Norges miljø- og biovitenskapelige universitet

Master's Thesis 2020 30 ECTS
Faculty of Science and Technology

Utilizing embedded machine learning as a conversion algorithm for converting sensor readings into sensor values on a bare metal embedded device

Balder Grenness Klanderud
Machine, Process and Product development

PREFACE

This thesis concludes my master study in Machine, Process and Product-development at Norwegian University of Life Sciences (NMBU). During my time as a student, I've been fortunate to have been a part of Eik Ideverksted. They have helped me learn beyond the curriculum offered by NMBU, and introduced me to Disruptive Engineering.

I am thankful to Disruptive Engineering, both for hiring me as an embedded systems engineer before the completion of my degree, and for being flexible during the writing of this thesis. I look forward to continue working with them and my colleagues. I would like to thank Ola and Kristian Omberg, along with Odd Ivar Lekang for feedback and guidance with this thesis.

Finally I would like to give a special thanks to my girlfriend, friends and roommates, for all the love and support. They have all been an instrumental part of my student experience, and I am grateful for all the good memories.

ABSTRACT

Even though artificial intelligence (AI) has been frequently used for decades, the number of applications utilizing AI are rapidly increasing. The technology is advancing fast, and its use is spreading to different and new fields. In this thesis, the goal is to further expand the use and value of AI, by utilizing machine learning (ML) as a conversion algorithm for converting raw sensor readings, into the intended measured component (e.g. CO concentrations from a gas sensor). Using machine learning for this purpose is not new, however previous research on this topic have not produced machine learning models designed for use on the edge where the sensor resides. Thus, in this thesis, the main focus is placed on developing a model that can be implemented on a bare metal embedded device.

In order to produce this machine learning model, a dataset containing sensor readings from a metal oxide (MOX) gas sensor, paired with reference gas measurements were used. The Keras framework was used for designing the machine learning model, then by using the X-Cube-AI tool by ST Microelectronics, the model was implemented in the firmware of a STM32F767zi high-performance microcontroller. This allows the microcontroller to use the machine learning model for sensor conversion. The sensor readings were stored in flash, along with the reference gas concentration, which allowed to simulate the sensor by reading values from flash, and compare the values converted by the machine learning model with the reference value from the dataset.

The results from this thesis shows that the developed machine learning model is able to convert the sensor readings into the reference values from the dataset with an average R2 score of 0.975. This score is the highest score achieved on the dataset used in this thesis, and if this score is representable of the expected performance of the MOX gas sensor, this performance would place it among top two gas sensors tested by South Coast Air Quality Management District (AQ-SPEC, 2019). When the model were used by the microcontroller, the conversion process took about 5ms to compute, without a loss in performance. By sacrificing the accuracy of the model, lower compute-time was obtained. The methods used in this paper can easily be adjusted in order to convert sensor readings from other types of sensors, given that a suitable dataset is available.

Keywords: Embedded machine learning; Edge computing; Artificial Intelligence

SAMMENDRAG

Selv om kunstig intelligens (AI) har blitt mye brukt i flere tiår, øker antallet applikasjoner som bruker AI raskt. Teknologien går fort fremover, og bruken sprer seg stadig til forskjellige og nye felt. I denne oppgaven er målet å utvide bruken og verdien av AI ytterligere, ved å benytte maskinlæring (ML) som en konverteringsalgoritme for å konvertere ubehandlet sensoravlesninger, til den tiltenkte målte komponenten (for eksempel CO-konsentrasjoner fra en gassensor). Å bruke maskinlæring for dette formålet er ikke nytt, men tidligere forskning på dette emnet har ikke produsert maskinlærings-modeller designet for bruk ytters i nettverket, der sensoren befinner seg. Dermed er hovedfokuset i denne oppgaven å utvikle en maskinlæringsmodell som kan implementeres på en mikrokontroller.

For å produsere denne maskinlæringsmodellen ble et datasett som inneholder sensoravlesninger fra en metalloksyd (MOX) gassensor sammen med referansemålinger av gasskonsentrasjoner benyttet. Keras rammeverket ble brukt til å designe maskinlæringsmodellen, deretter ved å bruke X-Cube-AI-verktøyet av ST Microelectronics, ble modellen implementert i firmware til en STM32F767zi mikrokontroller. Dette lar mikrokontrolleren bruke maskinlæringsmodellen for sensorkonvertering. Sensormålingene ble lagret i flash minnet til mikrokontrolleren, sammen med referansemålingene av gasskonsentrasjonen. Dette gjorde det mulig å simulere sensoren ved å lese verdier fra flash, og sammenligne verdiene fra maskinlæringsmodellen med referanseverdiene fra datasettet.

Resultatene fra denne oppgaven viser at den utviklede maskinlæringsmodellen er i stand til å konvertere sensoravlesningene fra datasettet med en gjennomsnittlig R2-score på 0,975. Denne poengsummen er den høyeste poengsum oppnådd på dette datasettet som vi vet om, og hvis denne poengsummen er representativ for den forventede ytelsen til MOX-gassensoren, vil denne ytelsen plassere den blant de to beste gasssensorene som er testet av South Coast Air Quality Management District (AQ-SPEC, 2019). Da maskinlæringsmodellen ble benyttet av mikrokontrolleren, tok konverteringsprosessen omtrent 5ms å beregne, uten tap i nøyaktighet. Ved å ofre modellens nøyaktighet ble det oppnådd lavere beregningstid. Metodene som brukes i denne oppgaven kan enkelt justeres for å konvertere sensoravlesninger fra andre typer sensorer, gitt at et passende datasett er tilgjengelig.

Keywords: Embedded machine learning; Edge computing; Artificial Intelligence

TABLE OF CONTENTS

Table of Figures

Index of Tables

ABBREVIATIONS AND GLOSSARY

MOX	-	Metal Oxide Semiconductor
ML	-	Machine Learning
DL	-	Deep Learning
NN	-	Neural Network
MCU	-	Microcontroller, often the computing component of a embedded system
Bare metal device	-	A device where program runs directly on machine hardware, without intermediary an operating system.
Edge	-	The edge is a term used for describing the very end points of a network. A sensor node is a typical edge devise.
Cloud	-	The could is a term used for describing computational resources located on the internet, often in the form of a data-center.
Embedded system	-	A small computational system designed to solve a specific problem. See chapter 2.3 for more information.
MHz	-	Megahertz, used to describe the clock speed of a computer system
GHz	-	Gigahertz, 1 GHz = 1000 MHz
Byte	-	A byte is a word that describes a group of eight bits. Computer memory is grouped in bytes.
Bit	-	The smallest chunk of information/memory, can either be 0 or 1. A bit represents a digit in the binary number system.
KByte	-	Kilo byte. 1KByte = 1000 Bytes
MByte	-	1 MByte = 1000 KByte = 1 000 000 Byte
GC	-	Gas chromatography
HAL	-	Hardware abstraction layer, a set of library functions that simplifies hardware specific operations such as controlling pins.

1 INTRODUCTION

Traditionally, the process of developing a conversion algorithms for converting sensor readings into the desired sensor values (e.g. CO concentrations from a gas sensor), consists of creating a mathematical model describing the sensor and the way it interacts with its environment is developed. This model is then used to find a mathematical relationship between the sensor reading with the desired component. This process requires profound physical and chemical knowledge of the aspects surrounding the sensor.

The metal oxide gas sensor have proven itself to be a low cost gas sensor with capability of detecting multiple gasses depending on the coated oxidized layer. The sensor readings taken from a MOX gas sensor corresponds to the electrical current that flows through a metal oxide element, which can be challenging to convert into gas concentrations (Burgués & Marco, 2018).

Machine learning is a tool that can be utilized for developing a conversion algorithm for converting between sensor readings and gas concentrations. For a machine learning algorithm to be able to compete with a regular conversion algorithm, it must be able to implement at the edge, close to the sensor. Thus this thesis will focus on the development of an machine learning algorithm for converting between the MOX gas sensor readings and gas concentration, and implementing this algorithm on a edge devise.

1.1 EARLIER WORK

There have been written two masters from NMBU which covers some of the same topics as this thesis:

In spring 2019 Julia Førde wrote her master thesis “Development of algorithm for pre-processing and prediction in Capacitive Micromachined Ultrasonic Transducers”. The thesis used a dataset containing sensor values from a MOX gas sensor to automate the development of a machine learning algorithm for prediction of gas concentrations. The MOX gas sensor was used due to the similarities to the CMUT gas sensor, and the lack of an existing dataset for the CMUT gas sensor. The same dataset will be used in this thesis.

In spring 2019 Jon Nordby wrote his master thesis “Environmental Sound Classification on Microcontrollers using Convolutional Neural Networks”. This thesis focused on the use of a microcontroller for embedded machine learning for sound classification. Some of the tools used in his thesis will also be used here.

1.2 PROJECT DETAILS

This thesis will build on the work done by Førde, but the focus is on implementing the machine learning algorithm on a embedded device. By implementing the machine learning on an embedded device, the sensor readings can be calculated on the edge, thus reducing the computation time needed in a data center and potentially save time, power, and infrastructure cost, as well as reducing the overall complexity of the system.

1.3 GOALS AND OBJECTIVES

The purpose of this project is to reduce the complexity and workload needed for creating specialized algorithms, that convert raw sensor readings into the intended sensor value in modern sensor technology.

The main goal in this report is to develop a machine learning model which can be implemented on a bare metal device. The model is to take readings from a MOX gas sensor, and convert them into gas concentrations of different gas compounds. In order to achieve this, the following objectives will be performed.

Literature review

1. Conduct a theoretical study for identifying and defining key concepts which forms the basis for further developed work. These concepts include:
    a) Machine learning
    b) Embedded systems
    c) Embedded machine learning
    d) Cloud, fog and edge computing
2. Get an overview over available tools for aid in development of the following:
    a) Machine learning model
    b) Embedded system firmware

Method

1. Development of machine learning model for converting sensor readings into gas concentrations. This development process consists of the following steps:
    a) Data processing
    b) Model design
    c) Model tuning
2. Implement the machine learning model on an embedded device
3. Present a overview of system functionalities.

Evaluation

1. System description
2. Test plan
3. Benchmarking
    a) Machine learning performance
    b) Embedded systems resource usage

Recommendations and suggestions for further work

Based on work done, suggestions will be put fourth for improving the product.

1.4 LIMITATIONS

Due to the size and time constraints, some limitations have been set:

- The main focus of this thesis will be on implementation on a embedded device
- An exhaustive evaluation of different machine learning models will not be performed, as this was nicely done by Julia Førde in her thesis.
- Only microcontrollers from ARM will be considered

2 THEORY AND KEY CONCEPTS

This section aims to present the relevant theory needed to understand the methods and techniques used in this thesis in order to develop, test and evaluate both the machine learning model and the embedded systems performance.

The literature used in this chapter consists of books and articles found using scientific databases such as Google Scholar and Oria. When possible, peer reviewed papers were used, and newer papers were preferred. Some inspiration have been taken from other similar master's theses. Since embedded machine learning is still new, some litterateur comes from code repositories, and documentation produced by companies such as ST and ARM.

2.1 MOX GAS SENSOR

The MOX gas sensor consists of a semiconductor and metal oxides as selectivity layer, where different oxides have different selectivity against different gas compounds. Depending on the gas mixture in the air, the conductivity of the semiconductor changes. The sensor reading is thus either a measurement of the resistance over or the electrical current through the semiconductor. However, the conductivity is also dependent on environmental factors such as temperature, humidity and pressure (Burgués & Marco, 2018). Since the resistance can vary based on multiple factors, combining multiple MOX sensor elements and a temperature and humidity sensor is common, as well as a heating element to regulate the temperature of the sensor.

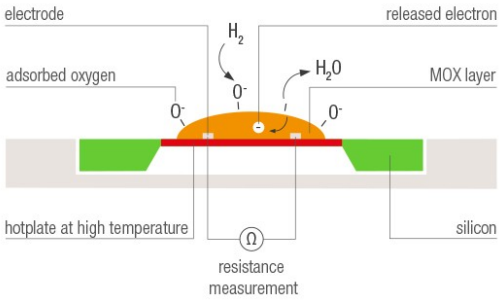

Figure 1: Basic working principle of a MOX gas sensor. Image is gathered from (Sensirion, 2019).

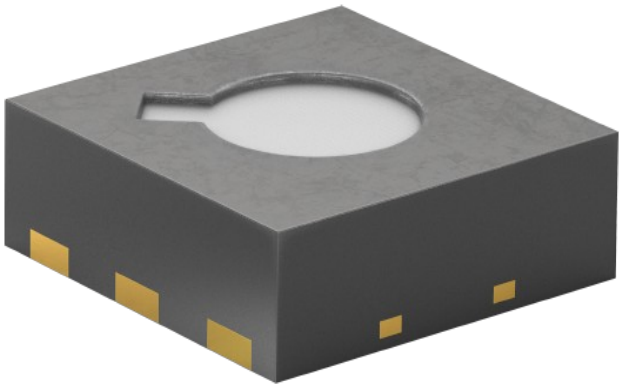

Figure 2: Rendered image of a MOX gas sensor produced by Sensirion. Image is gathered from (Sensirion, 2019).

Figure 1 illustrates the workings of a MOX gas sensor, a hot plate is covered with a metal oxide layer. This metal oxide reacts with the sounding air, and by doing so changes the resistance. It is this resistance that is measured in a MOX sensor.

2.2 MACHINE LEARNING

Artificial intelligence is a subfield of computer science that focuses on mimicking (human) intelligence in a computer program. Machine learning is a broad field within artificial intelligence which have been popular recently (Bainbridge, 2012).

2.2.1 WHAT IS IT?

A machine learning model is a program which learns from data, generally the more data it is given, the better the model performs. A machine learning model is used in two stages, training and predicting. The training stage is where the model is fed data to train on, and the prediction stage is the stage where the model predicts based on the data. Once a machine learning model have yielded sufficient performance, it is said that the model have *fitted* the data, and can be used in the field (Joshi, 2016).

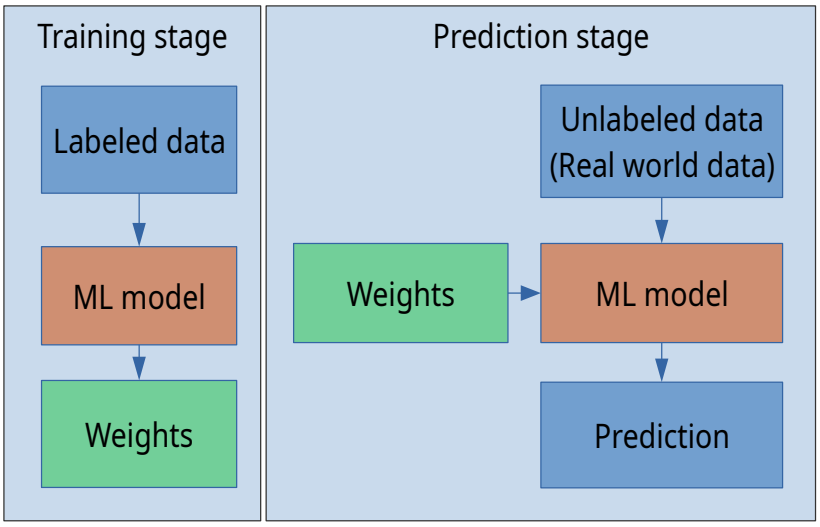

Figure 3: The different stages of a machine learning model. Made by the author.

2.2.2 HOW DOES IT WORK?

When the machine learning algorithm is in the training stage, it uses the labeled data to generate a set of *rules*. These rules are often called *weights*, and are used in the predicting stage.

Many different algorithms within machine learning exists, and more are being developed. Perhaps the simplest example of a machine learning algorithm is linear regression. Linear regression can be expressed by the following equation:

$$y = f(\mathbf{x}) = \mathbf{x} \cdot a + b \qquad \text{(Equation 2.1)}$$

Here **y** is the response, **f** is the predicting model, **a** and **b** are the *rules* or *weights* calculated by the algorithm.

In the training stage, the model is given pairs of ***x*** and ***y*** data points that belong together (labeled dataset). The model calculates its predicted ***y***, often labeled ***ŷ***, this process is called forward propagation. Based on the values calculated the error ***e*** = ***y*** – ***ŷ*** is calculated. Calculus is then used

combined with the error, to calculate how much ***a*** and ***b*** are to be changed by to reduce ***e***. This process is called backward propagation. Properly training a machine learning model is a iterative process, and the training often takes over 100 000 iterations.

An example of use case is to use the square footage of an apartment to predict the sell price of that apartment. Here **x** is the square footage and **y** is the sell price, **a** and **b** are the *weights* created by the machine learning model. This type of model is quite restricting, and can only model simple linear connections, but in many cases yield a good enough result.

2.2.3 GROUPS OF MACHINE LEARNING ALGORITHMS

Machine learning can be grouped into different groups depending on the task at hand. These groups are: supervised-, unsupervised- and reinforcement learning.

Supervised learning are used for classification and regression problems, and they require labeled training data. The labels is often called features and targets, in this thesis features is the raw sensor values, and the targets are the gas concentrations. Generating a high-quality data-set can be costly, and may require a lot of manual labor. Classification is when the model predicts a class (true, false, cat, dog, cancer, not cancer etc.). Regression on the other hand is when a continuous value is to be predicted (price of a house, concentration of a gas etc.) (Gibson, Rogers, & Zhu, 2013).

Unsupervised learning are when non labeled data is given to the model. This type of learning are most used for grouping (e.g. type of movie, book, customer etc.) (Hahn, 2019).

Reinforcement learning are often used for training of robots that are to perform a task (e.g. walk through a maze). This is done by rewarding the model when it does something correct (solves the maze), or punishing the model when it does something wrong (crashes into a wall, is to slow, etc.).

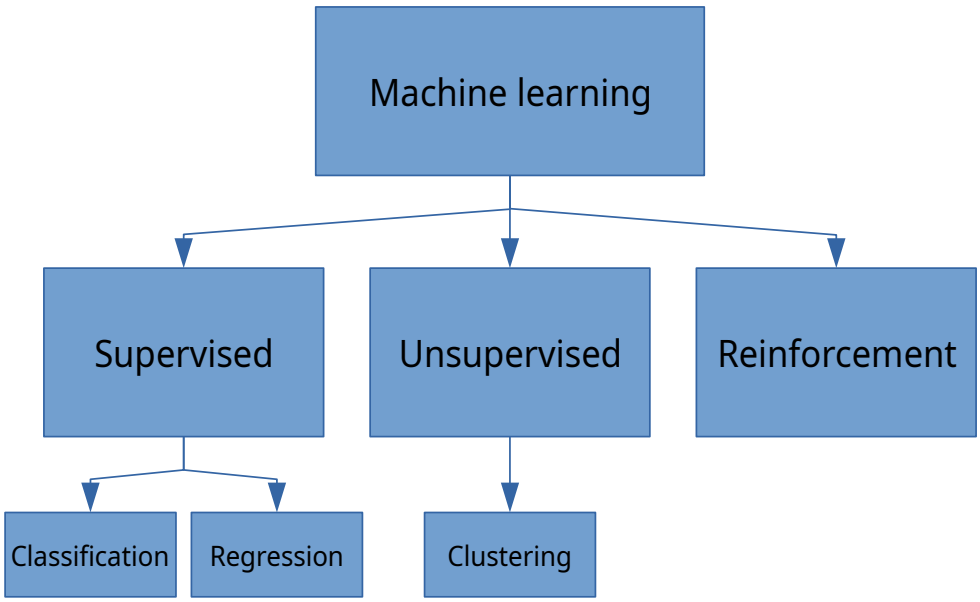

Figure 4: Different types of machine learning. Made by the author.

2.2.4 COST FUNCTIONS

Machine learning models may have different cost functions. A cost function is a function, which based on the output from a machine learning model, calculates a number that represents the performance of the model. In a regression problem, the cost function usually is mean squared error (MSE), or mean absolute error (MAE), but other cost functions are also used, such as mean absolute relative error (MAPE). In a classification problem, the cost functions above does not work, and functions such as binary cross entropy is used (E Silva, Jino, & De Abreu, 2010).

2.2.5 TRAINING AND TESTING

When a machine learning model is evaluated, it predicts the targets from the features of the dataset. Then based on the predicted targets, and actual targets, a **cost** is calculated using the cost function. Initially, when the machine learning model is new, the cost will be high, and the goal of the training process is to minimize the cost. This means that by changing the cost function, the model can be optimized for different performance metrics (e.g. optimize the relative error instead of error).

While training machine learning models, it is common to divide the dataset into a test set and training set. The training set is used for training the model, and the test set is used for testing the performance of the model (E Silva et al., 2010).

2.2.6 PRE-PROCESSING

Before data can be fed into a machine learning algorithm, it needs to be pre-processed. The pre-processing part of a machine learning model can vary in complexity, and when done right, it can improve both accuracy and compute time.

The most common way to pre-process data is to standardize it, this transforms the data such that the mean is zero, and standard deviation is one. This is important since most machine learning algorithms are optimized to work with data in this range, and if the data is not standardized, the model may take more iterations to train, and yield worse results (Joshi, 2016).

Other useful methods are feature extraction and feature augmentation. Feature extraction is the process of selecting some of the features of the data. It is useful if some features contains little or no information, or when there are too many features to efficiently process. Feature augmentation is when new features are created from the old features. There are many different methods for feature augmentation, and one common method is to use principal component analysis (PCA). By using PCA, the number of features can be reduced by orders of magnitude, without loosing significant *information* in the data. Another method is to simply manually create new features (e.g. $x_3 = x_1 / x_2$), however, this often requires knowledge about the data, and can be time consuming (Sun, Lu, Pang, & Sun, 2019).

2.2.7 THE NEURAL NETWORK

A neural network (NN) is a machine learning model that have been the key component in a new subfield of machine learning called deep learning (DL) (Moons, Bankman, & Verhelst, 2019). The design and structure of a neural network was inspired by the way neurons inside the brain work. Each

neuron takes multiple inputs, and produces an output, which can then given to other neurons. This structure is then used to build a network of neurons. Neural networks have shown to produce high performing machine learning models. Another benefit is that neural networks can be altered by changing the number and layout of the neurons, this changes the complexity of the network and allows for high flexibility when using neural networks.

2.3 EMBEDDED SYSTEMS

An embedded system is a small computer system designed to solve a specific problem, and that interacts with its environment, often through an electrical, chemical, or mechanical interface. They can vary a lot in size, price, and functionality, from controlling the windows in a car, to a fully fledge computer (Valvano, 2017).

In this thesis the focus on embedded systems will be limited to ARM processors due to their widespread use in the industry.

Embedded systems often comes in the form of a system on chip (SOC), which are either based on a microcontroller (MCU) (bare metal) or a microprocessor (MPU) (single board computer (SBC)). Both types of embedded systems are widely used, and are useful for different applications.

2.3.1 MICROCONTROLLER (MCU)

Microcontrollers are small chips focused on power-consumption and size, they have integrated CPU, memory and input/output (IO) on the silicon die (ARM, 2019b). Because of this, they often only require power in order to be integrated into a circuit board. This makes them cheap to use, however, much of the internal *silicon real estate* is taken up by components that normally is located on external components (memory, input/output etc.). This mean they have fewer functionalities and worse performance characteristics when compared to microprocessors (lower clock speed, less RAM etc.). The microcontrollers produced by ARM is known as the cortex-m family and are ideal for deeply embedded applications (ARM, 2019b).

There are very limited resources on a microcontroller, typically there is between 128-512kB of RAM (volatile storage) and 512-2048kB of flash (non volatile storage) on a **high-end** microcontroller. For context, a typical low-end modern laptop have a minimum of 4 GB of RAM and 128 GB of storage. This constraint mean that typically, microcontroller applications are written in a low-level programming language such as C, C++ or assembly, then compiled to machine specific instructions.

2.3.2 SINGLE BOARD COMPUTERS

A single board computer (SBC) is a complete computer built on a single circuit board. They usually have a separate microprocessors (MPU), memory, input/output, and other components required to operate. This means that they have more available resources than a microcontrollers, both in term of computational power and available memory. This gives them the performance needed to run an operating system.

Most single board computers are based on a micro processor unit (MPU) designed by ARM. These processors uses one of ARM's instruction sets (ARMv7, ARMv8 etc.) and does not understand the x86 instruction set used by Intel and AMD (x86 is the proprietary instruction set used by Intel and AMD, which combined had nearly 100% computer market share in 2017 (Aram, 2017)). This means that programs running on a SBC either have to be compiled for that specific architecture, or written in a interpreted language such as Python or Java (Indiana University, 2018).

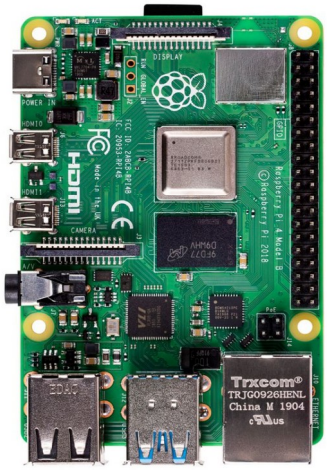

Figure 5: The Raspberry Pi 4 is a popular and modern high performance SBC capable of running a normal operations system. Image is gathered from (Pi Raspberry, 2019).

2.4 CLOUD, FOG AND EDGE COMPUTING

When designing a system that requires computations (e.g. processing sensor data) there are multiple infrastructures that can be utilized. These infrastructures can be grouped into cloud, fog and edge computing. The different infrastructures carries both pros and cons when it comes to cost, complexity, and latency.

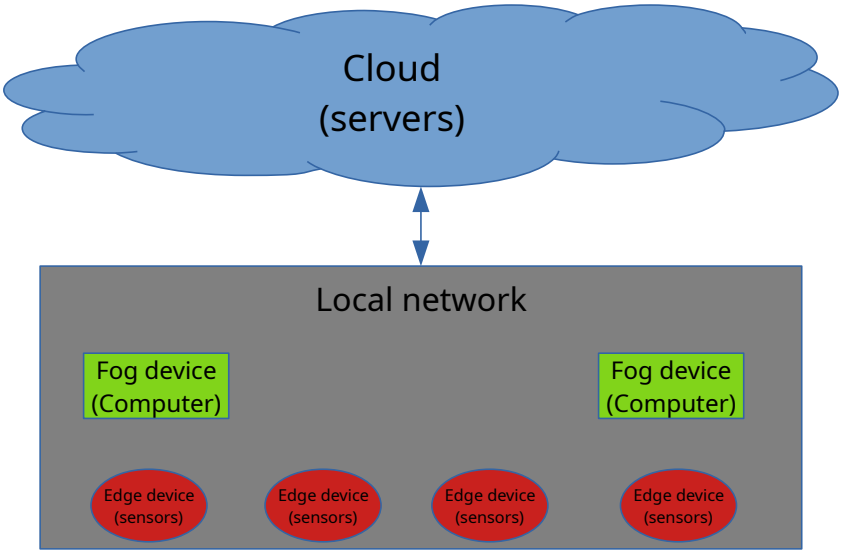

Figure 6: Difference between the Edge, Fog and Cloud. Made by the author.

2.4.1 CLOUD COMPUTING

The cloud is essentially a big computer (server) placed somewhere on the internet. It is commonly used for storage, hosting websites, and performing computations. The benefit to utilizing the cloud is that it is always available (with an internet connection), and have seemingly unlimited of resources. When utilizing the cloud for computation and processing of sensor data, the sensor data needs to be transmitted over the internet. In some cases, this requires transmission of sensitive data and thus security may be of concern. Data bandwidth and latency should also be considered, as some types of sensor devices generate large amount of data, and transmitting them could be expensive and time consuming (Das, Barik, Dubey, & Roy, 2018).

2.4.2 FOG COMPUTING

The fog is a group of devices located within the local network of a sensor device, and usually have the computation power similar to a regular computer or greater. Benefits of using a fog device, is that data does not need to be transmitted over the internet, and both security and bandwidth is thus less of a concern. However, a local device needs to be installed, and for small installations (e.g. one single sensor in one house), this may not be cost effective. However, in some cases it may be ideal, such as places with large networks of sensors (e.g. temperature and humidity monitoring different places in a warehouse) (Das et al., 2018).

2.4.3 EDGE COMPUTING

Edge computing is when the computations is performed locally where the data is produced, instead of transmitting the data for remote computations. In a sensor application, this means to process and analyze the sensor data at the sensing device. This removes the security, bandwidth and latency concerns that may arise with cloud computing. However, edge devices have far fewer computational resources compared to the cloud and fog, and upgrading a edge device is also harder. This makes it not always computational feasible to utilize edge computing (Bhat & Goh, 2017).

2.5 PROGRAMMING TOOLCHAINS

There are many different toolchains to choose from when it comes to both embedded programming, and machine learning. In this subchapter a few options will be presented.

2.5.1 EMBEDDED PROGRAMMING

When programming embedded systems, the companies that produce the microcontroller (MCU) often have their own recommended tools to use for programming. Since programming microcontrollers and single board computers (SBC) are programmed differently, the focus will be on microcontrollers.

Microcontrollers have to be programmed in a compiled language (in most cases), and not all compiled languages are designed for low level programming. C, C++ and Rust are three compiled programming languages frequently used in embedded programming.

C programming for embedded systems

The C programming language is the most used programming language for programming embedded systems. It was originally designed at Bell Labs by Dennis Ritchie as a system level programming language, and originally created for writing the UNIX operating system. The C programming language provides low level memory access, and is designed for having virtually no overhead (Chantelle, 2017).

C++ programming for embedded systems

C++ were developed as a response of the lack of objects/classes in the C programming language, and was originally designed as an extension of C (originally named C plus classes). However, it have been continually developed, and have become its own language since then. It keeps a lot of similarities with C, and most C programs can be compiled with a C++ compiler. C++ can be used for programming embedded systems, although C is more popular, many new projects decide on using C++ instead of C, but this is usually done with a syntax similar to C (Chantelle, 2017).

Rust for programming embedded systems

Rust is a relatively new programming language created to be a modern alternative to C and C++. The main focus of Rust is to guarantee a fast and memory safe language. Although it is possible to use Rust and there are benefits of doing so, it have still not reached a widespread use, and the development tools are in its infancy (Chantelle, 2017).

CMSIS

The CMSIS (Compliant ARM Cortex Microcontroller Software Interface Standard) is a vendor-independent hardware abstraction layer for Arm Cortex processors (ARM, 2019a). It contains different components/libraries that can be included into a project.

STM32CubeMX

ST is a manufacturer of ARM Cortex-M microcontrollers, which also produces accompanying software tools, one of which is STM32CubeMX. This tool lets developers configure a microcontroller in a graphical user interface (GUI). When the desired configurations are made, it produces source code with correct configurations (clock, pin configurations, etc.) and accompanying hardware abstraction layers (HAL). These HALs can be used for communicating with peripheral such as USB, I^2C and SPI etc (ST Microelectronics Inc., 2020). Some code generated by STM32CubeMX uses the CMSIS.

2.5.2 MACHINE LEARNING MODEL DEVELOPMENT

When creating machine learning models and applications, there are multiple programming languages, tools and libraries to choose from. All tools have their pros and cons, and are designed for different applications and use. Python is currently the programming language of choice when it comes to developing of machine learning models. The reason Python is well suited for machine learning applications, is because of the abundance of libraries written for it. When it comes to computation however, Python is slow, so many of the libraries are optimized with faster languages such as C and C++ (Joshi, 2016).

Sci-kit learn

The sci-kit learn library is a large library containing many different machine learning tools, and contain all the tools needed for most types of machine learning problem. It contains tools for splitting and pre-

processing data, as well as machine learning models for both regression, classification and clustering (Joshi, 2016).

TensorFlow

TensorFlow is a toolchain created by Google Brain, optimized for parallel computation. It have support for utilizing a GPU for computation (which Sci-kit learn does not). In recent years, the machine learning library *Keras* have been integrated into TensorFlow. This makes TensorFlow a good choice for creating neural networks. The networks within the *Keras* library is well written and optimized, and thus easy to use when creating neural networks (Joshi, 2016).

2.6 EMBEDDED MACHINE LEARNING TOOLS

There are a number of different tool developed for aiding in the development and deployment of embedded machine learning. In this chapter some of thees tools will be presented.

2.6.1 CMSIS NN

The CMSIS NN component contains a software library for neural network development on ARM microcontrollers. It is designed and highly optimized for high performance and low memory footprint on ARM devices. It utilizes the Single Instruction Multiple Data (SIMD) instruction set, which can yield up to 4x improvement in computational performance (Lai, Suda, & Chandra, 2018).

2.6.2 X-CUBE-AI

X-Cube-AI is a extension to the STM32CubeMX tool, and is used for implementing pre-trained neural networks on a microcontroller. This tool can import pre-trained models from different machine learning toolchains (Keras, Caffe, etc.) and convert them into C code which can be run on a microcontroller. X-Cube-AI utilizes the CMSIS DSP (Digital Signal Processing) component and the SIMD instruction set, but in a compiled binary object, and is thus not open source, in contrary to the other HALs produced by STM32CubeMX (ST Microelectronics Inc., 2019).

3 METHOD

In order to develop and test the embedded machine learning firmware, a microcontroller is needed. The microcontroller used is the STM32F767zi which is a ARM Cortex-M7F based microcontroller with 512kiB of RAM, 2MiB of flash, and a dedicated floating point unit (FPU) (ST Microelectronics Inc., 2017).

The source code for the firmware and machine learning models produced in this thesis is hosted on GitHub, this makes it easy to view the development process, and iterative design of the embedded system firmware. The machine learning models were developed using python and the Keras library.

All the code developed and used can be found here: https://github.com/balderk/embedded_ml , and all the tools used are free to download and install.

3.1 AIR QUALITY DATASET EVALUATION

Development of a machine learning algorithm require a good labeled dataset. In this thesis, the same dataset as used by Julia Førde in her thesis will be used. This dataset contains raw sensor readings from a MOX gas sensor, combined with reference measurements from a gas spectrometer. The MOX gas sensor have 5 sensing elements in addition to a temperature sensor, relative humidity, and absolute humidity. The reference measurements have measurements for 5 compounds: Carbon monoxide (CO), Non Methane Hydro Carbons (NMHC), Benzene (C_6H_6), Nitrogen oxides (NO_x) and Nitrogen dioxide (NO_2) (UCI, 2016). The full and original dataset can be gathered from https://archive.ics.uci.edu/ml/datasets/Air+Quality.

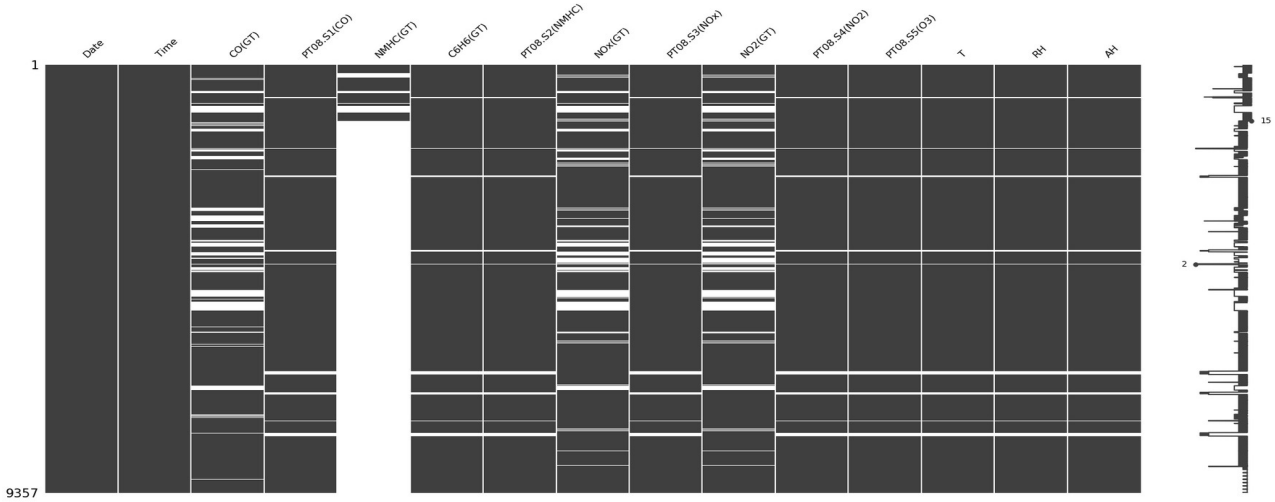

*Figure 7: Graphical view of missing values in the dataset. This visualization was created by using the **missingno** python library and is useful for deciding how to treat missing values. The white lines represent missing data. Made by the author.*

The sensor measurements were taken at hourly intervals between 10. March 2004 and 4. April 2005 with a total of 9357 readings. As can be seen in figure 7, there are many missing values, especially in the Non Methane Hydro Carbons (NMCH) column. After removing the NMHC column and all the rows with missing data, 6941 samples remain.

Something worth noting is that often where there are missing values in one of the reference measurements, the values is also lacking for all the other reference measurements. The same is true for the sensor readings. This makes it easy to justify the removal of these samples, as they contain no useful data for the machine learning model.

3.2 SOFTWARE TOOLS USED FOR DEVELOPMENT

For developing the machine learning model, the programming language Python was used, combined with the library **TensorFlow** and **Keras**. To aid in data management, the **Pandas** and **Numpy** libraries were used, and for visualizations the **Matplotlib** and **Seaborn** library were used.

The embedded code developed in this project was written in the C programming language, and the **STM32CubeMX** tool was used to generate template code for using the peripheral, and the **X-Cube-AI** extension was used for converting the machine learning model to embedded C.

3.3 PERFORMANCE ANALYSIS

To assess the different models, the performance of the system must be measured. The overall performance of the system developed can be divided into machine learning model performance and embedded systems performance. Since there is no single metric for analyzing the performance of these two parts together, they will be analyzed separate.

3.3.1 MACHINE LEARNING MODEL PERFORMANCE ANALYSIS

To analyze the performance of a machine learning model, the predicted values from the model must be compared with the actual values. This can be done by using multiple different metrics, but in this thesis three metrics have been chosen: Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and R2 score.

Mean absolute error tells us the expected deviance in the measurement, and is a much used metric for regression problems. In this case, the MAE tells us the expected error of the prediction in term of ppb NO_x.

There is a major limitation with using MAE, this is because the MAE score of different gasses have different units, and cannot be directly compared. This limitation is removed when using the mean absolute percentage error (MAPE) metric, as this measures how many percent the measurement will be off (e.i. independent of the gas component). When it comes to gas sensors, vendors such as UST also uses the MAPE metric to describe the accuracy of their gas sensor (UST, 2017).

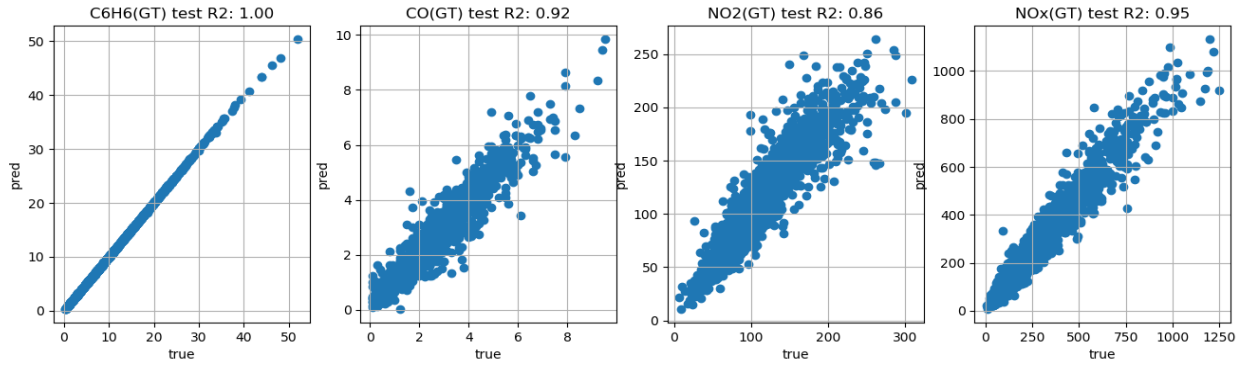

Figure 8: Example of R2 scores with accompanying plots showing the predicted vs true gas concentration. As can be seen on the plots, the model is much better at predicting C_6H_6 than NO_2. Made by the author.

The R2 score gives a value between -1 and 1, and tells the linearity between the predicted value, and actual value. A R2 value of 1 means perfect prediction, and is the measurement used by AQ-SPEC in their independent testing of different gas sensors (AQ-SPEC, 2019), and by Julia Førde in her master thesis.

3.3.2 EMBEDDED SYSTEM PERFORMANCE ANALYSIS

On an embedded device, performance can be measured by two metrics: compute time and memory usage. As they both have different impact on the system.

Compute time is the time needed to perform the conversion computation on the embedded device (lower is better). On a real-time system, this is especially important since they have timing requirements that can be broken if the computation takes to long. Not all embedded systems are real-time systems, and thus may not be of concern. Another consequence of high compute time is the power consumption. If the compute time is low, battery powered devices can go to an idle state between conversions. The more time needed for computation, the longer the microcontroller is in a high energy state, and the more power is drained. However, since the power usage of embedded devices is often low (a power consumption around 0.1W is common), this is only applicable for battery powered devices (Ŝimunić, Benini, & De Micheli, 2001).

Memory usage is important as it greatly affects the price of the microcontroller needed to run the computation, since both flash and ram directly affects the price of the device. Flash is needed for storing the model and weights, and RAM is needed to store intermediate values. Ram usage is usually much lower than flash usage, as there are fewer intermediate values than weights. Both flash and ram usage can be reduced by compression of the weights, but this may affect the performance of the machine learning models.

To measure compute time on an embedded device, multiple techniques can be used. One common method is to use the built in clock to measure the timestamp before and after the calculation, another is to use external pins on the microcontroller. When using external pins, one pin is set to a high voltage state before the calculation, then low after the calculation, this signal can then be monitored

by an external device. In this assignment the latter method will be used, as this yield the highest accuracy as the process of toggling pins is fast, and carries little overhead (Valvano, 2017).

3.4 MACHINE LEARNING MODEL STRUCTURE

The machine learning model structure is divided into tree parts: Preprocessing, Network, and Evaluation. The final goal with the development of the different parts, is that the final model can be implemented on a microcontroller. The X-Cube-AI tool was used for implementing the resulting Keras model on a microcontroller, thus the focus will be on only using Keras modules that is supported by the X-Cube-AI tool. This have the benefit of making the conversion process easy and reproducible.

Figure 9: Model structure overview. Made by the author.

3.4.1 PREPROCESSING STAGE

In the preprocessing stage the data is standardized. This makes the average value of each sensor zero, with a standard deviation of one. The formula for performing this conversion is presented in equation 3.1, where is the average of the data, and σ_X is the standard deviation of the data.

$$X_{std} = \frac{X - \overline{X}}{\sigma_X} \qquad \text{(Equation 3.1)}$$

This usually makes the training process run faster, and makes the models perform better. It is easy to implement this calculation with programming, however, for ease of portability this step is done as a layer in the Keras model.

3.4.2 NEURAL NETWORK STAGE

For prediction on the sensor data a fully connected neural network was used. When adding a fully connected layer to a Keras model, the two most important parameters is the activation function, and the number of neurons. The best activation function varies based on the data. A high number of neurons in each layer gives the ability to explain complex behavior, at the cost of increased compute time, memory usage, and higher probability of overfitting.

3.4.3 EVALUATION STAGE

To evaluate the result from the neural network and get the gas concentrations from the model, a fully connected layer is used. The layer have the same amount of neurons as the number of gas components in the dataset, this way each neuron gives the gas concentration of one component. This method is common to use when working with multi-class problems (i.e. multiple different gas components). This layer also require an activation function, since this is a regression problem, a linear activation function is used. In classification problems it is common to use the sigmoid or softmax activation function.

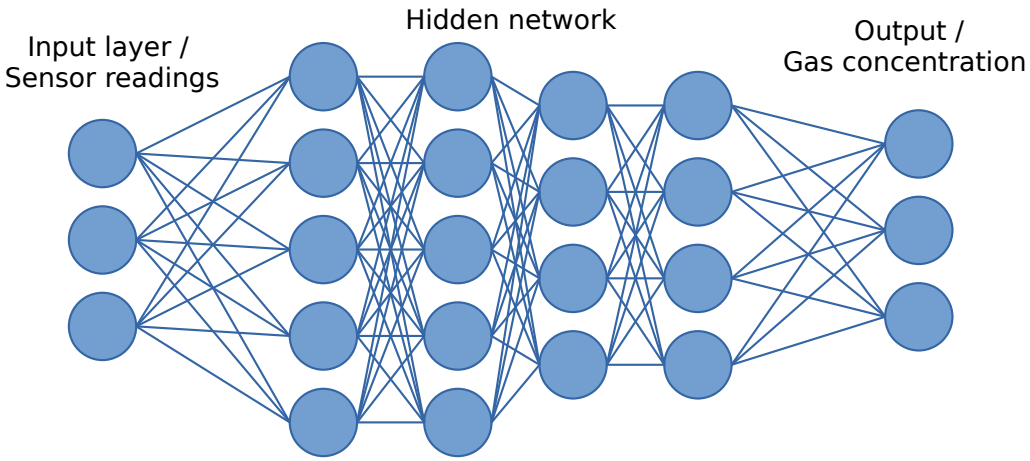

Figure 10: Neural network visualization. Each line represent a calculation, and each dot represents an intermediate stored value. Made by the author.

3.4.4 REGULARIZATION

When training the machine learning model overfitting of the training data may occur, regularization is a tool to prevent this from happening. Various different regularization methods were used in this thesis. One of these methods consist of adding random Gaussian noise to the sensor values, this can simulate inconsistent sensor readings, and work by artificially producing more (lower quality) data samples. L2 regularization for the kernel and activation of the network were also used, which encourages the neurons to use more of the input it is given (use values from multiple sensors).

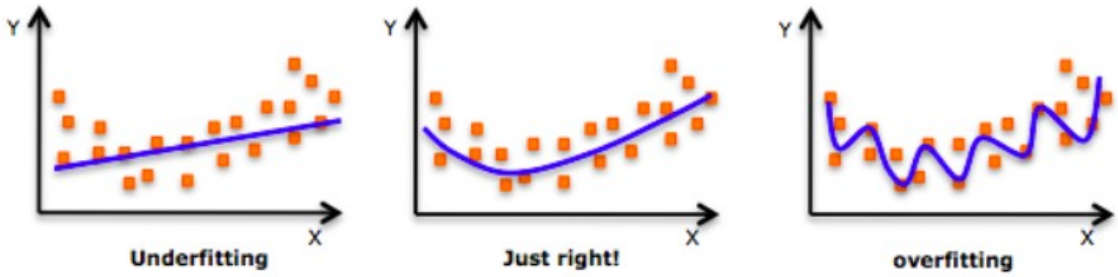

Figure 11: Illustration showing difference between overfitting and underfitting (Shubman, 2018).

3.5 EMBEDDED MACHINE LEARNING MODEL

After the machine learning model was created, it was imported into the X-Cube-AI tool in order to convert it into embedded code. Due to the memory constraints of the embedded device a maximum of 3 models could be implemented on the microcontroller at the same time, however, for real world applications one model would be sufficient. The X-Cube-AI tool informs if the model and modules used

by the model can be converted to embedded code. It also informs if there is a reduction in accuracy by converting the model.

Since the MOX gas sensor used to create the dataset is not available, sensor readings from the dataset have been stored on the flash. This gives the possibility to simulate the gas sensor, while also knowing the actual gas concentrations. In the microcontroller firmware, the microcontroller reads raw sensor readings from flash, then converts them into gas concentration by using the machine learning models. After converting the sensor readings the predicted values are printed, along with the true gas concentration.

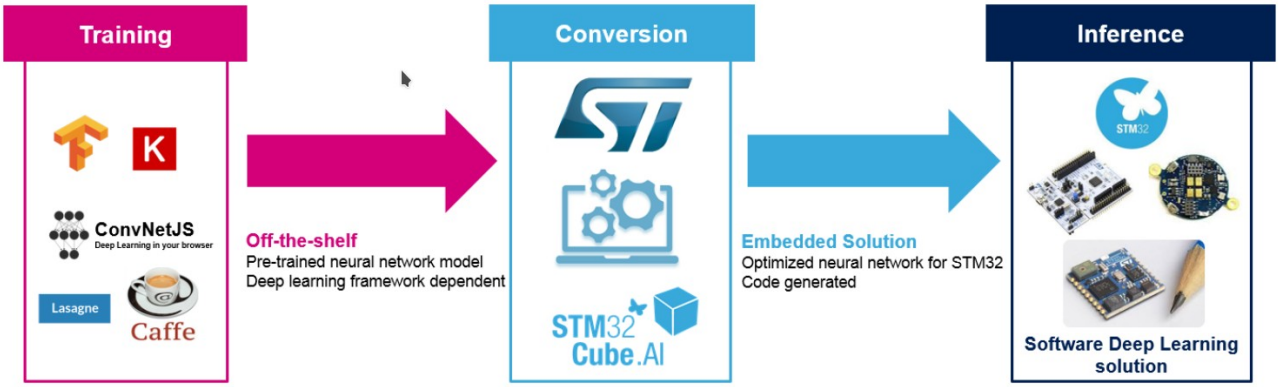

Figure 12: Embedded machine learning workflow (ST Microelectronics Inc., 2019)

Figure 12 illustrates the workflow of creating a embedded machine learning model using the X-Cube-AI tool. The first step is to create a regular machine learning model using one of the possible frameworks. Then the X-Cube-AI tool is used to convert this machine learning model into embedded code. This significantly simplifying the process, and reduces the development time.

4 PRESENTATION OF THE FINAL CODE

After some initial testing and tuning, a set of machine learning models were implemented on the embedded device. In this chapter some of the code will be presented and explained. Due to limitations to this thesis syntax of the programming languages, or how the libraries are to be used will not be explained. However, when necessary, an explanation of what the code does will be provided. The code presented in this chapter is only a small subset of the code produced in this thesis, and the code sections may have been slightly altered to improve the readability of the code. The original code can be found in the GitHub repository here: https://github.com/balderk/embedded_ml.

4.1.1 IMPORTING THE TRAINING DATA

The dataset is provided as an excel file, and must be imported into Python in order to use it to train the machine learning model. To make this process easier, a set of functions were created to perform this function.

```
def get_data() → pd.DataFrame:
    # target_keys = ['CO(GT)', 'NMHC(GT)', 'C6H6(GT)', 'NOx(GT)', 'NO2(GT)']
    df = pd.read_excel(get_filename())

    for key in df.columns:
        df[key].replace(-200, np.NaN, inplace=True)

    return df

def get_feature_targets(dropna=False, feature_keys=None, target_keys=None
                        ) → (pd.DataFrame, pd.DataFrame):
    if feature_keys is None:
        feature_keys = ['PT08.S1(CO)', 'PT08.S2(NMHC)', 'PT08.S3(NOx)', 'PT08.S4(NO2)',
                        'PT08.S5(O3)', 'T', 'RH', 'AH']
    if target_keys is None:
        target_keys = ['CO(GT)', 'NMHC(GT)', 'C6H6(GT)', 'NOx(GT)', 'NO2(GT)']

    data = get_data()[feature_keys + target_keys]
    if dropna:
        data = data.dropna()
        data.reset_index(drop=True, inplace=True)

    features = data[sorted(feature_keys)]
    targets = data[sorted(target_keys)]

    return features, targets
```

The code presented above is the function definitions for importing the targets (gas concentrations) and features (raw sensor readings) of the dataset. The ability to filter out rows containing missing values have been added, as well as the ability to select which columns to import from the dataset.

```
all_target_keys = ['CO(GT)', 'NMHC(GT)', 'C6H6(GT)', 'NOx(GT)', 'NO2(GT)']
drop_target = {'NMHC(GT)'}

f, t = get_feature_targets(
    dropna=True,
    target_keys=list(set(all_target_keys) - drop_target),
    drop_outliers=True
```

```
)

train_f_df, test_f_df, train_t_df, test_t_df = train_test_split(f, t, test_size=0.3,
                                                                random_state=1203)
train_f, test_f, train_t, test_t = train_f_df.values, test_f_df.values,
                                   train_t_df.values, test_t_df.values
```

When importing the data, it is split into features and targets, then it is separated into a train and test set. The train set is used when training the model, and test set is used to test the performance of the model, this is useful when comparing different models. The column containing **NMHC** data was dropped, due to the high number of missing values in this column.

4.1.2 CREATING THE NEURAL NETWORK

In order to quickly train multiple different models for evaluation and comparison, a function for training the models was created. This function took a model definition and a set of parameters, then trained the model and returned a fitted Keras model, which was saved to file. A typical model definition is presented below.

```
input_layer = layers.Input(shape=(f.shape[1],), dtype='float', name='Sensor_data')
first_layer = layers.BatchNormalization(name='Preproprocessing')
last_layer = layers.Dense(t.shape[1], 'linear', name='Output_layer')
model = [
first_layer,
layers.Dense(
    64,
    activation='relu',
    name='Layer_1',
    kernel_regularizer=regularizers.l2(0.01),
    activity_regularizer=regularizers.l2(0.01)
),
layers.Dense(
    32,
    activation='relu',
    name='Layer_2',
    kernel_regularizer=regularizers.l2(0.01),
    activity_regularizer=regularizers.l2(0.01)
),
layers.Dense(
    32,
    activation='relu',
    name='Layer_3',
    kernel_regularizer=regularizers.l2(0.01),
    activity_regularizer=regularizers.l2(0.01)
),
last_layer
]
```

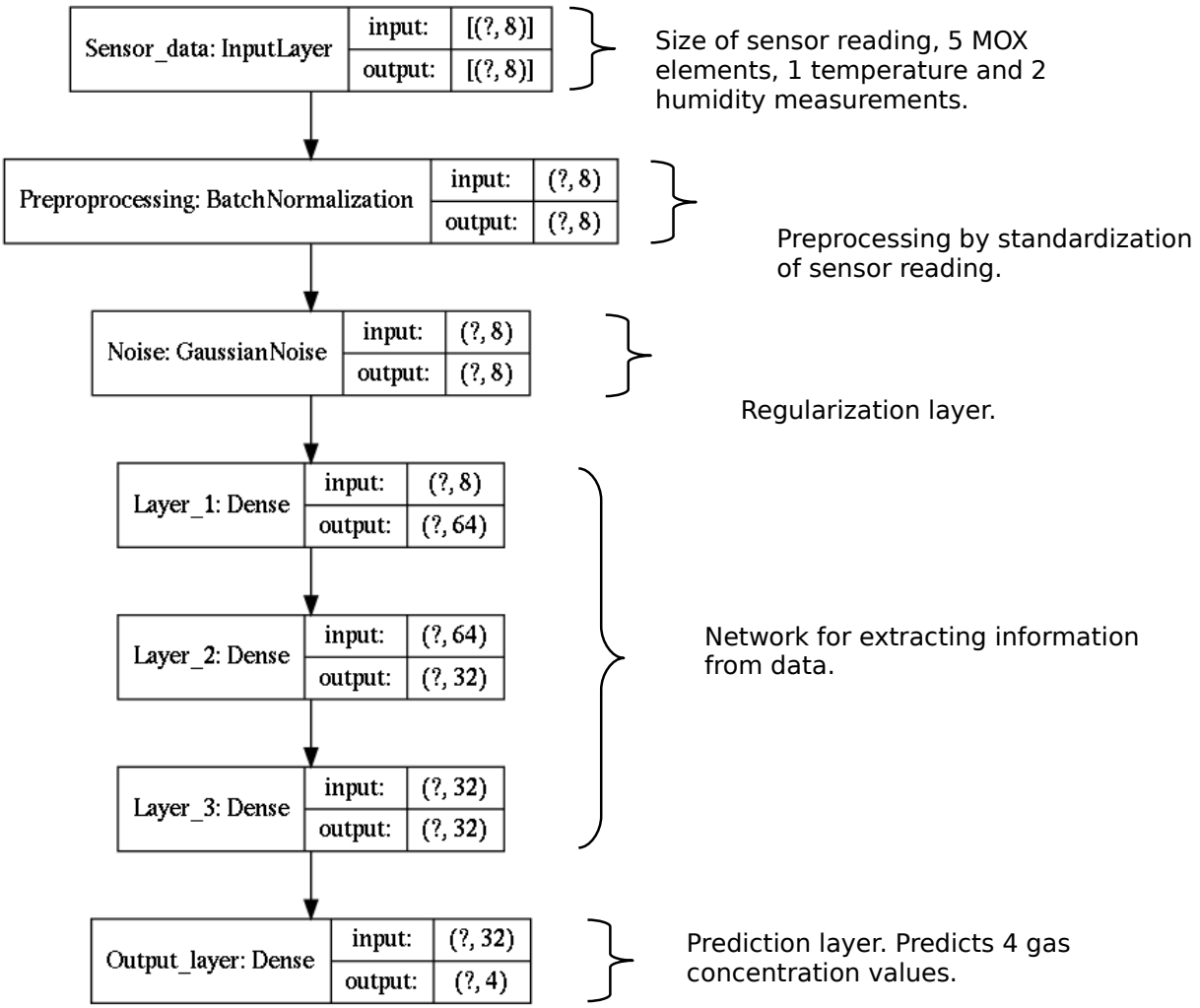

Figure 13: Example machine learning model containing three hidden layers. Made by the author.

Figure 13 shows the layout of one of the machine learning models. A short explanation of what the different parts of the model does is provided. As can be seen by looking at the upper most block, this model expects eight (sensor) values, and from the last block, the model outputs four gas concentration values.

4.1.3 TRAINING THE MODEL

For training the mode, the model definition is used to create a Keras model object as shown in the code section below. In order to use the model, the model definition have to be compiled with a loss function and optimizer. In this example, the mean absolute percentage error (MAPE) was chosen as loss function, and the Adam optimizer was chosen as optimizer.

```
opt = config.get('optimizer', optimizers.Adam(lr=1e-3))

model = Sequential(model_definition)  # creating the Keras model object
model.compile(
```

```
    optimizer=opt,
    loss=mean_absolute_percentage_error,
    metrics=['mean_absolute_error', mean_absolute_percentage_error]
)
```

The training process is the most time-consuming part of creating a machine learning model, this part however, can be accelerated by utilizing a graphics processing unit (GPU) or a tensor processing unit (TPU). Depending on the hardware of the computer and what software version of TensorFlow is used, GPU acceleration can be enabled by default. The model is first trained up to 100 000 iterations, then the learning-rate is reduced before it is trained for another 100 000 iterations. This is to ensure that the model properly converges to a optimal point.

```
hist = model.fit(
    train_f,
    train_t,
    initial_epoch=ini_epoch,
    epochs=epoch,
    batch_size=len(train_t),
    validation_data=(test_f, test_t),
    validation_freq=val_freq,
    verbose=False,
)
```

The model is trained on the train set and tested using the test set, this is to help detect when overfitting is occurring. The **hist** object that is returned can be used to view the progress during training.

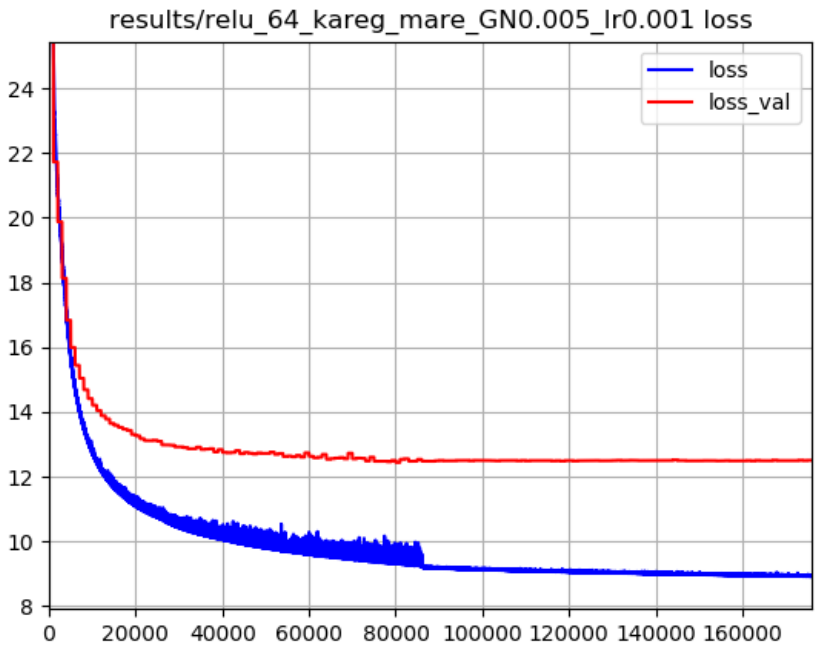

Figure 14: Graph that shows the loss in mean absolute percentage error (MAPE) during the different iterations. As can be seen from around iteration 90 000, which is where the learning-rate is reduced, the loss goes more steadily down. Made by the author.

Figure 14 shows a visualization of how the loss value changes over the number of iterations. This is mainly useful for two things: checking if something is wrong (e.g. the model performance does not improve) and evaluating if the model have converged to a optimum. Here we can see that the loss have stopped decreasing, and the model have converged.

4.1.4 SAVING THE MODEL

In order to use the model at a later stage, it have to be saved as a file. This makes it easy to import the model at a later stage. The Keras model object in Python have a built in function for saving the model as a file, this file contains both the model definition and the weights calculated in the training stage.

```
model.save(f'{name}.h5')
```

4.1.5 CREATING THE EMBEDDED SYSTEM TEMPLATE PROJECT

In order to create the embedded firmware, a template project was first created using the STM32CubeMX program. The firmware is using the UART module to transmit the predicted gas concentrations, as well as GPIO for use in performance measuring and controlling status LEDs. There is also configurations made to configure the clock layout and frequency, instruction and data cache, and interrupt vector table. An example of configuring the UART module is presented in figure 15.

Figure 15: Example of using STM32CubeMX for configuring the UART peripheral on the STM32F767zi. Made by the author.

After the peripheral have been configured, the machine learning model can be imported using the X-Cube-AI add-on tool. When importing machine learning models into this tool, information regarding the memory usage, and model complexity is shown.

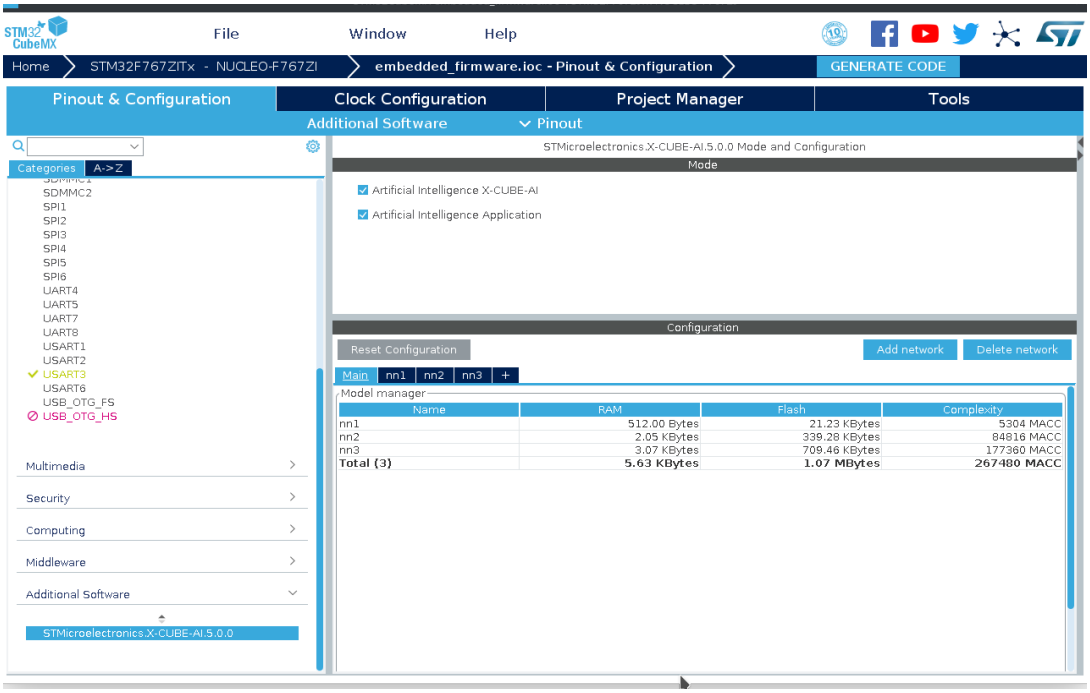

Figure 16: Example of using the X-CUBE-AI tool with three different models. Made by the author.

Figure 16 shows an example of three machine learning models imported into X-Cube-AI. For convenience, the models are named nn1, nn2, and nn3. From the table in figure 16, we know that the models require a total of 1.07MBytes flash, and the total amount of flash available on STM32F767zi is 2MBytes. This means that there is enough flash on the device to store all the models.

4.1.6 DEVELOPING THE EMBEDDED SYSTEM CODE FROM THE TEMPLATE PROJECT

After the template project is generated by STM32CubeMX, project specific code must be added. The embedded code is written in C, using the library functions provided by ST Microelectronics. Since the same calculations and initiations needs to be done on all the different models, there is a lot of repetition in the code. The example code shown in this subchapter have been simplified to remove thees repetitions. A flowchart of the program is shown in figure 17.

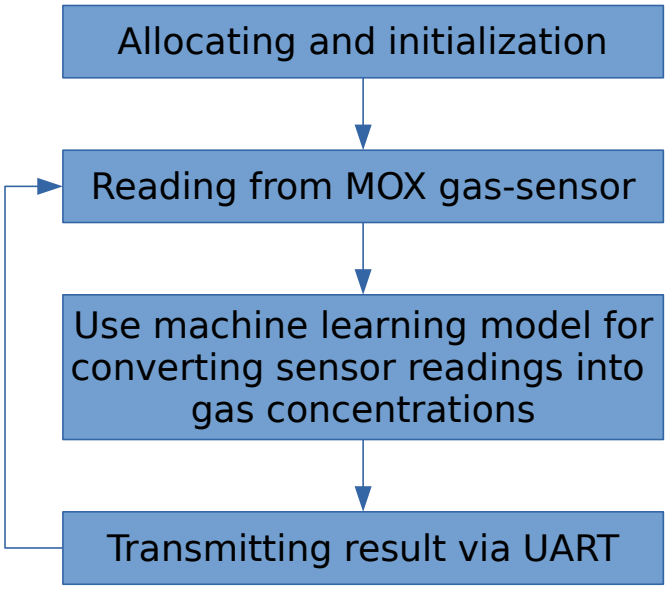

Figure 17: Overview over the embedded program flow. Made by the author.

The first step of the program is to allocate the memory needed and initialize the models. This process takes many lines of code, and is the same for all models, thus only code for allocating and initializing one network is shown. The original code is located in the **main.c** file in the GitHub repository.

```
/// allocate the data types
/// NN1
AI_ALIGNED(4)
static ai_u8 nn1_activations[AI_NN1_DATA_ACTIVATIONS_SIZE];
static ai_handle nn1_handle = AI_HANDLE_NULL;
const ai_network_params nn1_params = {
 AI_NN1_DATA_WEIGHTS(ai_nn1_data_weights_get()),
 AI_NN1_DATA_ACTIVATIONS(nn1_activations)};

/// Create the networks
err = ai_nn1_create(&nn1_handle, AI_NN1_DATA_CONFIG);
handle_ai_err(err);

/// Initialize the networks
if (!ai_nn1_init(nn1_handle, &nn1_params)) {
 err = ai_nn1_get_error(nn1_handle);
 ai_nn1_destroy(nn1_handle);
 nn1_handle = AI_HANDLE_NULL;
 handle_ai_err(err);
}

/// Preparing the data in an out
AI_ALIGNED(4)
float target_data[AI_NN1_OUT_1_SIZE];

AI_ALIGNED(4)
static ai_i8 in_data[AI_NN1_IN_1_SIZE_BYTES];

AI_ALIGNED(4)
static ai_i8 out_data_nn1[AI_NN1_OUT_1_SIZE_BYTES];
```

```
static float *output_data_nn1 = (float *) out_data_nn1;

static float *input_data = (float *) in_data;

ai_buffer inputs[] = AI_NN1_IN;
ai_buffer outputs_nn1[] = AI_NN1_OUT;

ai_buffer *input = &inputs[0];
ai_buffer *output_nn1 = &outputs_nn1[0];

input→data = input_data;
output_nn1→data = output_data_nn1;
```

After the machine learning models and accompanying data-structures have been allocated and initialized, they are ready to be used. In this example, different pointers have been used to make the input and output values easier to access. By using pointers this way, it is easy to analyze the values given to the machine learning model.

```
new_sensor_reading(type); // Get new reading
get_sensor_reading(input_data); // Copy the raw sensor values
get_sensor_values(target_data); // Copy the gas concentrations

n_batch = ai_nn1_run(nn1_handle, input, output_nn1);
```

The **new_sensor_reading** function reads sensor values from the test or train set, depending on the argument given. The functions **get_sensor_reading** and **get_sensor_values** then copies the sensor values into memory. The **ai_nn1_run** performs the conversion process, and the predicted gas concentrations are stored in the **output_nn1** variable. Because of how the data structures was initialized earlier, the gas concentrations are located and correctly encoded in the **output_data_nn1** variable, this makes it easy to access the predictions.

When transmitting the data, it is first converted into an ASCII (American Standard Code or Information Interchange) encoded string before it is transmitted through the UART interface. This encoding is the standard encoding for the characters commonly used in the American language. This makes it easy to read the result on a separate computer.

```
custom_print("\n%14s:\t", "output NN1");
for (int i = 0; i < AI_NN1_OUT_1_SIZE; i++) {
    custom_print("%6d.%.2d ", (int) output_data_nn1[i],
                 abs(((int) (output_data_nn1[i] * 100)) -
                 ((int) output_data_nn1[i]) * 100));
}
```

The **custom_print** function works as the regular **printf** function commonly used in C, but the formatting option for floating point numbers is not implemented, and the data is transmitted via the **UART** interface. By using this function, the process of printing formatted text is simplified, and removes a lot of repetitive code. The function definition is presented below.

```
void custom_print(const char *format, ...) {
    va_list arg;
    va_start(arg, format);
    int len = vsnprintf((char *) _str_buff, sizeof(_str_buff), format, arg);
    va_end(arg);
    if (len > 0 && len < sizeof(_str_buff)) {
```

```
        HAL_UART_Transmit(&huart3, _str_buff, len, 100);
    }
}
```

The transmitted data can be read by using a serial reading program, such as PUTTY or Arduino IDE, the output then looks as the shown in figure 18.

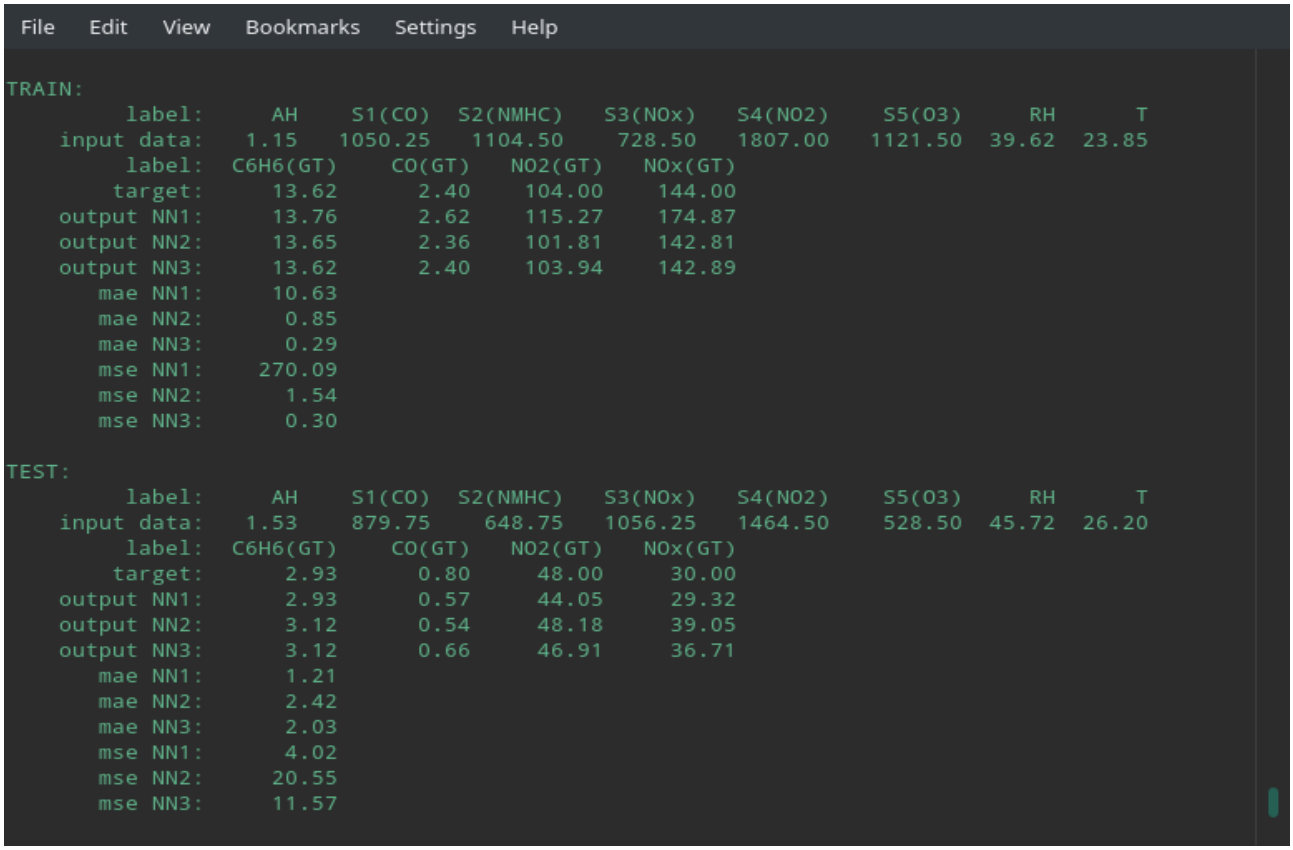

Figure 18: Serial output from the embedded systems showing evaluation from both the test and train dataset with true and predicted gas concentrations. Made by the author.

As can be seen from figure 18, the text is formatted in a human readable format, both in term of text encoding and text placement.

4.1.7 MEASURING COMPUTE TIME ON THE EMBEDDED SYSTEM

As mentioned earlier, in order to measure the compute time of the different machine learning models, the microcontroller would set a pin value high before performing a computation, and low after the computation. In order to make this process easier to perform, some macro functions were created that modified the desired pin. The functions were written as a macro, this is to make the overhead of changing the pin value as small as possible. By using the macro instead of a standard function, the function call is computed at compile time instead of runtime. This reduces the overhead, while also having a function that is easy to use when programming.

```
#define SET_DEBUG_PIN(x)                                                \
  ({ HAL_GPIO_WritePin(DEBUG_##x##_GPIO_Port, DEBUG_##x##_Pin, GPIO_PIN_SET); })
```

```
#define RESET_DEBUG_PIN(x)                                                  \
  ({ HAL_GPIO_WritePin(DEBUG_##x##_GPIO_Port, DEBUG_##x##_Pin, GPIO_PIN_RESET); })
```

Below is an example of using this macro for setting pin values high and low before and after performing conversion using one of the machine learning models.

```
SET_DEBUG_PIN(1);
n_batch = ai_nn1_run(nn1_handle, input, output_nn1);
RESET_DEBUG_PIN(1);
```

A logic analyzer can then be used for measuring the voltage of the pins on the microcontroller. On slow systems, a logic analyzer can be replaced with an LED, and the compute time can be visually evaluated. However, to achieve high accuracy measurements a logic analyzer is used. Output from the usage of a logic analyzer is presented later in this paper.

5 EVALUATION AND RESULTS

In this chapter, the results from both the machine learning model and the embedded system will be presented. Since the results from the machine learning model and embedded systems are not comparable, they will be presented and discussed separate.

5.1 COMPARING THE MACHINE LEARNING MODELS

For evaluating the machine learning models, the parameters mentioned earlier will be calculated on the test part of the data-set. This gives key metrics that represent the expected real life performance of the machine learning model. A total of 173 different models with different parameters were created, and to ease the process of comparing these models against each other, a simple naming system were created.

To explain this naming system, an example name will be used: the model named "relu_256_kareg_mare_GN0.05_lr0.001" uses the **relu** activation function. The first hidden layer have **256** nodes, a Gaussian noise with standard deviation of **0.05** is used. **MARE** tells that the optimizer tried to reduce the mean absolute relative error, and **lr0.001** says that the learning rate of the optimizer is set to 0.001. The name **kareg** tells that both the kernel and the layer activity are regularized, **areg** would mean that only the layer activity is regularized, and **kreg** would mean that only the kernel is regularized. In order to get all the details about the model, the .h5 model file must be imported and analyzed using the Keras library. The model files are located in the "model/results" directory in the GitHub repository.

After evaluating a model, we are left with one score for each gas and each metric (which gives a total of 12 values for each model). In order to compare the models and pick out the overall best model, the average score across the different gas component is used, this gives us the average performance for each metric. Table 1 contains subset of the different models sorted by the average R2 score on the test set.

*Table 1: Table containing the average performance across the different gas concentrations sorted by the **R2 score**.*

idx	name	mape mean	mae mean	R2 mean
140	relu_512_areg_mare_GN0.1_lr0.001	4.153	5.100	0.975
132	relu_512_areg_mare_GN0.05_lr0.001	4.488	5.500	0.972
124	relu_512_areg_mare_GN0.01_lr0.001	4.513	5.484	0.971
137	relu_512_kareg_mare_GN0.1_lr0.001	4.562	4.944	0.970
129	relu_512_kareg_mare_GN0.05_lr0.001	4.937	5.080	0.967
116	relu_512_areg_mare_GN0.005_lr0.001	4.998	5.501	0.964
138	relu_512_mare_GN0.1_lr0.001	5.229	5.348	0.964
109	relu_256_areg_mare_GN0.1_lr0.001	5.697	7.416	0.963
113	relu_512_kareg_mare_GN0.005_lr0.001	5.774	5.350	0.963
121	relu_512_kareg_mare_GN0.01_lr0.001	5.033	5.083	0.963

131	relu_512_kreg_mare_GN0.05_lr0.001	5.349	5.383	0.962
103	relu_256_areg_mare_GN0.05_lr0.001	5.792	7.672	0.961
107	relu_256_kareg_mare_GN0.1_lr0.001	5.791	7.060	0.961
139	relu_512_kreg_mare_GN0.1_lr0.001	5.498	5.254	0.961
130	relu_512_mare_GN0.05_lr0.001	5.436	5.184	0.960
55	relu_256_areg_GN0.5_lr0.001	8.840	7.660	0.960
136	relu_512_areg_mare_GN0.05_lr0.0001	5.795	5.271	0.959
63	relu_256_areg_GN0.5_lr0.0005	9.308	7.404	0.958
39	relu_256_areg_GN0.05_lr0.0005	9.182	8.194	0.956
31	relu_256_areg_GN0.05_lr0.001	8.985	8.387	0.954

Table 1 shows that the performance is really good based on the R2 score (the theoretical highest R2 score is 1). A more detailed view of the performance of the best model is presented later in this chapter. Table 2 is a summary of the best models based on the **mape** score.

*Table 2: Table containing the average performance across the different gas concentrations sorted by the **mape mean** metric.*

idx	name	mape mean	mae mean	R2 mean
140	relu_512_areg_mare_GN0.1_lr0.001	4.153	5.100	0.975
132	relu_512_areg_mare_GN0.05_lr0.001	4.488	5.500	0.972
124	relu_512_areg_mare_GN0.01_lr0.001	4.513	5.484	0.971
137	relu_512_kareg_mare_GN0.1_lr0.001	4.562	4.944	0.970
129	relu_512_kareg_mare_GN0.05_lr0.001	4.937	5.080	0.967
116	relu_512_areg_mare_GN0.005_lr0.001	4.998	5.501	0.964
121	relu_512_kareg_mare_GN0.01_lr0.001	5.033	5.083	0.963
138	relu_512_mare_GN0.1_lr0.001	5.229	5.348	0.964
131	relu_512_kreg_mare_GN0.05_lr0.001	5.349	5.383	0.962
130	relu_512_mare_GN0.05_lr0.001	5.436	5.184	0.960
139	relu_512_kreg_mare_GN0.1_lr0.001	5.498	5.254	0.961
109	relu_256_areg_mare_GN0.1_lr0.001	5.697	7.416	0.963
113	relu_512_kareg_mare_GN0.005_lr0.001	5.774	5.350	0.963
107	relu_256_kareg_mare_GN0.1_lr0.001	5.791	7.060	0.961
103	relu_256_areg_mare_GN0.05_lr0.001	5.792	7.672	0.961
136	relu_512_areg_mare_GN0.05_lr0.0001	5.795	5.271	0.959
114	relu_512_mare_GN0.005_lr0.001	5.822	5.355	0.954
122	relu_512_mare_GN0.01_lr0.001	6.061	5.452	0.953
115	relu_512_kreg_mare_GN0.005_lr0.001	6.088	4.953	0.951
128	relu_512_areg_mare_GN0.01_lr0.0001	6.243	5.488	0.954

As can be seen from table 2, the best model in this table is the same as in table 1. The large networks performs overall better than the smaller ones, the network parameters such as learning rate and Gaussian noise also affects performance. This implies that better performance can be achieved, both by further tuning of the network and by increasing the network size.

5.2 SELECTING MODELS FOR FURTHER EVALUATION

Evaluating the models on a embedded system is a manual and slow process. Therefore only three models were selected for further evaluation and testing. The chosen models were: the model with the overall best performance, the best model with **256** nodes in the first layer, and the best model with **64** nodes in the first layer. The results of these models are presented in chapter 5.2.1. The images were created using the **Seaborn** and **Matplotlib** libraries in Python. The code for creating these images is located in the **assess_model.py** file in the GitHub repository.

5.2.1 MODEL PERFORMANCE

For a more detailed evaluation and visualization of the models performance, two plots were created. One to show the distribution of the PAE score, in the form of a box plot, and one to show the R2 score from the different gas components.

The box-plot is a convenient way of showing how big the percentage error usually is. The black line within the colored box represents the average error, the box represents the 25% to 75% range, and the whiskers represent the 5% to 95% range. For a better understanding of the distribution of the predictions, a distribution plot can be created.

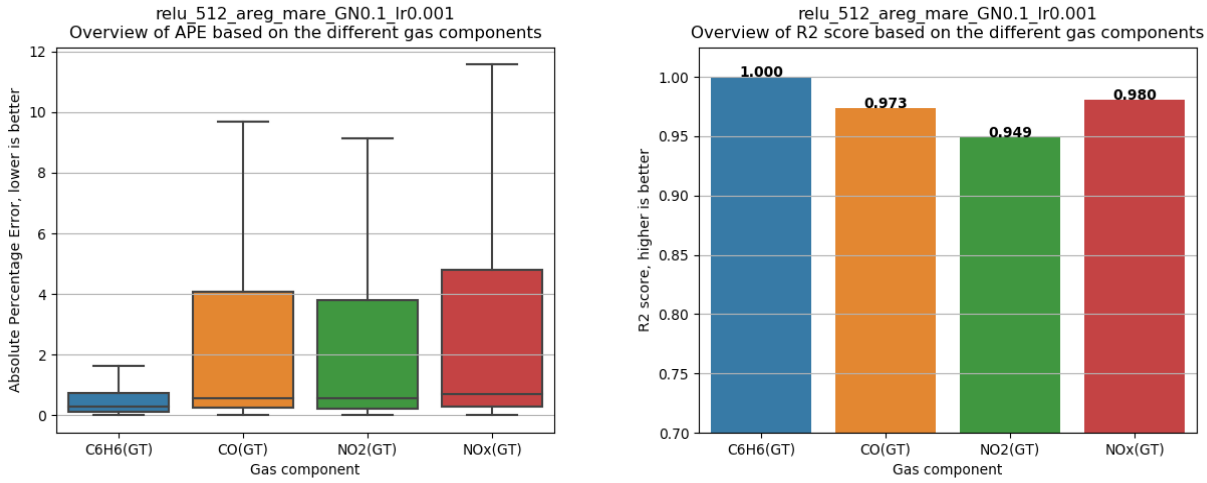

Figure 19: The overall highest performing model. Made by the author.

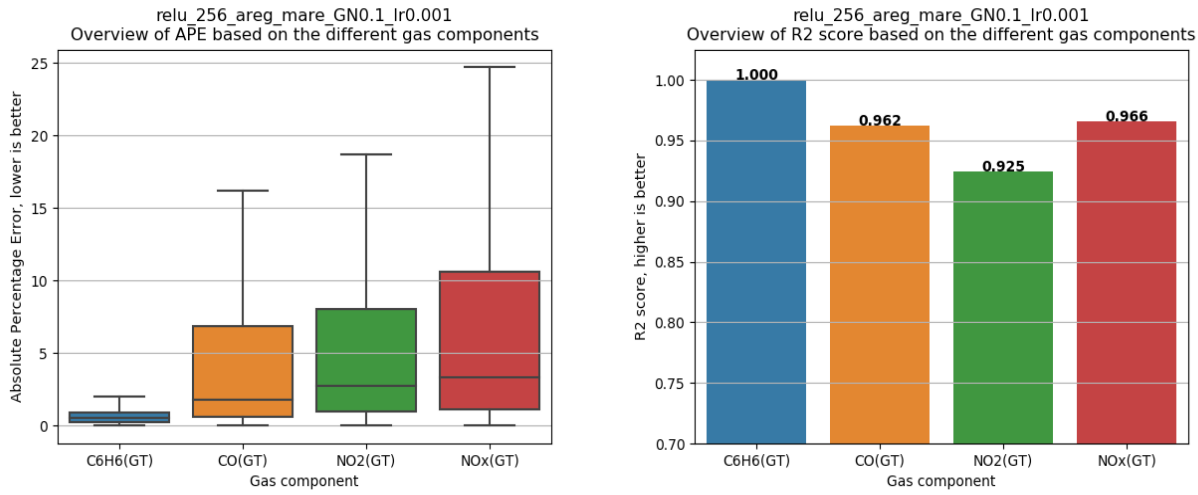

Figure 20: The best performing model with 256 neurons in the first layer. Made by the author.

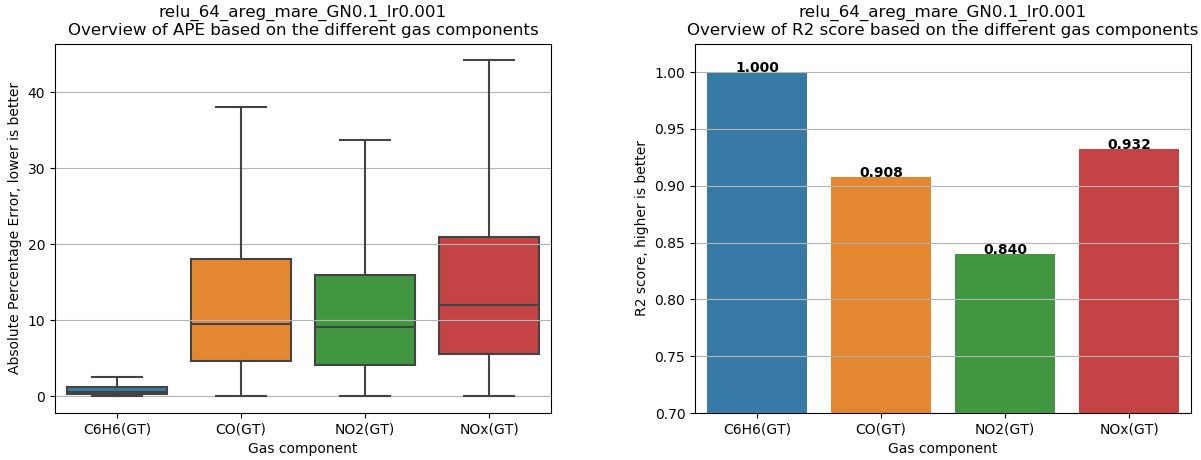

Figure 21: The best performing model with 64 neurons in the first layer. Made by the author.

What stands out in figures 19 – 21 is that C_6H_6 is easy to predict, giving a R2 score of 1.000 on all the models, but as can be seen on the APE plot, it have some variation. However, the other gas components are more difficult to predict. Something else that stands out, is that in all the models above, the same components have the same order in term of which gas components that are easy to predict (when looking at the R2 score).

When comparing the APE plot with the R2 plot, it stands out that the two metrics does not agree on the order of which component is easiest to predict. This is especially visible when looking at the results for the NO_x concentration. On the APE plot the NO_x is the hardest to predict, but on the R2 score plot it is the 2nd easiest to predict. This means that it have a high linearity between the predicted value and actual value, although the error may be significant.

5.2.2 SUMMARY OF MODEL PERFORMANCE

Table 3 contains detailed information about the performance of the best model obtained in this thesis. This is necessary in order to properly compare the performance with the performances achieved by other people on the same dataset. The table is created with the **assess_model.py** file in the GitHub repository, the file is located in the **summary** folder.

Table 3: Detailed summary of the model performance. This is useful when comparing with other research.

relu_512_areg_mare_GN0.1_lr0.001	R2 score	MAPE score
C6H6(GT)	0.9997	0.9549
CO(GT)	0.9732	5.8761
NO2(GT)	0.9486	4.3219
NOx(GT)	0.9804	5.4595

5.3 RESULTS FROM OTHER WORK

Earlier work for creating a machine learning model for converting the sensor readings into gas concentrations exists. In this subchapter, some of these results will be presented. Not all of the research have used the same metrics to measure the performance of their models, and some only used a subset of the gas components available in the reference gas measurements. The results from these studies are presented below.

Development of algorithm for preprocessing and prediction in Capacitive Micromachined Ultrasonic Transducers by J. Førde

This thesis developed a similar machine learning model, but only two of the gasses were used: NO_2 and CO. The only comparable metric between this thesis and her thesis is the R2 score. A copy of her results is provided in table 4.

Table 4: Copy of the results achieved by Førde in her master thesis. The original table is located on page 65 in her thesis (Førde, 2019).

R2 score NO_2	R2 score CO	Average R2 score
0.80	0.43	0.62

CO, NO2 and NOx urban pollution monitoring with on-field calibrated electronic nose by automatic bayesian regularization by De Vito, Piga, Martinotto and Di Francia

This paper created a machine learning model much like what was developed in this thesis. The main difference is that this paper was written 11 years ago. This means that both the computational capabilities of the hardware, and the machine learning tools were not as developed and refined than what was available in the writing of this thesis. In their paper, the authors calculated the Mean Absolute Relative Error (MARE), which can be converted to MAPE by multiplying with 100.

Table 5: Summary of the highest performance achieved in the paper. (De Vito, Piga, Martinotto, & Di Francia, 2009).

MAPE C_6H_6	MAPE CO	MAPE NO_2	MAPE NO_x
2.0	24.6	22.0	42.0

Summary table by AQ-SPEC

The South Coast Air Quality Management District (AQ-SPEC) have tested the performance of different gas sensors, and created a summary table containing this information (AQ-SPEC, 2019). Although this is not directly comparable with the model performance achieved on the test set, it gives an indication of the expected performance and value of a final product containing the MOX sensor from the dataset, and the machine learning model from this thesis. Due to the size of the table, it is attached in appendix A.

5.4 EVALUATING EMBEDDED SYSTEMS PERFORMANCE

In order to assess the viability of using a machine learning model on an embedded system for real time conversion of raw sensor readings into gas concentrations, the model must be evaluated on the device. The three models selected in chapter 5.2.1 were imported into the X-Cube-AI tool and installed on the microcontroller.

5.4.1 SYSTEM REQUIREMENTS

Table 6: Model information with microcontroller requirements. Table is gathered from the X-Cube-AI tool with minor modifications. Multiply-Accumulate operation (MACC) is a metric used for describing and comparing model complexity, and is an implication of compute time.

Name on MCU	Model name	Required RAM	Required Flash	Complexity
nn1	relu_64_areg_mare_GN0.1_lr0.001	384 Bytes	15.44 KBytes	3856 MACC
nn2	relu_256_areg_mare_GN0.1_lr0.001	1.54 KBytes	174.93 KBytes	43782 MACC
nn3	relu_512_areg_mare_GN0.1_lr0.001	3.07 KBytes	709.46 KBytes	177360 MACC

Table 6 shows the microcontroller requirements and the complexity of the different models. As can be seen from the RAM, flash, and complexity column, the larger models have higher requirements than the smaller models. The most restrictive number from the table is the flash requirements, as many microcontrollers have less than 64 KBytes of flash. However, there are methods that can be used both for reducing the required flash, and increase the available flash.

5.4.2 ON DEVICE TESTING

For testing the machine learning model on the microcontroller, the microcontroller transmits the data over serial. The resulting serial output can be seen from the section below, taken from a random sample. As can be seen from this example, the larger models performed better than the smaller models. This can be seen from the MAE and MSE score of the different models (lower is better).

```
TRAIN:
 label:          AH    S1(CO)  S2(NMHC)   S3(NOx)   S4(NO2)    S5(O3)     RH      T
 input data:   1.60    953.25    734.25    861.00   1400.50    825.25  81.02  17.44
 label:       C6H6(GT)    CO(GT)   NO2(GT)   Nox(GT)
 target:          4.40      0.80     61.00     94.00
 output NN1:      4.50      0.81     60.93    125.73
 output NN2:      4.43      0.80     62.12     93.97
 output NN3:      4.45      0.80     60.29     93.13
 mae NN1:      7.97
 mae NN2:      0.29
 mae NN3:      0.40
 mse NN1:    251.78
 mse NN2:      0.31
 mse NN3:      0.31

TEST:
 label:          AH    S1(CO)  S2(NMHC)   S3(NOx)   S4(NO2)    S5(O3)     RH      T
 input data:   0.84   1042.25    813.75    746.25   1167.25   1035.75  75.92   8.42
 label:       C6H6(GT)    CO(GT)   NO2(GT)   Nox(GT)
 target:          6.03      1.70    104.80    286.70
 output NN1:      6.04      1.31     89.88    232.79
 output NN2:      6.12      1.35    103.01    275.48
 output NN3:      6.05      1.69    104.58    284.02
 mae NN1:     17.30
 mae NN2:      3.35
 mae NN3:      0.73
 mse NN1:    782.03
 mse NN2:     32.27
 mse NN3:      1.80
```

For measuring the compute time, a logic analyzer is attached to the debug pins on the microcontroller. A output from this logic analyzer is shown in figure 22.

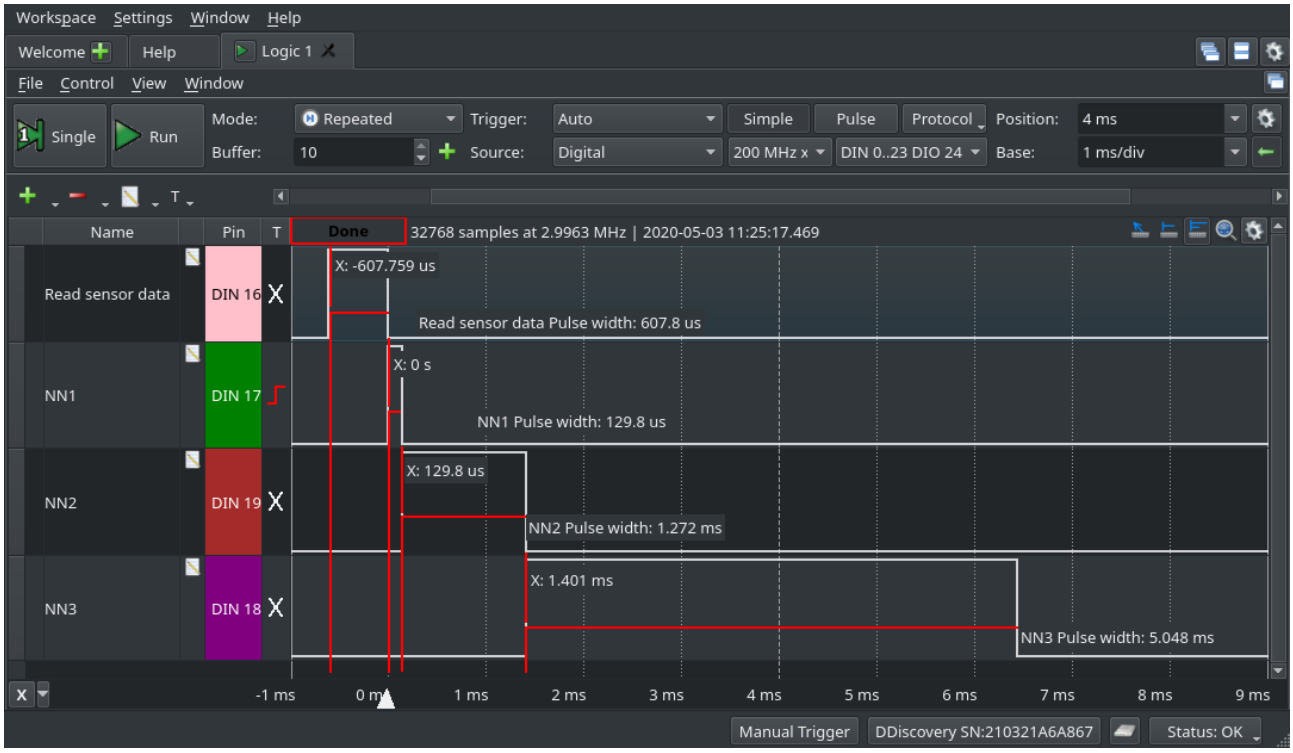

Figure 22: Compute time overview. Image is captured from the waveforms logic analyzer software, with small modifications for displaying pulse width time. Made by the author.

As can be seen from figure 22, the size and complexity of the model directly affects the compute time. The largest model took about 5 ms to compute, and the smallest network only took about 0.13 ms. Keep in mind that this is on a high end microcontroller running at its highest rated clock frequency, with data cache, instruction cache, and FPU enabled, which all reduces the compute time. A budget or low power microcontroller usually runs at a lower clock frequency, and does not have access to either a FPU or cache, which means that the compute time will increase.

There are techniques that can be used for reducing the compute time that are not used in this thesis, such as pruning the network or changing from floating point weights to fixed point weights. These alterations to the machine learning model may negatively affect the performance of the model, as well as require more development time to implement and test.

6 DISCUSSION

During this thesis, a machine learning model for converting raw sensor readings into sensor values have been created and implemented on a microcontroller. This was done by using a dataset containing sensor readings from a MOX gas sensor, and reference gas concentration measurements. Due to the lack of availability of an actual gas sensor, the sensor was simulated on the microcontroller. In this chapter, the commercial viability and performance of the model in comparison with other models will also be discussed. The value of using machine learning for developing a conversion algorithm will also be discussed, as well as limitations in this thesis.

6.1 THE COMMERCIAL VIABILITY OF THE MODEL

To assess the commercial viability of the model, a theoretical gas sensor containing a MOX gas sensor combined with the machine learning model will be used. The performance of this theoretical gas sensor is assumed to be equal to the performance presented in table 3.

The performance is compared with the gas sensors in the summary table from AQ-SPEC, a copy of this table is available in appendix A. When only looking at the R2 score of the model developed in this thesis, its performance is comparable with the most expensive gas sensors in the list, and also outperforms most of them, while also covering more gasses. The model is well within the top 10 gas sensors in the list, both when looking at each individual gas component and the average R2 score. This shows that this product would not only be commercial viable, but be among the highest performing gas sensors.

However, there are limitations with comparing this theoretical product with the gas sensors in the summary table. The main limitation is that the performance achieved in this thesis is completely based on the dataset produced from one gas sensor, whereas the sensors in the summary table are commercial available sensors tested in the field by a independent third party. This makes the results from the summary table more trustworthy and true to reality.

Another limitation is that the gas components measured are different, both in term of number of gas components, and which gas components that are being measured. As an example, the “2B Technologies POM” sensor in the summary table cost $4,500, measures only O_3 and have a field R2 score of 1.00, but the model in this thesis does not measure O_3 and can therefore not be compared with this sensor.

Out of the gas components measured by the theoretical gas sensor in this thesis, only NO_2 and CO exists in the summary table by AQ-SPEC. If focusing on only these two gas components, the highest R2 score for NO_2 is 0.77 in the field, and 0.98 in the lab, and for CO the highest R2 score is 0.84 – 0.90. The NO_2 score of our best model is 0.949 and for CO it is 0.973. This shows that based on the results from training and testing the machine learning model, the performance is as good, or better than the best sensors in the summary.

6.2 MACHINE LEARNING MODEL PERFORMANCE

As this thesis is uses machine learning on a dataset, it is useful to compare the results from this thesis, with the results from previous work. Unfortunately, only two papers were found that utilized the same dataset as used in this thesis: Førde's master thesis, and a research paper by De Vito et al.

The R2 score achieved by Førde in her thesis is lower than the R2 score achieved in this thesis. She got an R2 score of 0.80 for NO_2 and 0.43 for CO, and this paper got a R2 score of 0.95 for NO_2 and 0.97 for CO. This is probably due to different approaches taken. She treated each of the MOX sensor elements as a single sensor, and tried to predict one gas component (Førde, 2019). This thesis used all the MOX sensor elements as one unified sensor, and used that to predict all the gas components. She also used different libraries, and tested different models than was done in this thesis.

The research paper by De Vito et al used some of the same techniques as was used in this paper, however, the networks used were smaller. This may be because of limited computing resources and development tools. The MAPE score achieved by De Vito was 2.0, 24.6, 22.0, and 42.0 and this thesis achieved a MAPE score of 0.95, 5.88, 4.32, and 5.46 for C_6H_6, CO, NO_2, and NO_x respectively. As can be seen from the numbers listed, the results in this thesis yielded overall a lower MAPE score.

This shows that the machine learning performance achieved in this thesis is the highest performance achieved on this dataset, and is not negatively affected by being developed for implementation on a embedded system.

6.3 GENERALIZATION OF THE MODEL

The model developed in this thesis is specialized and will only be able to convert gas sensor readings into gas concentrations on the sensor used to create the dataset. However, the techniques used, and the neural networks developed in this thesis can be used on other similar datasets with only minor alterations. The only assumptions taken in the development of the model is that the sensor readings are continual and not categorical, thus to use this model for predicting concentration of other gas components or use other sensors, all that needs to be done is to create a new dataset and re-train the models.

6.4 TUNING THE MODEL

When creating neural networks for converting the type of sensor data used in this thesis, there are two major parameters that can be tuned: the number of layers and the number of nodes in each layer. In this thesis the majority of tuning revolved around varying the number of neurons in each layer, and keeping the number of layers small, as well as tuning regularization parameters. The reason for this was to make the training process manageable, and keep the network size small, as it took about an hour to train each model. With more time and better hardware, it is feasible to do further tuning and achieve even greater performance.

The number of different activation functions could also have been increased. Only the **tanh** and **relu** activation function were evaluated, but the **relu** activation function yielded far superior performance over **tanh**. Thus the majority of tuning was done using the **relu** activation function.

6.5 EMBEDDED PERFORMANCE

There are multiple tools and methods that can be used for reducing the compute time and memory usage of the machine learning model on the embedded system. This was not done since the these metrics was acceptable for many different applications. By using these tools, the evaluation and testing part of this thesis would be more complex and time-consuming, as well as taking away time that was used on developing and tuning the machine learning model.

Depending on the final application that is to use the machine learning model, these tools and methods may be needed, as optimizing the model can greatly affect the price of the microcontroller, and battery life of the system.

6.6 THE VALUE OF USING MACHINE LEARNING

In this thesis, the goal was to use machine learning to create a conversion algorithm on raw sensor values. Through this thesis it have been showed that this is possible, and that high accuracy can be achieved. Now the question is if machine learning is beneficial to use over other traditional methods for conversion.

Traditional methods yields simpler, and thus faster algorithms than machine learning (usually) achieves. However, the machine learning models were fast enough for the real time requirements of most gas sensing applications, and thus this is not an issue. One exception is for battery powered devices, as the increased compute time will affect battery life. Another exception is for high frequency real time systems, where the increased compute time may interfere with the real time requirements.

There are MOX gas sensors in the AQ-SPEC summary table, and the performance achieved in this paper is higher than the performance of the gas sensors in the summary table. This may imply that generally higher performance can be achieved by using machine learning than by traditional methods. This also reduces the required sensor specific knowledge to create the conversion algorithm, and may reduce the development cost.

Although higher performance can be achieved by using machine learning, this also requires a large high quality dataset. The process of generating such a dataset may not always be feasible or cost effective. If the either the size or quality of the dataset is not sufficient, machine learning may not produce a model that is good enough for the intended application.

6.7 LIMITATIONS

Although the results showed great commercial potential, there is many limiting factors to the value of the product that have been developed in this thesis. The biggest limitation is the lack of testing on an actual sensor. All the work is based on a dataset produced from one sensor, and all the readings are the hourly average sensor reading. This makes it impossible to know the accuracy the sensor have in a shorter time span. The lack of a sensor also limits the ways in which it can be tested in different environments.

All the performance metrics used for comparing the model developed in this thesis with the sensors in the AQ-SPEC summary table, are based on the test set and may therefore not be representative of the actual performance.

Calibration of the sensor is also not taken into account, and is something that probably needs to be adjusted for in a final model or product. It was not possible to adjust for calibration in this thesis due to the fact that all the readings are from the same sensor.

7 CONCLUSION

7.1 SUMMARY

The goal of this thesis was to develop and use embedded machine learning for converting raw sensor readings into sensor values. The results from this thesis clearly shows embedded machine learning can be used for this purpose, and that this can be done with great performance.

To achieve this, a dataset containing sensor readings from a MOX gas sensor paired with reference gas concentration values have been used. The machine learning model consists of a neural network developed using the Keras framework. The model was then ported over to embedded firmware by using the X-Cube-AI tool by ST Microelectronics.

Based on the work that have been done, the value of utilizing embedded machine learning have been presented. The techniques used in this thesis can with small modifications be used for creating other embedded machine learning products. The performance obtained during this thesis is as good or better than the best gas sensors in the AQ-SPEC summary table, with an average R2 score of 0.975 across the four different gas compounds predicted by the machine learning model. The machine learning performance is also the highest achieved performance on the dataset that were used, based on the research that have been found.

7.2 RECOMMENDATIONS AND FURTHER WORK

7.2.1 CREATE A NEW DATASET

The dataset used in this thesis is small and based on a unknown MOX sensor. Thus creating a new larger and higher quality dataset is preferred. The new dataset should contain sensor readings from an available gas sensor, combined with reference measurements from multiple different gas components, and at a higher sampling frequency. This allows more flexibility regarding the machine learning algorithm, and the capabilities of the sensor can be better analyzed.

If multiple identical sensors are used in parallel for creating the dataset, the possibility of creating an calibration process is made possible. This will also reduce the time needed for creating the dataset, as values can be measured in parallel.

7.2.2 NETWORK TUNING

Quantization and pruning of the neural network are also worth exploring, as this may both reduce the compute time, and memory requirements. Before this work is done, it should be evaluated if the increased performance is required for a given task. The sensor values used in this thesis are based on the hourly average of the sensor readings, which may imply that a compute time of 5ms is acceptable.

7.2.3 FOCUSING ON COMPLEX SYSTEMS

The main value of using machine learning for generating conversion algorithms when using it to describe complex systems. If a new sensor only produces one sensor value (e.g. a MOX gas sensor with only one sensing element), machine learning may not be beneficial. It is when the system becomes complex with multiple sensing elements, machine learning becomes valuable. If a new MOX sensor containing hundreds of different sensing elements were developed, the accuracy of the model may become even higher. Machine learning can also be used for finding which sensing elements are useful for predicting which gasses, and smaller specialized sensors can be made.

8 BIBLIOGRAPHY

Akram, A. (2017). *A Study on the Impact of Instruction Set Architectures on Processor's Performance*. 189–214.

AQ-SPEC. (2019). Air Quality Sensor Performance summary table. Retrieved from http://www.aqmd.gov/aq-spec/evaluations/summary-gas

ARM. (2019a). CMSIS Version 5.6.0. Retrieved from https://arm-software.github.io/CMSIS_5/General/html/index.html

ARM. (2019b). CPU architecture. Retrieved from https://developer.arm.com/architectures/cpu-architecture

Bainbridge, W. S. (2012). *Artificial Intelligence* (pp. 464–471). pp. 464–471.

Bhat, R., & Goh, K. M. (2017). Sonication treatment convalesce the overall quality of hand-pressed strawberry juice. *Food Chemistry*. https://doi.org/10.1016/j.foodchem.2016.07.160

Burgués, J., & Marco, S. (2018). Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated MOX sensors. *Analytica Chimica Acta*, *1019*, 49–64. https://doi.org/10.1016/j.aca.2018.03.005

Chantelle, D. (2017). Programming Languages for Embedded Systems 101: Background and Resources. Retrieved from All About Circuits website: https://www.allaboutcircuits.com/news/programming-languages-for-embedded-systems-101-background-and-resources/

Das, H., Barik, R. K., Dubey, H., & Roy, D. S. (2018). *Cloud Computing for Geospatial Big Data Analytics: Intelligent Edge, Fog and Mist Computing*. Retrieved from https://dl.acm.org/citation.cfm?id=3350446

De Vito, S., Piga, M., Martinotto, L., & Di Francia, G. (2009). CO, NO2 and NOx urban pollution monitoring with on-field calibrated electronic nose by automatic bayesian regularization. *Sensors and Actuators, B: Chemical*, *143*(1), 182–191. https://doi.org/10.1016/j.snb.2009.08.041

E Silva, D. G., Jino, M., & De Abreu, B. T. (2010). Machine learning methods and asymmetric cost function to estimate execution effort of software testing. *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, 275–284. https://doi.org/10.1109/ICST.2010.46

Førde, J. (2019). *Development of algorithm for preprocessing and prediction in Capacitive Micromachined Ultrasonic Transducers*. Norwegian University of Life Sciences.

Gibson, B. R., Rogers, T. T., & Zhu, X. (2013). Human Semi-Supervised Learning. *Topics in Cognitive Science*, *5*(1), 132–172.

Hahn, P. (2019). Artificial intelligence and machine learning. In *Handchirurgie Mikrochirurgie Plastische Chirurgie* (Vol. 51). https://doi.org/10.1055/a-0826-4789

Indiana University. (2018). What is the difference between a compiled and interpreted program. Retrieved from https://kb.iu.edu/d/agsz

Joshi, P. (2016). *Python Machine Learning Cookbook*. Retrieved from http://proquest.safaribooksonline.com.ezproxy.lib.vt.edu/9781786464477

Lai, L., Suda, N., & Chandra, V. (2018). *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs*. 1–10. Retrieved from http://arxiv.org/abs/1801.06601

Moons, B., Bankman, D., & Verhelst, M. (2019). Embedded Deep Learning. In *Embedded Deep Learning*. https://doi.org/10.1007/978-3-319-99223-5

Pi Raspberry. (2019). *Raspberry Pi 4 Computer*. (June).

Sensirion. (2019). Multi pixel gas sensors. Retrieved from https://www.sensirion.com/en/environmental-sensors/gas-sensors/multi-pixel-gas-sensors/

Shubman, J. (2018). An Overview of Regularization Techniques in Deep Learning. Retrieved from analytics vidhya website: https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

Ŝimunić, T., Benini, L., & De Micheli, G. (2001). Energy-efficient design of battery-powered embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *9*(1), 15–28. https://doi.org/10.1109/92.920814

ST Microelectronics Inc. (2017). *STM32F765xx, STM32F767xx Datasheet*. (May). Retrieved from https://pdf1.alldatasheet.com/datasheet-pdf/view/933989/STMICROELECTRONICS/STM32F767ZI.html

ST Microelectronics Inc. (2019). *X-CUBE-AI Data brief Artificial intelligence (AI) software expansion for STM32Cube X-CUBE-AI*. (October).

ST Microelectronics Inc. (2020). STM32CubeMX. Retrieved from https://www.st.com/en/development-tools/stm32cubemx.html

Sun, J., Lu, S., Pang, W., & Sun, Z. (2019). Deep representation learning with feature augmentation for face recognition. *2019 IEEE 4th International Conference on Signal and Image Processing, ICSIP 2019*, 171–175. https://doi.org/10.1109/SIPROCESS.2019.8868386

UCI. (2016). Air Quality Data Set. Retrieved from https://archive.ics.uci.edu/ml/datasets/Air+Quality

UST. (2017). UST Miniaturized VOC sensor. Retrieved from http://www.umweltsensortechnik.de/en/gas-sensors/vocco2-sensor.html

Valvano, J. W. (2017). *Embedded systems: Introduction to ARM cortex-m microcontrollers* (5th ed.). Self published.

9 APPENDIX A

This appendix contain a copy of the table found on the AQ-SPEC website where links do documents and images have been removed. The information in the table were collected 2. may 2020, and may therefore be outdated at the time of reading. The complete table can be found here: http://www.aqmd.gov/aq-spec/evaluations/summary-gas.

Table 7: Summary of different gas sensor performance (AQ-SPEC, 2019).

Make and Model	Est. Cost (USD)	Type	Measurement	Field R2	Lab R2
2B Technologies POM	$4,500	UV absorption	O_3	1.00	0.99
Aeroqual AQY Ver. 0.5	$3,000	Electrochem	NO_2	0.77	0.98
		Metal Oxide	O_3	0.95	0.98
Aeroqual S-500	$500	Metal Oxide	O_3	0.85	0.99
Air Quality Egg Ver. 1	$200	Metal Oxide	CO	0.00	-
			NO_2	0.40	-
			O_3	0.85	-
Air Quality Egg Ver. 2	$240	Electrochem	CO	0.00	-
			NO_2	0.00	-
Air Quality Egg Ver. 2	$240	Electrochem	O_3	0.00 – 0.20	-
			SO_2	n/a	-
AQMesh Ver. 4.0 (Discontinued)	$10,000	Electrochem	CO	0.42 – 0.80	-
			NO	0.00 – 0.44	-
			NO_2	0.00 – 0.46	-
			O_3	0.46 – 0.83	-
APIS	$4,995	Electrochem	CO	0.88	-
			NO	0.93	-
			NO_2	0.37	-
			O_3	0.77	-
CairPol Cairsens (CO)	$1,243	Electrochem	CO	0.94	-
CairPol Cairsens (NO_2)	$1,198	Electrochem	NO_2	0.00 – 0.12	-
Kunak Air A10	~$5,000	Electrochem	CO	0.58	-
			NO	0.87	-
			NO_2	0.29	-
			O_3	0.87	-
Magnasci SRL uRADMonitor INDUSTRIAL HW103	~$1,300	Electrochem	CO	0.03	-
			NO_2	0.03	-
			O_3	0.03	-
Perkin Elmer ELM	$5,200	Metal Oxide	NO	n/a	-
			NO_2	0.00	-
			O_3	0.89 – 0.96	-

Smart Citizen Kit	$200	Metal Oxide	CO	0.50 – 0.85	-
			NO_2	0.00	-
Spec Sensors	$500	Electrochem	CO	0.84 – 0.90	-
			NO_2	0.00 – 0.16	-
			O_3	0.00 – 0.24	-
uHoo	$300	Metal Oxide	CO	0.00	-
			O_3	0.43 – 0.72	-
UNITEC SENS-IT (CO)	$2,200	Metal Oxide	CO	0.33 – 0.43	0.99
UNITEC SENS-IT (NO_2)	$2,200	Metal Oxide	NO_2	0.60 – 0.65	-
UNITEC SENS-IT (O_3)	$2,200	Metal Oxide	O_3	0.72 – 0.83	0.82 – 0.90
Vaisala AQT410 Ver. 1.11	$3,700	Electrochem	CO	0.28 – 0.31	-
			NO_2	0.00	-
			O_3	0.40 – 0.58	-
			SO_2	n/a	-
Vaisala AQT410 Ver. 1.15	$3,700	Electrochem	CO	0.80 – 0.83	-
			NO_2	0.48 – 0.61	-
			O_3	0.66 – 0.82	-
			SO_2	n/a	-

Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway