moving the center of the convolution by one pixel horizontally or vertically, a different value can also be used. Since the whole image should be considered for the convolution, values larger than three did not make sense since this is the size of the filter. Matching

this value also means that the convolution never overlaps which is not ideal, therefore only the stride value of two remains for testing. If the center is moved by two pixels in both directions, only about every fourth pixel is the center of the operation which would reduce the computational costs by a factor of approximately four as well. The seventh CNN uses only one filter and stride size two followed by sixteen, sixteen and one neurons. The runtime on the microcontroller for CNN7 matches the previous thoughts and is only 11.73 ms which is about one fourth of the quickest previous CNN structure. The test accuracy of 94.8% is better than the small dense NN but not quite as well as the larger dense NN. Overfitting was a slight problem for this structure and combined with the fluctuating validation accuracy this might explain why it did not perform quite as well as the best dense NN, but the CNN still achieved a great runtime that is sufficient for many real-time scenarios.
Since increasing the stride further is deemed to not be sensible, the remaining option to reduce runtime is to reduce the computational cost of the dense layers. The eight CNN structure uses max-pooling with size two after the convolution and only eight and one neurons in the dense layer. The runtime was only reduced by 0.84 ms to 10.89 ms compared to CNN7. The accuracy, however, suffered and dropped to a low 85.0%. Therefore, CNN7 is deemed to be the best structure for low runtimes while outperforming the smallest dense NNs and the original SVM in accuracy. The structure of CNN7 is also shown in figure 6.2. The results of this example show, that very lightweight convolutional layers can be successfully deployed on a single-core microcontroller for real-time image classification.

Results for the 11 different NN structures			
NN structure	Accuracy	Runtime	Size
1024-8-4-1	93.5%	4.25 ms	34388 B
1024-16-16-16-1	97.0%	8.60 ms	69668 B
1024-32-32-16-4-1	96.7%	17.07 ms	140004 B
CNN1	96.6%	81.02 ms	59764 B
CNN2	96.9%	264.16 ms	7732 B
CNN3	97.3%	168.56 ms	30752 B
CNN4	94.1%	78.47 ms	3408 B
CNN5	96.9%	45.25 ms	60744 B
CNN6	88.4%	40.65 ms	3400 B
CNN7	94.8%	11.73 ms	17544 B
CNN8	85.0%	10.89 ms	3404 B

Table 6.1: Results for different NN

Figure 6.1: The structure of the best performing dense NN

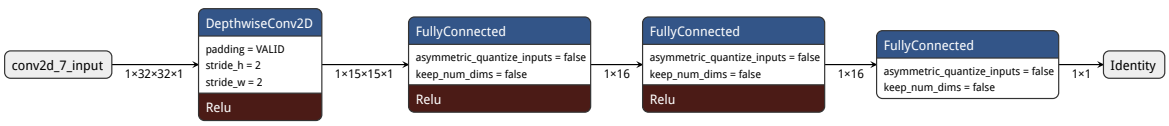

Figure 6.2: The structure of CNN7

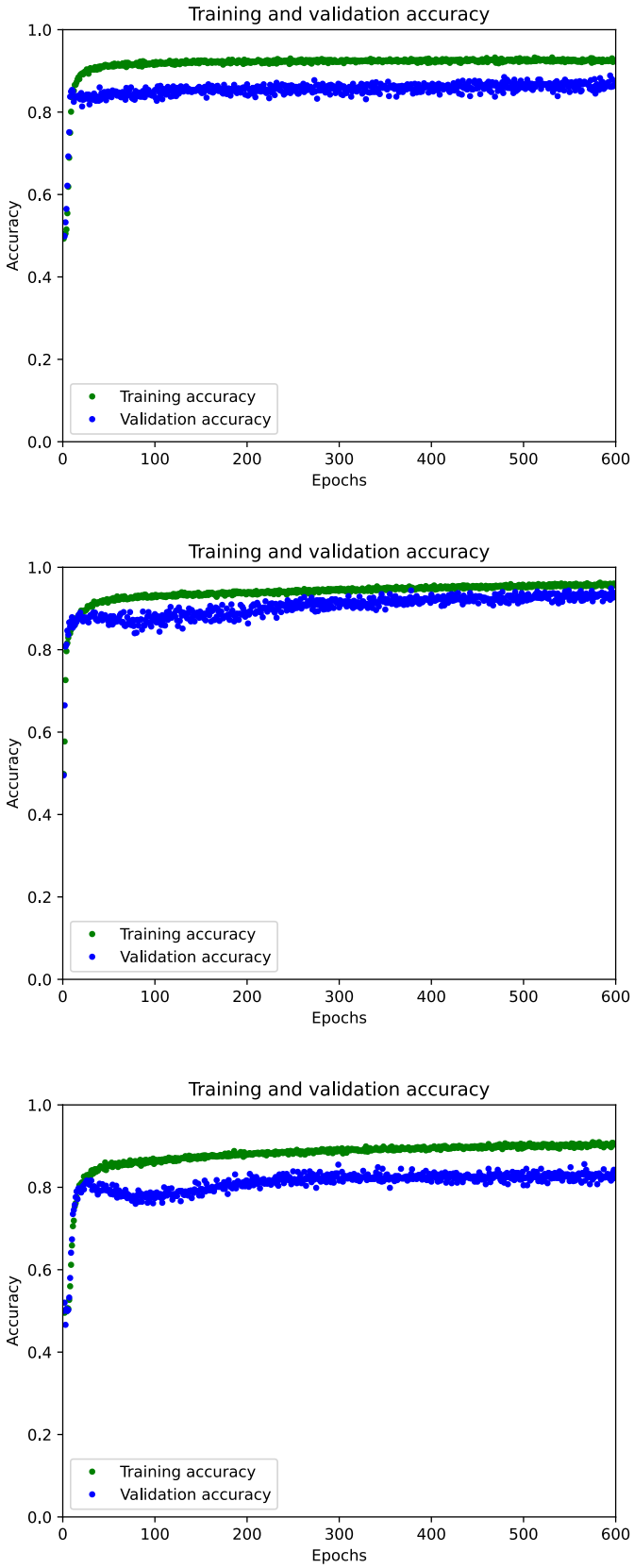

Figure 6.3: Overfitting for CNN6, CNN7, and CNN8

6.3 Fault Diagnosis and Predictive Maintenance: Engine Fault Diagnosis

The last example will showcase how machine learning on microcontrollers could be used for fault diagnosis and predictive maintenance. As described in the use case analysis, there are many applications of machine learning in the automotive industry for these goals.

The dataset Daniel Jung, the author of "Engine Fault Diagnosis Combining Model-based Residuals and Data-Driven Classifiers"[Jun19], was so kind as to share the dataset used in his paper. It will be used for inference on the STM32F746 microcontroller by deploying small fully-connected neural networks.
As described in the use case analysis, the dataset contains 20276 measurements with 42 residuals each and the corresponding labels which are seven fault types and one normal state. In the original paper, Daniel Jung performed a feature selection and only used 9 of the 42 residuals that were most relevant for the classification. The code by Daniel Jung has *sol_set = [1, 2, 21, 23, 28, 29, 31, 37, 40];* in Matlab where the index starts at one, hence all indexes have to be decremented in Python where indexes start at zero. With more computing power, all 42 values can be taken into account which will also be done in this thesis. The original paper used random forest (RF) and One-class SVM for classification and achieved an average accuracy of up to 83% with the 9 residuals and up to 86% with all 42 residuals[Jun19].
Since the example uses real data from an internal combustion engine, its results can show how machine learning on microcontrollers could be used in the automotive industry. Compared to the two previous computer vision examples, there are significantly fewer input values for the machine learning model and the difficulty of classification is also lower. This makes it a good fit for the limited resources of a microcontroller.

Training in Python The data was provided in a Matlab data file and to increase compatibility the data was converted to a csv file first. The conversion was done using a Python script with the libraries SciPy and Pandas.
Afterwards, another Python script was used to create the model, as is needed in the TensorFlow Lite for Microcontrollers workflow. First, the data is read from the csv file and depending on the test case either all 42 residuals or only the 9 most important ones are used. The 20276 measurements are then split up into a training-, validation-, and test-set with a 70%, 15% and 15% split. As was shown by Faroq et al.[Far+10] and Kumar et al.[KK12], small dense NNs can be used on single-core microcontrollers for automotive applications. This approach will also be taken for this example, a TensorFlow Keras model is created. Depending on the test case, the number of layers and nodes changes. In general, it has 9 or 42 input values, some hidden dense layers with rectified linear activation function, and a dense layer with 8 neurons for the 8 labels with a softmax activation function at the end. The model is then trained with accuracy as the training metric until the evaluation accuracy does not improve anymore after 5000 epochs. The Adam optimizer is used and a stable learning curve was achieved with a learning rate of 0.0001. The model is then converted to TensorFlow Lite, once without and once with quantization by using *tf.lite.Optimize.OPTIMIZE_FOR_SIZE*. Unfortunately, for this example, quantization sometimes led to hybrid models which are not supported by TensorFlow Lite for Microcontrollers yet. This error seems to occur every time, a big size reduction using quantization was achieved by the Python script. The error message is the same as for the previous example: *Hybrid models are not supported on TFLite Micro. Node FULLY_CONNECTED (number 0f) failed to prepare with status 1 AllocateTensors() failed*. If "quantization" worked, the size remained nearly identical, the biggest size reduction was from 5340 B to 5304 B. It's likely, that no real quantization occurred and the data type re-

mained the same. These cases where the "quantized" models were working also had nearly identical runtimes, between 0.0013 ms slower to 0.0002 ms quicker. Hence, this example will again only include results for the normal tflite models. Afterwards, xxd is run to convert the models to C files for later use with the microcontroller. The test data is then used to evaluate the two models through the TensorFlow inference API. Each evaluation includes the confusion matrix, f1 score, precision, recall, and accuracy. All calculations are performed using "micro" for averaging which calculates metrics by counting the total true positives, false negatives and false positives. Since the models and their weights are not changed afterwards, this evaluation has the same inference results, no matter if it is run using the TensorFlow inference API in the Python Script on a PC or TensorFlow Lite for Microcontrollers on the STM board. The evaluation on the microcontroller will be performed to analyze the inference times, the accuracy already having been determined on the PC. Again, all results can also be found on the USB stick that is part of this thesis.

The C++ code After the work of the Python script is done, the C++ code can be deployed on the STM32F746 board. Since this example is also not based on previous C++ code, an empty STM32Project was once again created using STM32CubeIDE and the TensorFlow Lite for Microcontrollers library was included. The output is again handled by print statements over the debug output of the STM32F746.
The structure of the program again follows the typical TensorFlow Lite for Microcontrollers structure that was previously described without significant changes to the *setup()* function. The loop part uses 2 example inputs for inference which are loaded in alternating order by copying the values into the float input tensor of the NN. The inference result is determined by taking the maximum of the 8 outputs. The result can be printed out on the debug output if needed. The runtime was again measured using *HAL_GetTick()* over 10000 inference cycles with the changing inputs to increase the accuracy of the time measurement, the debug output is turned off during the measurement. Afterwards, the time is printed in ms and divided by 10000 for the results in table 6.2. To change the used model, the model array from the corresponding cc file is copied into *model.cc* as in the previous example.

Results The testing in Python was done using 6 different fully-connected Neural Networks and each with 9 or all 42 residuals as input resulting in 12 different structures. The results can be seen in Table 6.2, the accuracy was rounded to three digits. The size of the NN was determined by measuring the size of the *.tflite* files in which the weights of the NN are stored, as is also done in TensorFlow's own examples[Teni].
In all cases, using 42 instead of only 9 input values improved the accuracy by about 4.4%-7.3%. Clear overfitting occurred for the larger NN, 9-32-32-8, 42-32-32-8, 42-16-16-16-8, 9-32-32-32-8, and 42-32-32-32-8, where the training accuracy was close or even equal to 100% but the validation accuracy did not increase as much and a significant gap between them arose. See Figure 6.4 for an example for 42-32-32-8.
The confusion matrix for the best performing NN, 42-32-32-32-8, normalized for the true labels in each row, can be seen in Figure 6.5 and its structure is visualized in figure 6.6.
The largest model was only about 17 KB in size while 320 KB of memory is available. Due to overfitting, even larger NN structures would not be beneficial as training accuracy is already at 100%. The best achieved accuracy of 96.8% is better than the 86% in the original work by Daniel Jung[Jun19] The longest runtime of up to 1.7 ms is also good and sufficient since it achieves a 588 hz diagnosing frequency while other "real-time" fault detection systems e.g. achieve only 40 hz[ŞBU16]. Deploying a NN on a microcontroller for fault detection or pre-

dictive maintenance in the automotive industry is thus shown to be possible.

Results for the 12 different NN measurements			
NN structure	Accuracy	Runtime	Size
9-8-8	84.4%	0.2243 ms	1840 B
42-8-8	91.5%	0.3294 ms	2896 B
9-8-8-8	85.9%	0.2882 ms	2516B B
42-8-8-8	93.2%	0.3922 ms	3572 B
9-16-16-8	89.2%	0.4346 ms	3892 B
42-16-16-8	94.9%	0.6317 ms	6004 B
9-32-32-8	92.2%	0.8657 ms	8184 B
42-32-32-8	96.6%	1.250 ms	12408 B
9-16-16-16-8	90.3%	0.5814 ms	5340 B
42-16-16-16-8	95.8%	0.7797 ms	7452 B
9-32-32-32-8	93.1%	1.315 ms	12764 B
42-32-32-32-8	96.8%	1.700 ms	16988 B

Table 6.2: Results for different NN

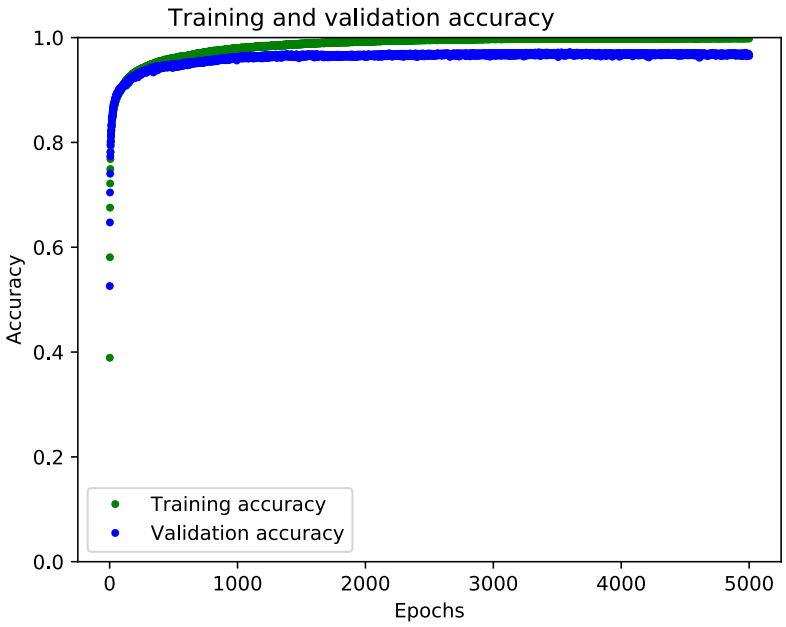

Figure 6.4: Overfitting for 42-32-32-8

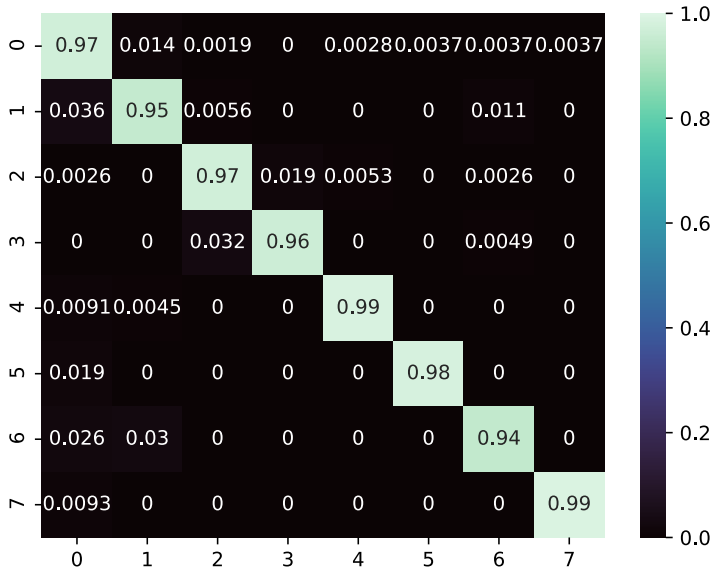

Figure 6.5: Confusion matrix of 42-32-32-32-8 normalized for the true labels in each row

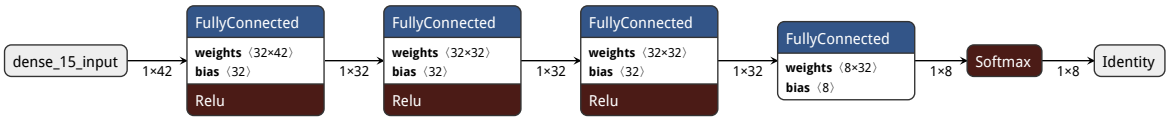

Figure 6.6: Structure of 42-32-32-32-8

Chapter 7

Conclusion

This thesis aimed to answer the question of how microcontrollers can be used for machine learning in the automotive industry.
The related work section has shown how different ML algorithms have been used on embedded devices. Applications with high-resolution images as input and with large NNs require fast embedded hardware and are often deployed on Nvidia Jetson or Drive boards. Less complex problems, however, have been solved using ML on single-core microcontrollers before but NN are only applicable if they are relatively small in size to fit into the little available RAM and to be quick enough at inference.
The use case analysis identified different groups of applications for ML in the automotive industry: car development and production, driver experience, fault diagnosis and predictive maintenance, and mobility services. The topic of fault diagnosis and predictive maintenance seemed very promising for possible applications using ML on microcontrollers and multiple authors were contacted and asked for their opinion on a solution running on a microcontroller, as well as any datasets they might be able to share.
To test possible applications, the single-core microcontroller STM32F746 is used for this thesis. The TensorFlow Lite for Microcontrollers library was chosen since many ML developers already have experience using TensorFlow and existing models can be converted for usage in C++ with the microcontroller. The library is described to be "experimental"[Tenj] by their developers and the problems encountered during the three example programs confirm this verdict. Especially the limited support for quantization was a problem and many quantized models could not be run on the microcontroller. With increased adoption, the quality of this library will hopefully improve in the future.
Finally, the performance of this approach was tested using three example programs. Typical NN architectures used with smartphones or computers like MobileNet proved to be slow and not applicable to many real-time tasks. If such a NN is needed, faster hardware is necessary to achieve low inference times. This matched the observations from the related work section. ADAS based on image data for driver experience is still possible using ML on a microcontroller but the size of the input images needs to be reduced and the objects should fill out a majority of the frame. The UPM Vehicle Image Database fulfilled these criteria and the detection of cars in the low-resolution images was possible using dense NN or CNN with very small convolutional layers. The accuracies of the applicable NNs were sufficient, better than the solution of Arróspide et al.[ASN12], while keeping the inference time low by reducing the number of convolutions for the CNNs. The best dense NN achieved an accuracy of 97.0% with an inference time of 8.6 ms while inference took 11.73 ms for a CNN with an accuracy of 94.8%.
The promising area of fault diagnosis was tested using a dataset shared by Daniel Jung[Jun19]. Dense NN achieved up to 96.8% accurate results for the identification of seven different fault types of an internal combustion engine. The runtime of the inference was also very quick at

only up to 1.7 ms. This proof-of-concept showed that real-time solutions for fault detection and predictive maintenance are possible.
The two successful examples have shown how microcontrollers could be used for machine learning in the automotive industry, the original research question of this thesis. They can also be used as a reference for future applications of which there will be many if the trend of edge machine learning continues.

Appendix A

User Documentation

A.1 Person Detection Example

The code for this example only contains slight modifications to the example found on the official TensorFlow github[Tenf] and is not original work of this thesis. The example was modified to use test images as input, display them on the screen, and output the inference time using the serial communication.
The code found in the folder "PersonDetectionExample" of the USB-Stick included with this thesis can be used to deploy the program on the STM32F746 microcontroller. It is deployed using the ARM mbed OS which provides a way to output text using the serial connection, measure runtime, or display images on the screen. The program can be compiled either using the command line tool mbed-cli[ARMa] or the Mbed Studio IDE[ARMb]. The following text will explain the second option since the Mbed Studio IDE simplifies the process.
After Mbed studio is installed and the GCC_ARM compiler can be used, the folder can be opened as the workspace in Mbed Studio. The IDE recognized when the STM32F746 is connected to a USB port of the computer and offers a run option in addition to the normal build. The active program should be "ObjectRecognition", the target "DISCO-F746NG", and the build profile "Debug". When run is now started, the program is compiled and the binary is automatically copied and deployed on the microcontroller. To see the runtime of the program, open the Serial Monitor using *View > Serial Monitor*. The baud rate needs to be set to 9600. The output should look similar to this:

```
The time taken was 1.1895834*2^2 seconds
person score:107 no person score 215
```

The two inference scores are also displayed on the screen together with the input image. The input image and output results remain on the screen until they are overwritten in the next iteration. Since loading and displaying the image is very quick compared to the inference which takes multiple seconds, the image on the screen is the input for the current inference while the output is from the last cycle. This might seem like the predictions are wrong and opposite, when they actually are not. The input image stored in the input tensor needs to be copied to the display. Unfortunately, the input tensor is modified during inference. Hence, the image needs to be copied before the inference is started which results in the described behavior.

The code to train the model in Python can also be found on github[Tenf] but is not needed since the finished model is already part of the C++ program. The Python script was not run or modified for this thesis.

A.2 UPM Vehicle Image Database

The data, code and results for this example can be found in the folder "ImageRecognition" and subfolders of the USB-Stick included with this thesis.

A.2.1 Python Scripts

The Python scripts are run using Python 3.7.9 and TensorFlow 2.1.0. The full anaconda environment and the list of package versions can be found on the included USB Stick. The Python code was run in PyCharm. The Python code for this example is stored in the folder "ImageRecognitionPython".
The UPM Vehicle Image Database can be found online[Gru] and this thesis uses the version with all 8000 images by adding the selected images from the Caltech Database and the TU Graz-02 Database which can be found on the UPM website as well[Gru]. All 8000 images are stored in *rawImages* and contain subfolders for the different positions they were taken from and the two labels. To perform the train-validate-test split, run *ExtractImages.py*. This copies the images into subfolders of *images* and splits them into 80%, 10%, and 10% of the data as is needed to train the model. Next, the main program *CreateModel.py* can be run. It creates the eleven different models and outputs them as *.tflite* files and C-arrays inside of *.cc* files. Furthermore, it plots the accuracy and loss during training and stores it as *.svg* images. The results of testing are saved as *.txt* files and include all described metrics as well as the confusion matrix. The results described in this thesis are included in the *results* folder. To test the performance of the STM32F746 microcontroller, the images are needed as C-arrays. They were created using *CreateTestImages.py* and some possible input arrays can be found in the folder *TestImages*.

A.2.2 C++ Code

The C++ code for this example is stored in the folder "STM32CubeImageRecognition" and can be deployed on the STM32F746 using the STM32CubeIDE[STMb]. When the project is imported, the debug settings need to be configured correctly. The HCLK Clock is set to 216MHz and all debug settings for the Core Clock need to match this value. To start the program, create a debug configuration matching the screenshots in figures A.1-A.3:

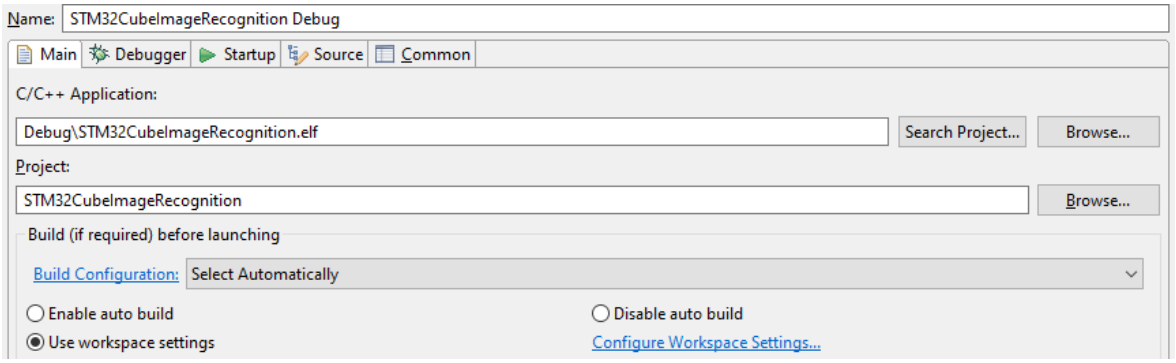

Figure A.1: Debug settings for STM32CubeImageRecognition - Main

Also, check if all source files are part of the Source Lookup Path as should be the case by default. Now the program can be started using the debug configuration. It should stop

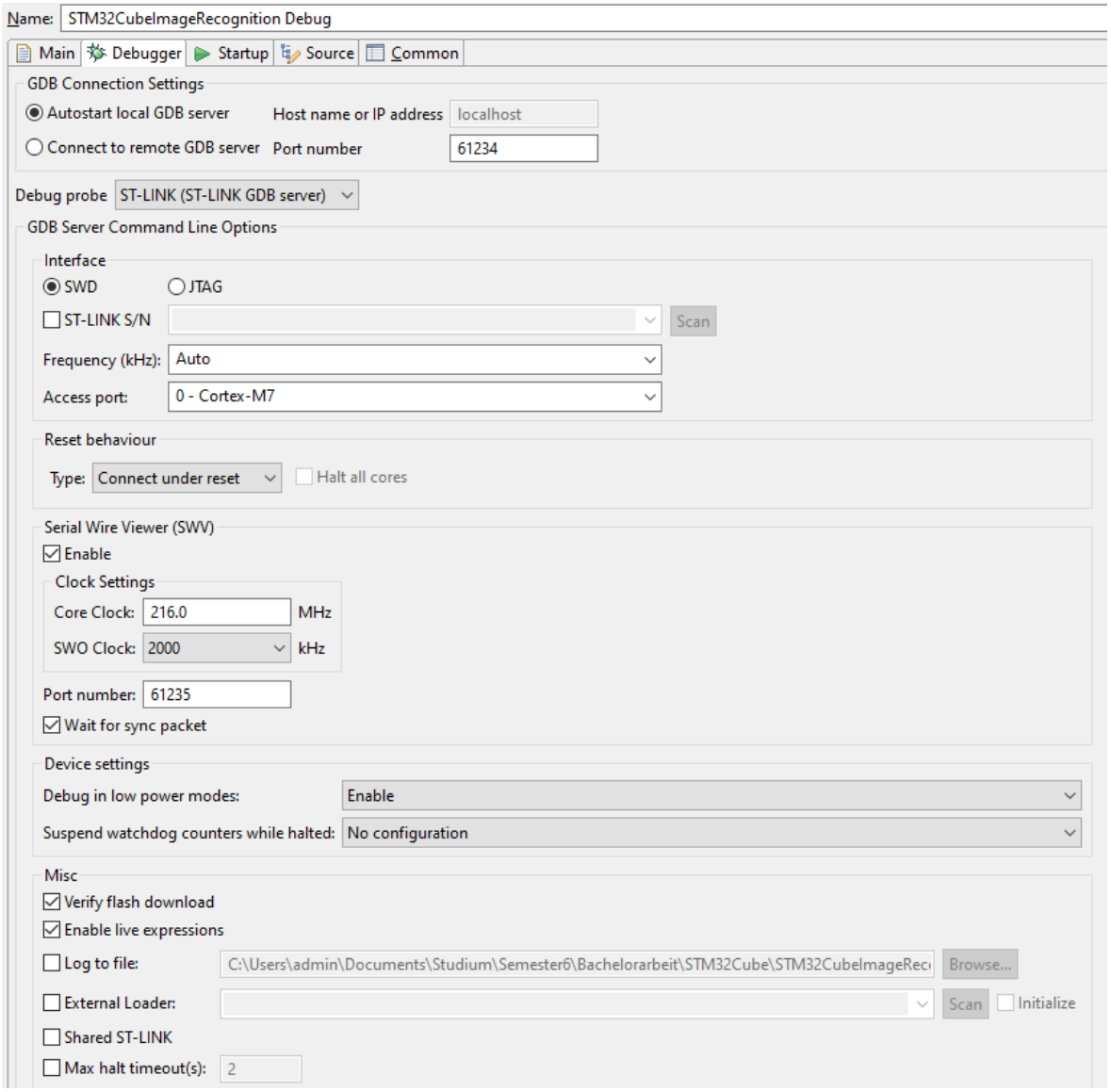

Figure A.2: Debug settings for STM32CubeImageRecognition - Debugger

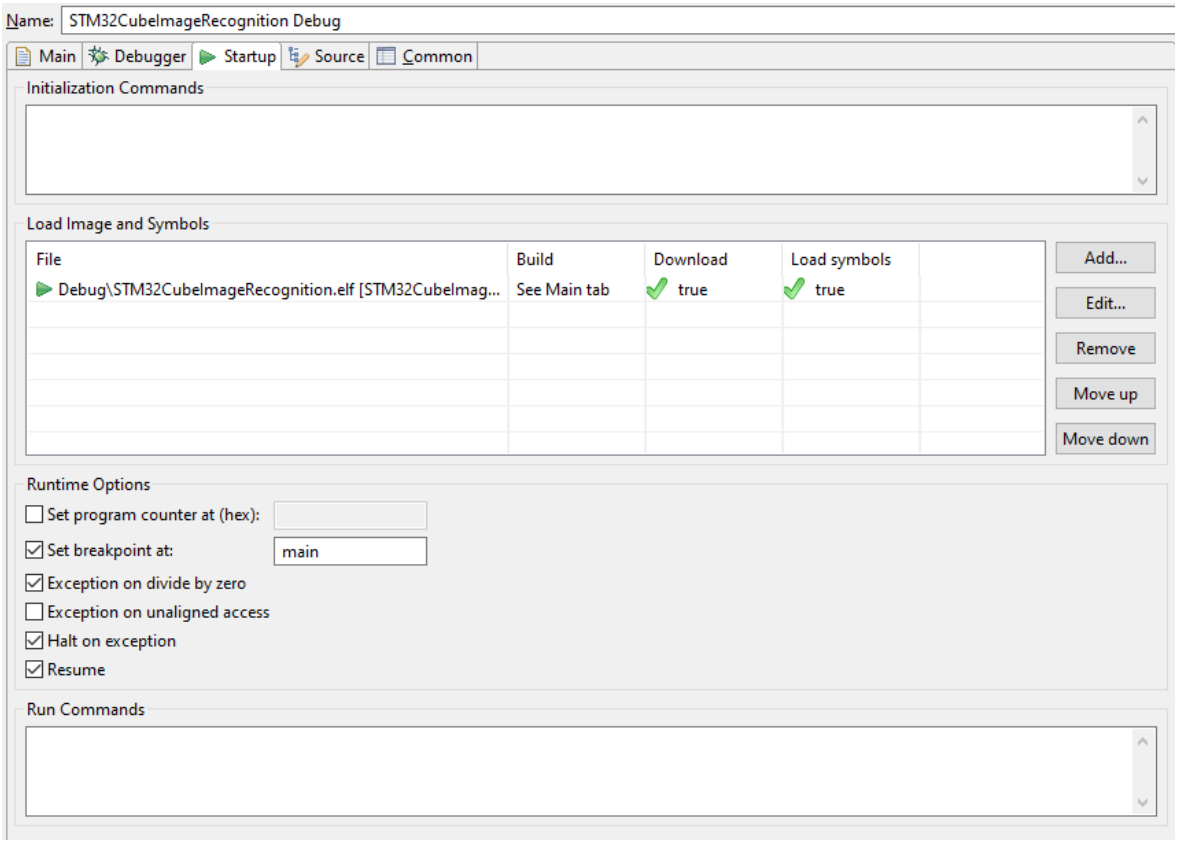

Figure A.3: Debug settings for STM32CubeImageRecognition - Startup

automatically, once the main function is started since a breakpoint is inserted here automatically. At this point, the debug output via the serial connection needs to be activated in the STM32CubeIDE. This is done using the "SWV ITM Data Console" which can be found under *Window > Show View > SWV > SWV ITM Data Console*. This view also contains settings that need to match the configuration of the microcontroller and its clocks. Click on "Configure trace" in the SWV ITM Data Console view and make sure that the settings match the screenshot in figure A.4.

Next, click on "Start Trace" in the SWV ITM Data Console to log the output and continue the program using the debugger (F8). The time measurements should now appear in the SWV ITM Data Console and look similar to this: *It took 1707 milliseconds*.
To display the inference results as well, make sure that the boolean *inferenceOutputActive* is set to true in *main_functions.cc*.
To test different models, copy the C-array created by the Python script into *model.cc* and make sure that the name of the array is changed to *model_tflite* as this is the reference name used in *main_functions.cc*.

A.3 Engine Fault Diagnosis

The data, code and results for this example can be found in the folder "EngineFaultDiagnosis" and subfolders of the USB-Stick included with this thesis. This example uses the dataset shared by Daniel Jung[Jun19]. His work was done in Matlab and he shared the two files *ACC2019classification_runme.m* and *engine_residual_data.mat* with me. The structure of the data was slightly modified for easier use with Python and is now stored in *en-*

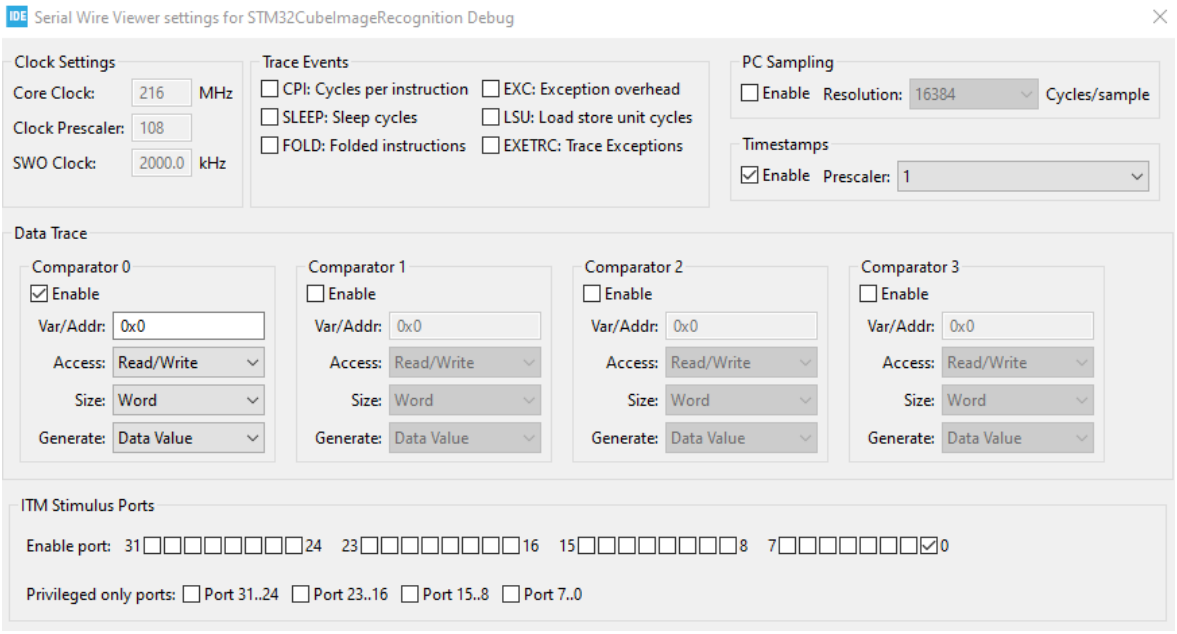

Figure A.4: Serial Wire Viewer settings for STM32CubeImageRecognition

gine_residuals.mat but the measurements themselves were not changed at all.

A.3.1 Python Scripts

The Python scripts are run using Python 3.7.9 and TensorFlow 2.1.0. The full anaconda environment and the list of package versions can be found on the included USB Stick. The Python code was run in PyCharm. The Python code for this example is stored in the folder "ImageRecognitionPython".
First, the data stored in the matlab file *engine_residuals.mat* is converted to a CSV file for easier use with the Python script *CreateCSV.py*.
Next, the main Python script *CreateModel.py* can be run. To switch between 9 and 42 residuals, change the *residualSelection* parameter of *load_data*. If *residualSelection* is true, only 9 residuals are selected, else all 42 ones are being used. The script creates the six different models and outputs them as *.tflite* files and C-arrays inside of *.cc* files. Furthermore, it plots the accuracy and loss during training and stores it as *.svg* images. The results of testing are saved as *.txt* files and include all described metrics as well as the confusion matrix. The results described in this thesis using only 9 residuals are included in the *Results9Residuals* folder. The results for all 42 residuals can be found in the *Results42Residuals folder*.
Since visualization of the confusion matrix was helpful for this thesis, the script *ConfusionMatrix.py* is included. It evaluates the model in the same way as *CreateModel.py* but creates a *.svg* file for the confusion matrix.

A.3.2 C++ Code

The C++ code for this example is stored in the folder "STM32CubeEngineFaultDiagnosis" and can be deployed on the STM32F746 using the STM32CubeIDE[STMb]. When the project is imported, the debug settings need to be configured correctly. The HCLK Clock is set to 216MHz and all debug settings for the Core Clock need to match this value. To start the program, create a debug configuration matching the screenshots in figures A.5-A.7:

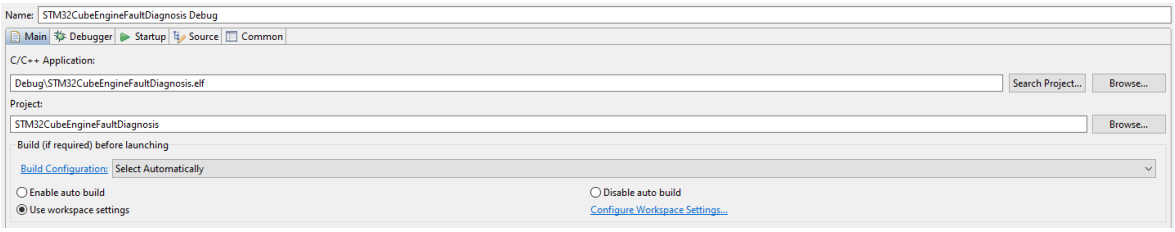

Figure A.5: Debug settings for STM32CubeEngineFaultDiagnosis - Main

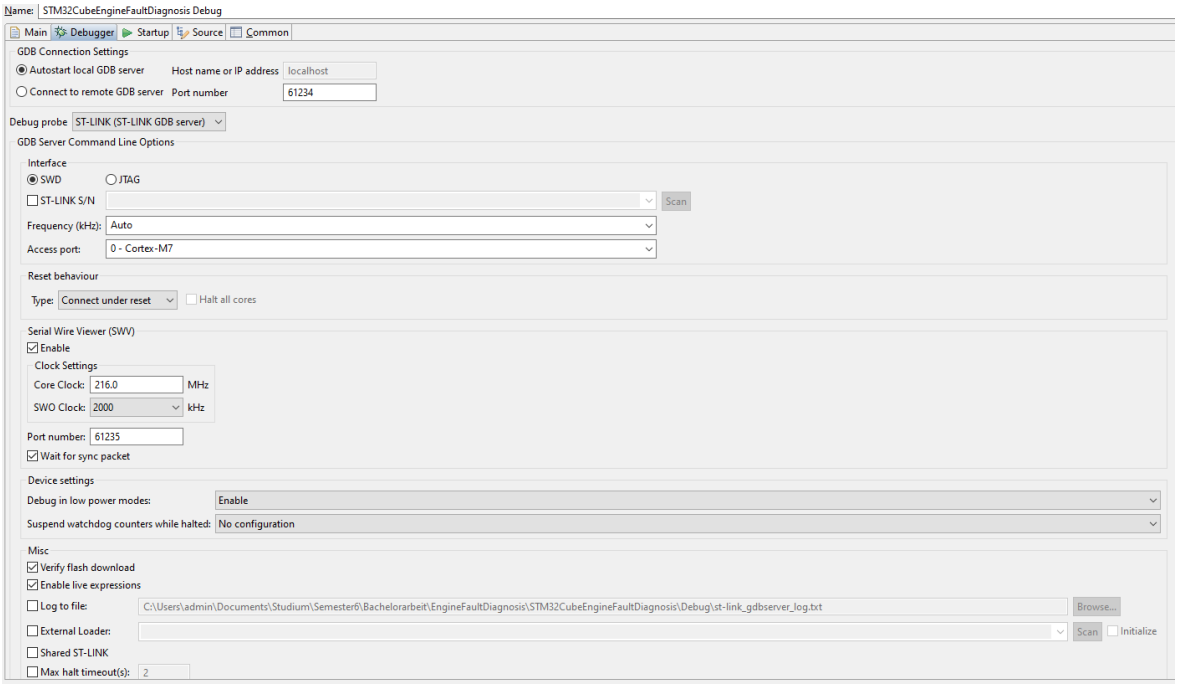

Figure A.6: Debug settings for STM32CubeEngineFaultDiagnosis - Debugger

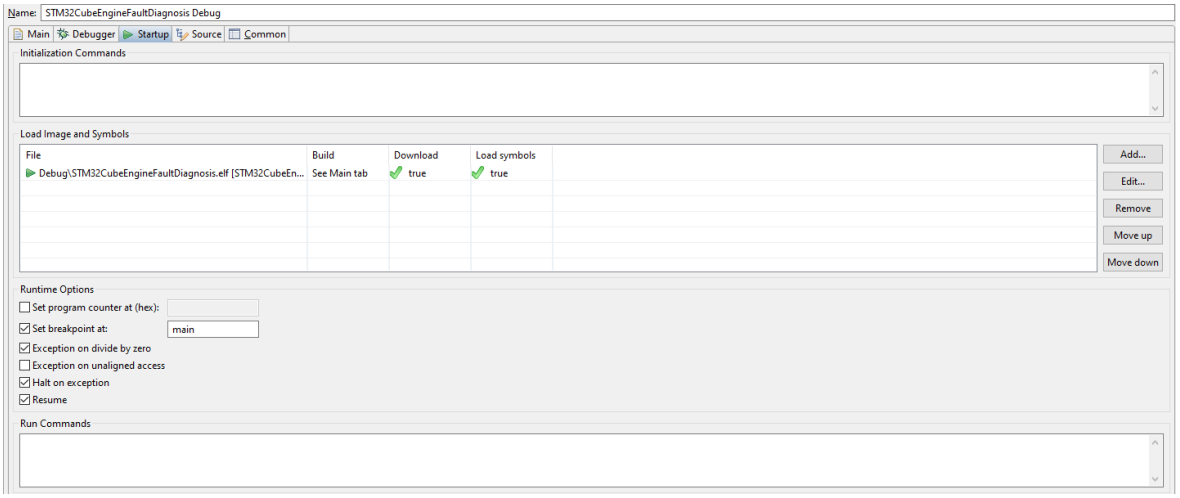

Figure A.7: Debug settings for STM32CubeEngineFaultDiagnosis - Startup

Also, check if all source files are part of the Source Lookup Path as should be the case by default. Now the program can be started using the debug configuration. The output works in the same way as for the previous example. Make sure that the trace configuration of the SWV ITM Data Console matches the screenshot in figure A.8.

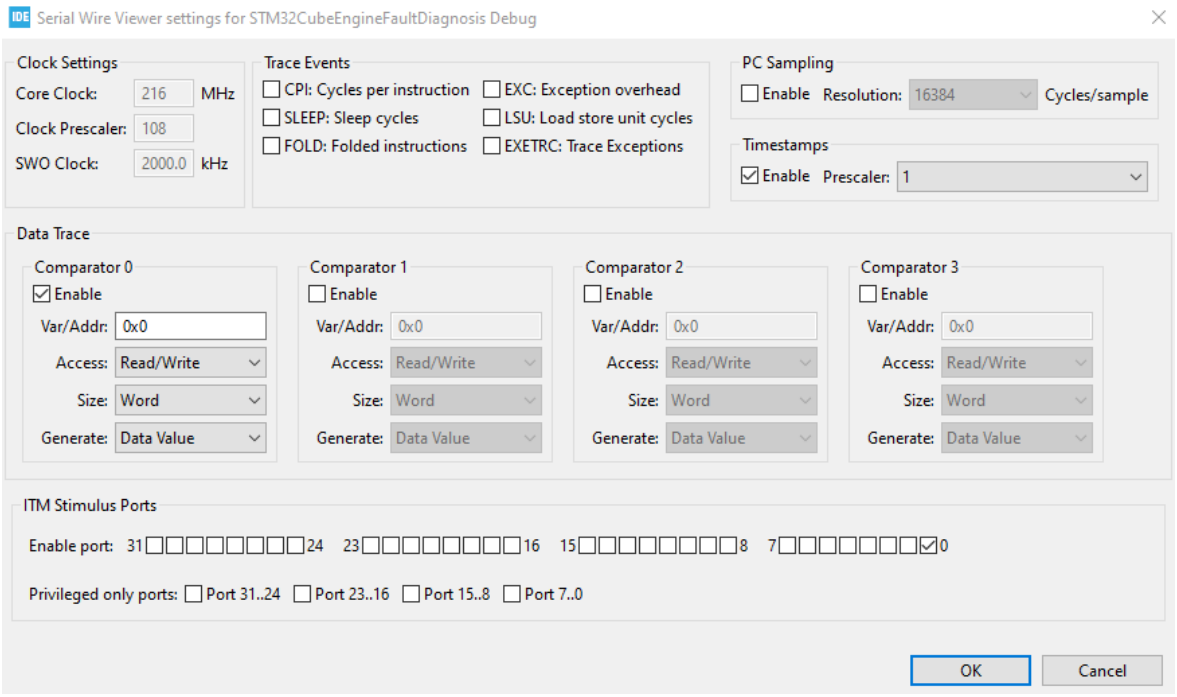

Figure A.8: Serial Wire Viewer settings for STM32CubeEngineFaultDiagnosis

Next, click on "Start Trace" in the SWV ITM Data Console to log the output and continue the program using the debugger (F8). The time measurements should now appear in the SWV ITM Data Console and look similar to this: *It took 16995 milliseconds*.
To display the inference results as well, make sure that the debug output is not commented out in line 144 of *main_functions.cc*.
To test different models, copy the C-array created by the Python script into *model.cc* and make sure that the name of the array is changed to *model_tflite* as this is the reference name used in *main_functions.cc*.

Bibliography

[Ahm+15] Ahmed, R., El Sayed, M., Gadsden, S. A., Tjong, J., and Habibi, S. "Automotive Internal-Combustion-Engine Fault Detection and Classification Using Artificial Neural Network Techniques". In: *IEEE Transactions on Vehicular Technology* 64.1 (2015), pp. 21–33.

[Alf+17] Alfian, G., Rhee, J., Ijaz, M., Syafrudin, M., and Fitriyani, N. "Performance Analysis of a Forecasting Relocation Model for One-Way Carsharing". In: *Applied Sciences* 7.6 (June 2017), p. 598. DOI: 10.3390/app7060598. URL: http://dx.doi.org/10.3390/app7060598.

[Ang+07] Anguita, D., Ghio, A., Pischiutta, S., and Ridella, S. "A Hardware-friendly Support Vector Machine for Embedded Automotive Applications". In: *2007 International Joint Conference on Neural Networks*. IEEE, Aug. 2007. DOI: 10.1109/ijcnn.2007.4371156. URL: http://dx.doi.org/10.1109/IJCNN.2007.4371156.

[Ard] Arduino. *Language Reference*. URL: https://www.arduino.cc/reference/en/ (visited on 08/11/2020).

[ARMa] ARM. *Arm Mbed CLI*. URL: https://github.com/ARMmbed/mbed-cli (visited on 11/06/2020).

[ARMb] ARM. *Mbed Studio*. URL: https://os.mbed.com/studio/ (visited on 11/06/2020).

[Arm] Arm Limited. *Arm Cortex-M Series Processors*. URL: https://developer.arm.com/ip-products/processors/cortex-m (visited on 08/11/2020).

[ASN12] Arróspide, J., Salgado, L., and Nieto, M. "Video analysis-based vehicle detection and tracking using an MCMC sampling framework". In: *EURASIP Journal on Advances in Signal Processing* 2012.1 (Jan. 2012). DOI: 10.1186/1687-6180-2012-2. URL: http://dx.doi.org/10.1186/1687-6180-2012-2.

[Bat17] Batra, D. *CS 7643: Deep Learning*. 2017. URL: https://www.cc.gatech.edu/classes/AY2018/cs7643_fall/slides/L6_cnns_annotated.pdf (visited on 10/20/2020).

[Ben18] Beningo, J. *5 Reasons Embedded Developers Should Care about Machine Learning*. July 23, 2018. URL: https://www.designnews.com/artificial-intelligence/5-reasons-embedded-developers-should-care-about-machine-learning (visited on 10/09/2020).

[Ber03] Berwick, R. *An Idiot's guide to Support vector machines (SVMs)*. 2003. URL: http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf (visited on 10/17/2020).

[Bha] Bhalla, D. *Support Vector Machine Simplified Using R*. URL: https://www.listendata.com/2017/01/support-vector-machine-in-r-tutorial.html? (visited on 11/09/2020).

[Bla] BlaBlaCar. *BlaBlaCar*. URL: https://www.blablacar.de/ (visited on 09/17/2020).

[BMW] BMW Group. *Autonomous Driving*. URL: https://www.bmwgroup.com/en/innovation/technologies-and-mobility/autonomes-fahren.html (visited on 11/03/2020).

[Bor+17] Borghi, G., Gasparini, R., Vezzani, R., and Cucchiara, R. "Embedded recurrent network for head pose estimation in car". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, June 2017. DOI: 10.1109/ivs.2017.7995922. URL: http://dx.doi.org/10.1109/IVS.2017.7995922.

[Bor+20] Borrego-Carazo, J., Castells-Rufas, D., Biempica, E., and Carrabina, J. "Resource-Constrained Machine Learning for ADAS: A Systematic Review". In: *IEEE Access* 8 (2020), pp. 40573–40598. DOI: 10.1109/access.2020.2976513. URL: http://dx.doi.org/10.1109/ACCESS.2020.2976513.

[Boy19] Boyland, P. *The state of mobility network experience*. May 2019. URL: https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2019-05/the_state_of_mobile_experience_may_2019_0.pdf (visited on 11/03/2020).

[BRK17] Brendel, A. B., Rockenkamm, C., and Kolbe, L. M. "Generating Rental Data for Car Sharing Relocation Simulations on the Example of Station-Based One-Way Car Sharing". In: *HICSS*. 2017.

[Bur19] Burkert, A. "Artificial Intelligence Conquers the Microcontroller". In: *ATZelectronics worldwide* 14.4 (Apr. 2019), pp. 56–59. DOI: 10.1007/s38314-019-0035-3. URL: http://dx.doi.org/10.1007/s38314-019-0035-3.

[CPS06] Chellapilla, K., Puri, S., and Simard, P. "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Lorette, G. http://www.suvisoft.com. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. URL: https://hal.inria.fr/inria-00112631.

[CPC20] Chicco, A., Pirra, M., and Carboni, A. "Preliminary Investigation of Women Car Sharing Perceptions Through a Machine Learning Approach". In: *Communications in Computer and Information Science*. Springer International Publishing, 2020, pp. 622–630. DOI: 10.1007/978-3-030-50726-8_81. URL: http://dx.doi.org/10.1007/978-3-030-50726-8_81.

[Cho+19] Chowdhery, A., Warden, P., Shlens, J., Howard, A., and Rhodes, R. *Visual Wake Words Dataset*. 2019. eprint: arXiv:1906.05721.

[CV95] Cortes, C. and Vapnik, V. "Support-vector networks". In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. DOI: 10.1007/bf00994018. URL: http://dx.doi.org/10.1007/BF00994018.

[CH67] Cover, T. and Hart, P. "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* 13.1 (Jan. 1967), pp. 21–27. DOI: 10.1109/tit.1967.1053964. URL: http://dx.doi.org/10.1109/TIT.1967.1053964.

[Dai] Daimler. *Autonomous Driving*. URL: https://www.daimler.com/innovation/product-innovation/autonomous-driving/ (visited on 11/03/2020).

[Dar+20] Daraio, E., Cagliero, L., Chiusano, S., Garza, P., and Giordano, D. "Predicting Car Availability in Free Floating Car Sharing Systems: Leveraging Machine Learning in Challenging Contexts". In: *Electronics* 9.8 (Aug. 2020), p. 1322. DOI: 10.3390/electronics9081322. URL: http://dx.doi.org/10.3390/electronics9081322.

[Den+19] Dendaluce Jahnke, M., Cosco, F., Novickis, R., Pérez Rastelli, J., and Gomez-Garay, V. "Efficient Neural Network Implementations on Parallel Embedded Platforms Applied to Real-Time Torque-Vectoring Optimization Using Predictions for Multi-Motor Electric Vehicles". In: *Electronics* 8.2 (Feb. 2019), p. 250. DOI: 10.3390/electronics8020250. URL: http://dx.doi.org/10.3390/electronics8020250.

[Den+20] Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S., and Zomaya, A. Y. "Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence". In: *IEEE Internet of Things Journal* 7.8 (Aug. 2020), pp. 7457–7469. DOI: 10.1109/jiot.2020.2984887. URL: http://dx.doi.org/10.1109/JIOT.2020.2984887.

[Dev19] Developer Economics. *The battle: Tensorflow vs Pytorch*. 2019. URL: https://medium.com/@DevEconomics/the-battle-tensorflow-vs-pytorch-61d90b04f33b (visited on 06/18/2020).

[Eat17] Eaton, E. *Introduction to Machine Learning*. 2017. URL: https://www.seas.upenn.edu/~cis519/fall2017/lectures/01_introduction.pdf (visited on 10/18/2020).

[EBE19] Einabadi, B., Baboli, A., and Ebrahimi, M. "Dynamic Predictive Maintenance in industry 4.0 based on real time information: Case study in automotive industries". In: *IFAC-PapersOnLine* 52.13 (2019). 9th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2019, pp. 1069–1074. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2019.11.337. URL: http://www.sciencedirect.com/science/article/pii/S2405896319313151.

[Fak+19] Fakhar, S., Saad, M., Fauzan, A., Affendi, R., and Aidil, M. "Development of portable automatic number plate recognition (ANPR) system on Raspberry Pi". In: *International Journal of Electrical and Computer Engineering (IJECE)* 9.3 (June 2019), p. 1805. DOI: 10.11591/ijece.v9i3.pp1805-1813. URL: http://dx.doi.org/10.11591/ijece.v9i3.pp1805-1813.

[Far+10] Farooq, U., Amar, M., Haq, E. ul, Asad, M. U., and Atiq, H. M. "Microcontroller Based Neural Network Controlled Low Cost Autonomous Vehicle". In: *2010 Second International Conference on Machine Learning and Computing*. IEEE, 2010. DOI: 10.1109/icmlc.2010.71. URL: http://dx.doi.org/10.1109/ICMLC.2010.71.

[Fle11] Fleming, B. "Microcontroller Units in Automobiles [Automotive Electronics]". In: *IEEE Vehicular Technology Magazine* 6.3 (Sept. 2011), pp. 4–8. DOI: 10.1109/mvt.2011.941888. URL: http://dx.doi.org/10.1109/MVT.2011.941888.

[For] Ford Mobility. *Autonomous Vehicles*. URL: https://www.ford-mobility.eu/autonomous-vehicles (visited on 11/03/2020).

[FK18] Frisk, E. and Krysander, M. "Residual Selection for Consistency Based Diagnosis Using Machine Learning Models". In: *IFAC-PapersOnLine* 51.24 (2018). 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2018, pp. 139–146. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.09.547. URL: http://www.sciencedirect.com/science/article/pii/S2405896318322389.

[FKJ17] Frisk, E., Krysander, M., and Jung, D. "A Toolbox for Analysis and Design of Model Based Diagnosis Systems for Large Scale Models". In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 3287–3293. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2017.08.504. URL: http://www.sciencedirect.com/science/article/pii/S2405896317308728.

[FRP18] Fritzsche, R., Richter, A., and Putz, M. "Product Flexible Car Body Fixtures with Position-dependent Load Balancing Based on Finite Element Method in Combination with Methods of Artificial Intelligence". In: *Procedia CIRP* 67 (2018), pp. 452–457. DOI: 10.1016/j.procir.2017.12.241. URL: http://dx.doi.org/10.1016/j.procir.2017.12.241.

[Fuk80] Fukushima, K. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. DOI: 10.1007/bf00344251. URL: http://dx.doi.org/10.1007/BF00344251.

[Für08] Fürnkranz, J. *Decision-Tree Learning*. 2008. URL: https://www.ke.tu-darmstadt.de/lehre/archiv/ws0809/mldm/dt.pdf (visited on 10/17/2020).

[GK16] Gaffar, A. and Kouchak, S. M. “Using artificial intelligence to automatically customize modern car infotainment systems”. In: *Proceedings of the 2016 International Conference on Artificial Intelligence, ICAI 2016 - WORLDCOMP 2016*. Proceedings of the 2016 International Conference on Artificial Intelligence, ICAI 2016 - WORLDCOMP 2016. CSREA Press, Jan. 2016, pp. 151–156.

[Gee20] Geeks for Geeks. *Understanding of LSTM Networks*. Nov. 5, 2020. URL: https://www.geeksforgeeks.org/understanding-of-lstm-networks/ (visited on 10/20/2020).

[Ger19] Gerstl, S. *Is AI ready for use in IoT, automotive and industry?* Aug. 27, 2019. URL: https://www.intelligent-mobility-xperience.com/is-ai-ready-for-use-in-iot-automotive-and-industry-a-859272/ (visited on 10/09/2020).

[Gio+18] Giobergia, F., Baralis, E., Camuglia, M., Cerquitelli, T., Mellia, M., Neri, A., Tricarico, D., and Tuninetti, A. “Mining Sensor Data for Predictive Maintenance in the Automotive Industry”. In: (Oct. 2018), pp. 351–360. DOI: 10.1109/DSAA.2018.00046.

[Goo] Google. *What is Google Lens?* URL: https://lens.google.com/howlensworks/ (visited on 11/03/2020).

[Gru] Grupo de Tratamiento de Imágenes (GTI), E.T.S.Ing. Telecomunicación Universidad Politécnica de Madrid (UPM). *Vehicle Image Database*. URL: http://www.gti.ssr.upm.es/data/Vehicle_database.html (visited on 10/28/2020).

[GRF07] Gusikhin, O., Rychtyckyj, N., and Filev, D. “Intelligent systems in the automotive industry: applications and trends”. In: *Knowledge and Information Systems* 12.2 (2007), pp. 147–168. ISSN: 0219-3116. DOI: 10.1007/s10115-006-0063-1. URL: https://doi.org/10.1007/s10115-006-0063-1.

[Har16] Hardawar, D. *NVIDIA’s Drive PX 2 is a liquid-cooled supercomputer for cars*. Jan. 4, 2016. URL: https://www.engadget.com/2016-01-04-nvidia-drive-px2.html (visited on 10/13/2020).

[HW17] Hasenjager, M. and Wersing, H. “Personalization in advanced driver assistance systems and autonomous vehicles: A review”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, Oct. 2017. DOI: 10.1109/itsc.2017.8317803. URL: http://dx.doi.org/10.1109/ITSC.2017.8317803.

[HS97] Hochreiter, S. and Schmidhuber, J. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[Hom18] Home Lab. *Microcontrollers and Robotics*. Dec. 27, 2018. URL: http://home.roboticlab.eu/en/microcontrollers (visited on 11/03/2020).

[HYC18] Hongyu, A. W., Yupeng, B. Y., and Chuamg, C. M. “A Modeling Method of Whole Vehicle Electrical Balance Simulation System Based on Neural Network Training”. In: *IFAC-PapersOnLine* 51.31 (2018), pp. 87–91. DOI: 10.1016/j.ifacol.2018.10.017. URL: http://dx.doi.org/10.1016/j.ifacol.2018.10.017.

[Hop84] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons." In: *Proceedings of the National Academy of Sciences* 81.10 (May 1984), pp. 3088–3092. DOI: 10.1073/pnas.81.10.3088. URL: http://dx.doi.org/10.1073/pnas.81.10.3088.

[How+17] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. URL: https://arxiv.org/abs/1704.04861v1.

[HW59] Hubel, D. H. and Wiesel, T. N. "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of Physiology* 148.3 (Oct. 1959), pp. 574–591. DOI: 10.1113/jphysiol.1959.sp006308. URL: http://dx.doi.org/10.1113/jphysiol.1959.sp006308.

[JMB14] Jafari, S. M., Mehdigholi, H., and Behzad, M. "Valve Fault Diagnosis in Internal Combustion Engines Using Acoustic Emission and Artificial Neural Network". In: *Shock and Vibration* 2014 (2014), p. 823514. ISSN: 1070-9622. DOI: 10.1155/2014/823514. URL: https://doi.org/10.1155/2014/823514.

[Joh] John, S. *Vehicle Image DatabaseHow to connect your Amazon Echo to Wi-Fi to get the most out of your Alexa-enabled smart speaker*. URL: https://www.businessinsider.com/how-to-connect-amazon-echo-to-wifi (visited on 11/03/2020).

[Jun19] Jung, D. "Engine Fault Diagnosis Combining Model-based Residuals and Data-Driven Classifiers". In: *IFAC-PapersOnLine* 52.5 (2019). 9th IFAC Symposium on Advances in Automotive Control AAC 2019, pp. 285–290. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2019.09.046. URL: http://www.sciencedirect.com/science/article/pii/S2405896319306664.

[JRR18] Junior, E. L. L., Rosa, R. L., and Rodriguez, D. Z. "A Recommendation System for Shared-Use Mobility Service". In: *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, Sept. 2018. DOI: 10.23919/softcom.2018.8555845. URL: http://dx.doi.org/10.23919/SOFTCOM.2018.8555845.

[Kat+18] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y., and Azumi, T. "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems". In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, Apr. 2018. DOI: 10.1109/iccps.2018.00035. URL: http://dx.doi.org/10.1109/ICCPS.2018.00035.

[Kha+20] Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. "A survey of the recent architectures of deep convolutional neural networks". In: *Artificial Intelligence Review* 53.8 (Apr. 2020), pp. 5455–5516. DOI: 10.1007/s10462-020-09825-6. URL: http://dx.doi.org/10.1007/s10462-020-09825-6.

[Kim+17] Kim, I.-H., Bong, J.-H., Park, J., and Park, S. "Prediction of Driver's Intention of Lane Change by Augmenting Sensor Information Using Machine Learning Techniques". In: *Sensors* 17.6 (June 2017). DOI: 10.3390/s17061350. URL: http://dx.doi.org/10.3390/s17061350.

[KJD19] Kocić, J., Jovičić, N., and Drndarević, V. "An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms". In: *Sensors* 19.9 (May 2019), p. 2064. DOI: 10.3390/s19092064. URL: http://dx.doi.org/10.3390/s19092064.

[KGV17] Kumar, A., Goyal, S., and Varma, M. "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things". In: *ICML*. 2017.

[KK12] Kumar, R. and Khare, N. "Smart Instrument for Automotive Battery Application". In: *2012 Fourth International Conference on Computational Intelligence and Communication Networks*. IEEE, Nov. 2012. DOI: 10.1109/cicn.2012.186. URL: http://dx.doi.org/10.1109/CICN.2012.186.

[Lam16] Lambert, F. *All new Teslas are equipped with NVIDIA's new Drive PX 2 AI platform for self-driving*. Oct. 21, 2016. URL: https://electrek.co/2016/10/21/all-new-teslas-are-equipped-with-nvidias-new-drive-px-2-ai-platform-for-self-driving (visited on 10/13/2020).

[LeC+90] LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems 2*. Ed. by Touretzky, D. S. Morgan-Kaufmann, 1990, pp. 396–404. URL: http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.pdf.

[Lia16] Liang, Y. *Machine Learning Basics Lecture 6: Overfitting*. 2016. URL: https://www.cs.princeton.edu/courses/archive/spring16/cos495/slides/ML_basics_lecture6_overfitting.pdf (visited on 10/18/2020).

[Lin+14] Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. *Microsoft COCO: Common Objects in Context*. 2014. eprint: arXiv:1405.0312.

[Lu+18] Lu, K., Li, J., Zhou, L., Hu, X., An, X., and He, H. "Generalized Haar Filter-Based Object Detection for Car Sharing Services". In: *IEEE Transactions on Automation Science and Engineering* 15.4 (Oct. 2018), pp. 1448–1458. DOI: 10.1109/tase.2018.2830655. URL: http://dx.doi.org/10.1109/TASE.2018.2830655.

[Luc+18] Luckow, A., Kennedy, K., Ziolkowski, M., Djerekarov, E., Cook, M., Duffy, E., Schleiss, M., Vorster, B., Weill, E., Kulshrestha, A., and al., et. "Artificial Intelligence and Deep Learning Applications for Automotive Manufacturing". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, Dec. 2018. DOI: 10.1109/bigdata.2018.8622357. URL: http://dx.doi.org/10.1109/BigData.2018.8622357.

[Luo+19] Luo, M., Wen, H., Luo, Y., Du, B., Klemmer, K., and Zhu, H. "Dynamic Demand Prediction for Expanding Electric Vehicle Sharing Systems: A Graph Sequence Learning Approach". In: *CoRR* abs/1903.04051 (2019). URL: http://arxiv.org/abs/1903.04051.

[Mar20] Marius, H. *Multiclass Classification with Support Vector Machines (SVM), Dual Problem and Kernel Functions*. June 9, 2020. URL: https://towardsdatascience.com/multiclass-classification-with-support-vector-machines-svm-kernel-trick-kernel-functions-f9d5377d6f02 (visited on 10/18/2020).

[MPI20] Merenda, M., Porcaro, C., and Iero, D. "Edge Machine Learning for AI-Enabled IoT Devices: A Review". In: *Sensors* 20.9 (Apr. 2020), p. 2533. DOI: 10.3390/s20092533. URL: http://dx.doi.org/10.3390/s20092533.

[Mic] Michelin Noth America Inc. *Michelin Tire Care*. URL: https://www.michelintruck.com/services-and-programs/tire-care/ (visited on 09/12/2020).

[Mic20] Microsoft. *Deep learning vs. machine learning in Azure Machine Learning*. Sept. 22, 2020. URL: https://docs.microsoft.com/en-us/azure/machine-learning/concept-deep-learning-vs-machine-learning (visited on 10/20/2020).

[Moo+16] Moosavi, S. S., Djerdir, A., Ait-Amirat, Y., Arab Khaburi, D., and N'Diaye, A. "Artificial neural network-based fault diagnosis in the AC–DC converter of the power supply of series hybrid electric vehicle". In: *IET Electrical Systems in Transportation* 6.2 (2016), pp. 96–106.

[OAK19] Othman, N. A., Aydin, I., and Karakose, M. "An Efficient Embedded Security System for Reduce Car Accident to Build Safer World Based On IoT". In: *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE, Sept. 2019. DOI: 10.1109/idap.2019.8875933. URL: http://dx.doi.org/10.1109/IDAP.2019.8875933.

[Pet19] Pete Warden, D. S. *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers*. O'Reilly Media, 2019. ISBN: 1492052043.

[Pir+19] Pirasteh, P., Nowaczyk, S., Pashami, S., Löwenadler, M., Thunberg, K., Ydreskog, H., and Berck, P. "Interactive feature extraction for diagnostic trouble codes in predictive maintenance: A case study from automotive domain". In: (Feb. 2019), pp. 1–10. DOI: 10.1145/3304079.3310288.

[Pla+18] Plastiras, G., Terzi, M., Kyrkou, C., and Theocharidcs, T. "Edge Intelligence: Challenges and Opportunities of Near-Sensor Machine Learning Applications". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, July 2018. DOI: 10.1109/asap.2018.8445118. URL: http://dx.doi.org/10.1109/ASAP.2018.8445118.

[Pry+15] Prytz, R., Nowaczyk, S., Rögnvaldsson, T., and Byttner, S. "Predicting the need for vehicle compressor repairs using maintenance records and logged vehicle data". In: *Engineering Applications of Artificial Intelligence* 41 (2015), pp. 139–150. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2015.02.009. URL: http://www.sciencedirect.com/science/article/pii/S0952197615000391.

[Rasa] Raspberry Pi Foundation. *Raspberry Pi Documentation - FAQ*. URL: https://www.raspberrypi.org/documentation/faqs (visited on 10/15/2020).

[Rasb] Raspberry Pi Geek. *Der Raspberry Pi 2 und sein Vorgänger im Vergleich*. URL: https://www.raspberry-pi-geek.de/ausgaben/rpg/2015/03/der-raspberry-pi-2-und-sein-vorgaenger-im-vergleich/2/ (visited on 10/15/2020).

[Ros58] Rosenblatt, F. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[RN09] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. 2009. ISBN: 0136042597.

[Sak+20] Sakr, F., Bellotti, F., Berta, R., and De Gloria, A. "Machine Learning on Mainstream Microcontrollers". In: *Sensors* 20.9 (May 2020), p. 2638. DOI: 10.3390/s20092638. URL: http://dx.doi.org/10.3390/s20092638.

[Sha+18] Shafi, U., Safi, A., Shahid, A. R., Ziauddin, S., and Saleem, M. Q. "Vehicle Remote Health Monitoring and Prognostic Maintenance System". en. In: *Journal of Advanced Transportation* (2018). DOI: https://doi.org/10.1155/2018/8061514. URL: https://www.hindawi.com/journals/jat/2018/8061514/ (visited on 04/13/2020).

[Sha] Shalizi, C. *k-Nearest Neighbors*. URL: http://www.stat.cmu.edu/~cshalizi/dm/19/lectures/09/lecture-09.html (visited on 10/17/2020).

[Shi+17] Shin, J., Kim, U., Lee, D., Park, S., Oh, S., and Yun, T. “Real-time vehicle detection using deep learning scheme on embedded system”. In: *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2017, pp. 272–274.

[ŞBU16] Şimşir, M., Bayır, R., and Uyaroğlu, Y. “Real-Time Monitoring and Fault Diagnosis of a Low Power Hub Motor Using Feedforward Neural Network”. In: *Computational Intelligence and Neuroscience* 2016 (2016), pp. 1–13. DOI: 10.1155/2016/7129376. URL: http://dx.doi.org/10.1155/2016/7129376.

[Sri18] Srivastava, T. *Introduction to k-Nearest Neighbors*. Mar. 26, 2018. URL: https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/ (visited on 11/09/2020).

[STMa] STMicroelectronics. *32F746GDISCOVERY*. URL: https://www.st.com/en/evaluation-tools/32f746gdiscovery.html (visited on 08/05/2020).

[STMb] STMicroelectronics. *STM32CubeIDE*. URL: https://www.st.com/en/development-tools/stm32cubeide.html (visited on 11/06/2020).

[STMc] STMicroelectronics. *STM32F746xx datasheet*. URL: https://github.com/tensorflow/models (visited on 08/11/2020).

[SL19] Suda, N. and Loh, D. “Machine Learning on Arm Cortex-M Microcontrollers”. In: (2019).

[Teca] Technative. *Artificial Intelligence: Taking driverless navigation up a gear*. URL: https://www.technative.io/artificial-intelligence-taking-driverless-navigation-up-a-gear/ (visited on 09/11/2020).

[Tecb] Techopedia. *Dictionary - Embedded Device*. URL: https://www.techopedia.com/definition/10179/embedded-device (visited on 10/07/2020).

[Tena] TensorFlow. *all_ops_resolver.cc*. URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/all_ops_resolver.cc (visited on 08/04/2020).

[Tenb] TensorFlow. *Issue - TF Micro requires CONV_2D version ’2’ when applying quantization*. URL: https://github.com/tensorflow/tensorflow/issues/32468 (visited on 11/08/2020).

[Tenc] TensorFlow. *Issue - TFLu wrong predictions for optimized model*. URL: https://github.com/tensorflow/tensorflow/issues/40866 (visited on 11/08/2020).

[Tend] TensorFlow. *Model Garden for TensorFlow*. URL: https://github.com/tensorflow/models (visited on 08/07/2020).

[Tene] TensorFlow. *Open Pull Request - Tflite backend*. URL: https://github.com/uTensor/utensor_cgen/pull/93 (visited on 11/08/2020).

[Tenf] TensorFlow. *Person detection example*. URL: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro/examples/person_detection (visited on 08/07/2020).

[Teng] TensorFlow. *Person detection example - Training a model*. URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/person_detection/training_a_model.md (visited on 08/07/2020).

[Tenh] TensorFlow. *Post-training quantization*. URL: https://www.tensorflow.org/lite/performance/post_training_quantization (visited on 11/08/2020).

[Teni] TensorFlow. *Quantization aware training in Keras example*. URL: https://www.tensorflow.org/model_optimization/guide/quantization/training_example#see_4x_smaller_model_from_quantization (visited on 10/27/2020).

[Tenj] TensorFlow. *TensorFlow Lite for Microcontrollers*. URL: https://www.tensorflow.org/lite/microcontrollers (visited on 06/18/2020).

[Tenk] TensorFlow. *TensorFlow Lite for Microcontrollers - Build and convert models*. URL: https://www.tensorflow.org/lite/microcontrollers/build_convert#operation_support (visited on 08/04/2020).

[Tenl] TensorFlow. *TensorFlow Lite for Microcontrollers - Get started with microcontrollers*. URL: https://www.tensorflow.org/lite/microcontrollers/get_started (visited on 08/04/2020).

[Tes] Tesla. *Tesla Autopilot - Future of Driving*. URL: https://www.tesla.com/autopilot (visited on 11/03/2020).

[The] The Weaver Computer Engineering Research Group. *The GFLOPS/W of the various machines in the VMW Research Group*. URL: http://web.eece.maine.edu/~vweaver/group/green_machines.html (visited on 10/15/2020).

[Tur+16] Turkson, R. F., Yan, F., Ali, M. K. A., and Hu, J. "Artificial neural network applications in the calibration of spark-ignition engines: An overview". In: *Engineering Science and Technology, an International Journal* 19.3 (Sept. 2016), pp. 1346–1359. DOI: 10.1016/j.jestch.2016.03.003. URL: http://dx.doi.org/10.1016/j.jestch.2016.03.003.

[Ube] Uber Technology Inc. *UberPool*. URL: https://www.uber.com/us/en/ride/uberpool/ (visited on 09/17/2020).

[Val+17] Vallon, C., Ercan, Z., Carvalho, A., and Borrelli, F. "A machine learning approach for personalized autonomous lane change initiation and control". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, June 2017. DOI: 10.1109/ivs.2017.7995936. URL: http://dx.doi.org/10.1109/IVS.2017.7995936.

[War19] Warden, P. *Launching TensorFlow Lite for Microcontrollers*. 2019. URL: https://petewarden.com/2019/03/07/launching-tensorflow-lite-for-microcontrollers/ (visited on 06/18/2020).

[Wei] Weigert, J. *xxd(1) - Linux man page*. URL: https://linux.die.net/man/1/xxd (visited on 08/04/2020).

[Wer90] Werbos, P. "Backpropagation through time: what it does and how to do it". In: 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337. URL: http://dx.doi.org/10.1109/5.58337.

[Win+19] Winkler, M., Thieullent, A.-L., Khadikar, A., Tolido, R., Finck, I., Buvat, J., and Shah, H. *Accelerating automotive's AI transformation: How driving AI enterprise-wide can turbo-charge organizational value*. Capgemini Research Institute, 2019. URL: https://www.capgemini.com/research/accelerating-automotives-ai-transformation/.

[Wu+04] Wu, B., Filipi, Z., Assanis, D. N., Kramer, D. M., Ohl, G. L., Prucka, M. J., and DiValentin, E. "Using Artificial Neural Networks for Representing the Air Flow Rate through a 2.4 Liter VVT Engine". In: *SAE Technical Paper Series*. SAE International, Oct. 2004. DOI: 10.4271/2004-01-3054. URL: http://dx.doi.org/10.4271/2004-01-3054.

[XIE+18] XIE, C., Wang, Y., MacIntyre, J., Sheikh, M., and Elkady, M. "Using Sensors Data and Emissions Information to Diagnose Engine's Faults". In: *International Journal of Computational Intelligence Systems* 11 (1 2018), pp. 1142–1152. ISSN: 1875-6883. DOI: https://doi.org/10.2991/ijcis.11.1.86. URL: https://doi.org/10.2991/ijcis.11.1.86.

[Xu+20] Xu, D., Li, T., Li, Y., Su, X., Tarkoma, S., Jiang, T., Crowcroft, J., and Hui, P. *Edge Intelligence: Architectures, Challenges, and Applications*. 2020. arXiv: 2003.12172 [cs.NI].

[Yin+15] Ying, K., Ameri, A., Trivedi, A., Ravindra, D., Patel, D., and Mozumdar, M. "Decision tree-based machine learning algorithm for in-node vehicle classification". In: *2015 IEEE Green Energy and Systems Conference (IGESC)*. IEEE, Nov. 2015. DOI: 10.1109/igesc.2015.7359454. URL: http://dx.doi.org/10.1109/IGESC.2015.7359454.

[Yu+12] Yu, D., Hamad, A., Gomm, J., and Sangha, M. "Dynamic fault detection and isolation for automotive engine air path by independent neural network model". In: *International Journal of Engine Research* 15 (Jan. 2012), pp. 87–100. DOI: 10.1177/1468087412461267.

[Zha+18] Zhang, Y., Sun, P., Yin, Y., Lin, L., and Wang, X. "Human-like Autonomous Vehicle Speed Control by Deep Reinforcement Learning with Double Q-Learning". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, June 2018. DOI: 10.1109/ivs.2018.8500630. URL: http://dx.doi.org/10.1109/IVS.2018.8500630.