TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Tanel Peet 178248IASM

CHALLENGES IN IMPLEMENTING ARTIFICIAL NEURAL NETWORKS ON MICROCONTROLLERS: AN ARM CORTEX-M EXAMPLE

Master’s Thesis

Supervisor: Johannes Ehala
MSc

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Tanel Peet 178248IASM

VÄLJAKUTSED TEHISNÄRVIVÕRKUDE RAKENDAMISEL MIKROKONTROLLERITEL ARM CORTEX-M NÄITEL

Magistritöö

Juhendaja: Johannes Ehala
MSc

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, the literature and the work of others have been referenced. This thesis has not been presented for examination anywhere else.

Author: Tanel Peet

16.05.2019

Abstract

Implementing Artificial Neural Networks (ANNs) on embedded systems is a challenging task, but can have a big impact on improving Internet of Things (IoT) applications. The goal of this thesis is to research how to improve and simplify the process of creating ANNs for embedded systems. Developing embedded ANN solutions involves training a model on regular computer and converting it to a suitable format for the embedded system. The work concentrates on the development of audio classification solutions running on Cortex-M microcontrollers. Existing work and process for creating embedded ANN solutions was analyzed and an ANN model was developed for Cortex-M microcontrollers on a custom dataset for bird sound classification use-case. Based on the use-case experience, improvements and simplifications were proposed and implemented to the existing process and framework. The biggest improvement introduced in this work involved automating the conversion process for ANN models trained in Tensorflow by generating code for Cortex-M microcontrollers. Further improvements and analysis was introduced for reducing the effect of noise imposed by embedded systems. In addition, the effects of reducing complexity of ANNs, which is necessary for embedded systems, was analyzed.

The thesis is written in English and contains 78 pages of text, 5 chapters , 11 figures, 9 tables.

Annotatsioon

Käesoleva lõputöö eesmärk on lihtsustada ja täiendada protsessi tehisnärvivõrkude arendamiseks sardsüsteemidele. Töö keskendub heli klassifitseerimise lahendustele Cortex-M mikrokontrolleritel. Töö raames uuriti eksisteerivaid protsesse ja raamistike tehsinärvivõrkudel põhinevate sardsüsteemide arendamiseks. Selleks arendati närvivõrgu mudel Cortex-M mikrokontrolleri jaoks, mis treeniti omaloodud linnuhelide andmestikul. Arendustööst saadud kogemuse põhjal täiendati ja lihtsustati eksisteerivat protsessi tehisnärvivõrkude arendamiseks mikrokontrolleritele.

Sardsüsteemidel jooksvate tehisnärvivõrkude lahenduste loomine nõuab mudeli treenimist tavalisel arvutil ning seejärel mudeli mikrokontrollerile sobivasse formaati viimist. Suurim panus lõputöö raames oli treenitud Tensorflow mudeli automaatne muutmine Cortex-M mikrokontrollerile sobivasse formaati. See protsess sisaldab endas närvivõrgu parameetrite resolutsiooni vähendamist (kvantiseerimine) ning automaatse koodi genereerimist, mis kiirendab, lihtsustab ja väldib inimvigu treenitud mudeli mikrokontrollerile viimisel. Tehisnärvivõrkude mikrokontrollerile viimine nõuab nende arvutusliku keerukuse ning suuruse vähendamist. Selleks kasutati kompakset tehisnärvivõrgu struktuuri ning kvantiseerimist. Arvutusliku keerukuse ning suuruse vähendamisest tulenevat mõju analüüsiti nii säästetud mälu, võidetud arvutuskeerukuse kui ka klassifitseerimistäpsuse muutumise näol.

Töö raames tehtud täiendused ja analüüs sisaldas ka sardsüsteemile spetsiifilise müra mõju vähendamist, mis parandas klassifitseerimise täpsust. Täpsuse parandamiseks loodi protsess, kus treeningandmetele lisati müra, mis salvestati läbi sardsüsteemi. Lisaks analüüsiti eeltöötluse lahendust tehisnärvivõrgule vajalike tunnuste leidmiseks, mis on müra suhtes vähemtundlikum, kui eksisteerivas protsessis väljatoodud lahendus. Uuritud eeltöötluse algoritm parandas klassifitseerimise täpsust ning seetõttu arendati ka tarkvara, mis lubab antud eeltöötlust rakendada mikrokontrolleritel. Arendatud lahenduse arvutusliku keerukust uuriti ning võrreldi varasema lahendusega.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 78 leheküljel, 5 peatükki, 11 joonist, 9 tabelit.

List of abbreviations and terms

ADC	Analog-to-Digital Converter
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CMSIS	Cortex Microcontroller Software Interface Standard
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DS-CNN	Depthwise Separable Convolutional Neural Network
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations per Second
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GFCCs	Gammatone Frequency Cepstral Coefficients
IoT	Internet of Things
KWS	Keyword Spotting
MFCCs	Mel-Frequency Cepstral Coefficients
PCM	Pulse-Code Modulation

RAM	Random-Access Memory
RMS	Root Mean Square
SIMD	Single Instruction, Multiple Data
STFT	Short-Time Fourier Transform
TPE	Tree of Parzen Estimators
WASN	Wireless Acoustic Sensor Network

Table of Contents

List of Figures

List of Tables

1 Introduction

Internet of Things (IoT) has become one of the fastest growing paradigms in the history of computing [1]. IoT solutions can have a huge impact on the economy, modernization and quality of life [2]. Projections for year 2025 show that IoT related technologies could represent 11% of worlds economy, while deploying 1 trillion devices [3].

The usefulness of IoT technologies comes largely from the ability of collecting data and understanding the surrounding environment [2]. The rise in the number of IoT devices brings a challenge of how to deal with the vast amount of data generated by the devices. The concept of edge computing has been proposed to solve this problem by moving computing to the IoT devices themselves, instead of moving large amounts of raw data to central database for processing [4].

Edge devices often find usage in complex and noisy environments, making it hard to develop software for interpreting the measured data. In recent years, Artificial Neural Networks (ANNs) have shown good results in processing data from such environments, especially in the field of computer vision and speech recognition [4]. While ANNs work well with complex and noisy data, they often require a lot of memory and computational power. Edge devices on the other hand are usually affordable embedded systems with low power usage, limited memory and computational power [3]. This makes running ANNs on embedded systems a challenging task.

As ANNs have proved their usefulness in recent years, more research effort is going towards making ANN algorithms more practical for real-life problems [5]. One part of it involves research for compressing ANNs, so these could be more practical and efficient to run. This has been achieved by introducing more efficient ANN architectures and by reducing the resolution of learnable parameters. Research in this field has made it possible to apply ANNs on embedded devices, while not loosing significant amount of accuracy [6].

Research and development of embedded systems has also made them more suitable for running ANNs. There are various types of embedded systems where ANNs can be run.

General purpose microcontrollers have become more efficient, while having more memory and computational power [7]. Microcontrollers allow to easily develop preprocessing methods and ANNs with programming languages such as C or C++. Application-specific integrated circuits (ASICs) are often used to get more specialized and efficient embedded systems, which are developed for specific tasks. The drawback is that creating ASIC is much more complex, expensive and requires specialized knowledge of hardware, while also needing physical development of the hardware. For these reasons ASICs are usually used only with large production volumes. Field programmable gate arrays (FPGAs) offer a way to configure already manufactured hardware for the specific task needed, by wiring together different gates based on the programming. It also requires specialized programming skills, but unlike with ASICs this can applied on already developed hardware, which is faster and more affordable. [8]

Using ASICs or FPGAs is desirable when efficiency of embedded systems is paramount, but the drawback is that developing these solutions takes time and lengthens the development loop. As microcontrollers have become more efficient, they often satisfy the efficiency requirements, while allowing short feedback loop in research and development process. ARM has a popular Cortex-M family of microcontrollers and great ecosystem suitable for IoT devices [7]. Research Laboratory for Proactive Technologies at Tallinn University of Technology has applied IoT devices based on the Cortex-M processors. The same devices are also used in this work.

ARM has developed Cortex Microcontroller Software Interface Standard (CMSIS) [7] for standardizing and simplifying the development on different types of Cortex-M processors. Among other packages, CMSIS includes CMSIS-DSP and CMSIS-NN, which are libraries for digital signal processing and neural networks. This makes Cortex-M microcontrollers good candidate for developing edge devices, which could be used for processing and understanding complex data.

Developing ANNs for microcontrollers usually involves training an ANN on high performance computers, converting the trained model with found weights into a code which can then be transferred to a microcontroller [6]. The available research often concentrates on creating suitable ANNs and preprocessing techniques for microcontrollers, while not describing the code development and deployment process. There is also lack of research on how the noise introduced by the embedded system can affect accuracy of the trained ANNs and what countermeasures could be used.

One of the areas, where ANNs have become state of the art, and where embedded systems are often used, is automatic bird sound classification, as discussed in section 2.1.2. Automatic bird classification would help researchers to get better overview of the ecosystem while also helping to conserve the biodiversity. Usage of autonomous IoT solutions, which are affordable and easy to develop, would help to scale up the monitoring process and help researches to gain more information about our ecosystem. Thus far, the research in autonomous bird classification devices has concentrated on applying simpler machine learning methods or other statistical techniques on embedded systems, while only suggesting the possibility of using ANNs instead.

1.1 Research Problem

Making IoT devices more capable of processing and interpreting data from a complex environment helps to increase the impact of computing on our daily lives. ANNs have proven to be a good tool for interpreting complex data. Although there is research done on implementing ANNs on edge devices, the used process is often custom-tailored for certain tasks and problems [9]–[13]. Currently, the process of running ANNs on embedded systems requires expertise in machine learning for training the models, and in developing embedded systems, in order to implement the model efficiently on a edge device. This problem impedes the research for groups or individuals, who are mainly concentrated in developing machine learning models and do not have access to skills required to develop embedded systems.

One of the research areas where ANNs have proven to be useful is bird sound classification. The field would also benefit a lot, when these researchers could easily apply their trained models to embedded systems, allowing to test the solutions in real-life [14]. The problem is that researchers in this area often lack the skills and knowledge about developing embedded systems, as they are specialized in developing preprocessing and machine learning methods [15].

Running ANNs on microcontrollers raises several problems. The memory limitations and available computational power needs to be taken into account when selecting preprocessing methods or architecture of an ANN. Training of ANNs usually occurs on high performance computers, using high level languages, raising a problem of how to convert the model into suitable format for microcontrollers. This also includes solving the problem of how to do it efficiently for the specific hardware. There could be other aspects of embedded systems which introduce problems for researcher who are dealing with model developments, such as noise in collecting data, which can be specific to the embedded system. The mentioned problems related to running ANNs on a microcontroller are researched in this work, to introduce new research opportunities for audio classification researchers. The work focuses on the following problems:

- how to develop complex ANNs for microcontrollers which have limited resources;
- how to deal with noise specific to the embedded device;
- how to automate the processes in developing complex ANNs for microcontrollers.

1.2 Method

The problems involved with running ANNs on IoT devices are researched in this thesis by implementing an ANN on microcontroller. This involves training the ANN on a custom dataset and researching ways to improve the process of converting that model to a suitable format for microcontrollers.

ARM Cortex-M microcontroller is chosen for this work, as it is widely used and offers good ecosystem for developing such solutions. ARM's CMSIS-NN library offers building blocks for ANNs and standard interface, which allows to run the solution on all Cortex-M microcontrollers. ARM has provided an example project called Hello Edge, accompanied with a research paper [6], to show how ANNs can be developed on Cortex-M microcontrollers for an audio classification task. The project includes scripts for preprocessing data, training various types of ANNs and a manual for quantizing trained models, so they could be run on microcontrollers. There is also an example of C++ code, to demonstrate how ANNs could be implemented with CMSIS-NN. The Hello Edge work is taken as a basis for this thesis for training and running an ANN on Cortex-M device.

The questions of how to implement ANNs on microcontrollers and how to deal with noise introduced by the embedded system are researched by an audio classification use-case. The use-case involves creating an ANN and accompanying preprocessing techniques for bird sound classification. A custom dataset is created for this purpose, which is largerly based on the author's previous thesis [16].

1.3 Overview of the Thesis

The necessary background information and theoretical knowledge relevant for this thesis is introduced in section 2. Necessary aspects about machine learning and embedded systems are covered in this section, while also giving brief overview about bird sound classification.

The overall technical implementation for preprocessing, training the ANN and converting the model to C++ code, together with improvements to the Hello Edge work, are described in section 3. This includes overview of used frameworks, tools and the embedded system. The description of how data gathering, preprocessing, training, model conversion and implementation on the embedded system was performed is also given in this section.

The ways to make ANNs more efficient and the impact of preprocessing are analyzed in section 4. This includes analysis and effects what replacing Convolutional Neural Networks (CNNs) with Depthwise Separable Convolutional Neural Networks (DS-CNNs) and quantization have on the classification accuracy. The effects of adding background noise to training samples and comparison of different preprocessing techniques can also be found in this section.

2 Background

The section gives background information necessary for understanding rest of the thesis. The section starts with describing the problem of automatic bird classification, by first looking at the properties of the bird sounds, followed by the research done on automatic bird classification. The concepts relevant to machine learning are then explained, which includes explaining relevant machine learning algorithms and techniques used throughout this work.

2.1 Automatic Bird Classification

Automatic bird classification improves the way how birds are monitored, which benefits ecological surveillance and conservation of biodiversity [17]. This section gives overview of the challenges and methods for automatic bird sound classification. The section starts by introducing the properties of bird sounds, to describe the complexity of the challenge. This is followed by an overview of the state of the art solutions in bird classification and research on how the task is performed on embedded systems.

2.1.1 Properties of Bird Sounds

Birds generally depend on vocal signals in order to communicate over long distances, in poor light conditions or when communicating through dense vegetation [18]. Sounds produced by birds can be divided into two groups – calls and songs. Calls are simpler sounds that are used in a variety of contexts, such as providing information about food sources or dangers. Calls are produced by both females and males. Song is associated with more complex and elaborate sounds and it is being used in the context of courtship and mating. Birdsong is usually produced by males, although there are exceptions. Birds can also create other types of distinguishable sounds, such as pecking of the woodpecker or fast flapping of wings [19].

Most birds have several versions of their species songs, where each version is called a song type. Each instance of a song is called a strophe and it usually occurs in bouts.

Birdsong usually consists of a number of distinct sections, which are called phrases. Each phrase can consist of series of units which form particular patterns, as illustrated on figure 1. These units are called syllables and consist of one or several elements, sometimes also called notes [20].

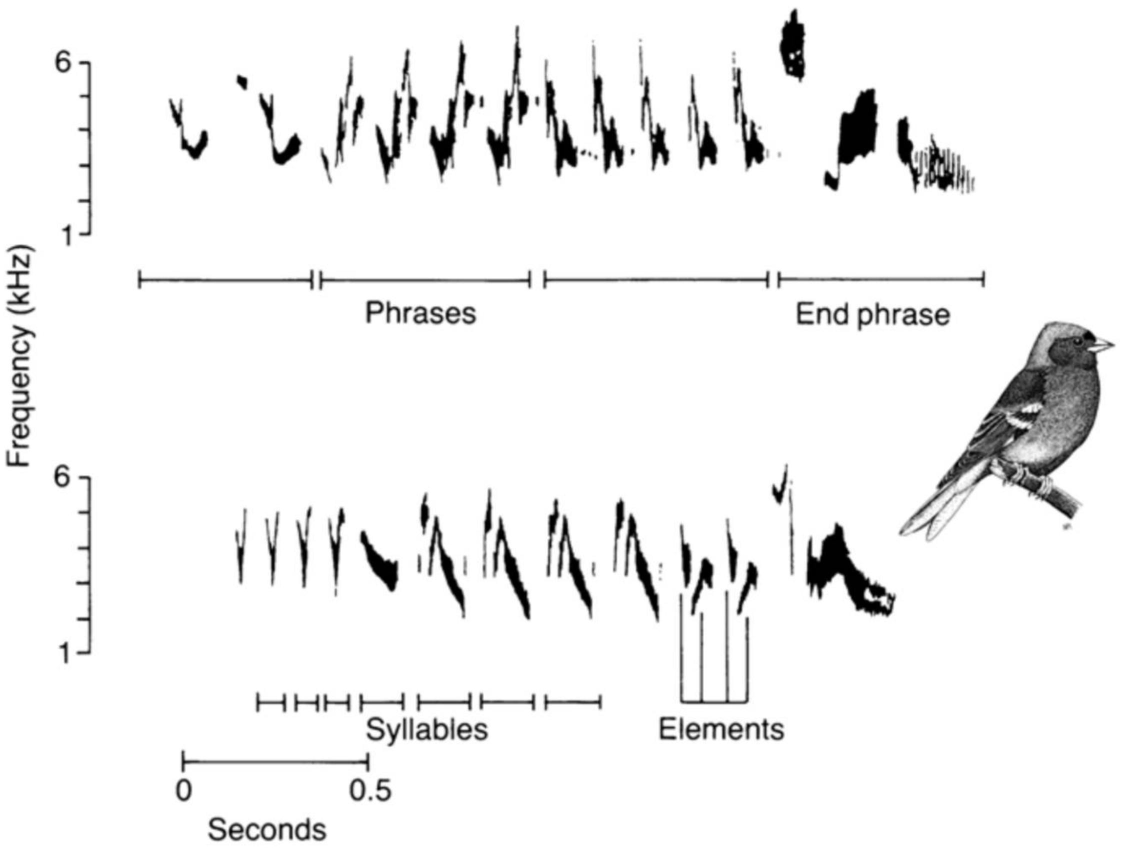


Figure 1. Sonogram illustrating elements, syllables and phrases of a male chaffinch [20].

Different songbirds can have different vocabulary – some birds can produce only one syllable, while some can repeat a very large vocabulary of syllables in different patterns. Birdsong is slightly less complex than human speech, as there appears to be a less intricate grammar and there is no vowel-consonant alternation. Nevertheless, the order and pattern of syllables in birdsong, including pauses in between syllables, hold important information for distinguishing different species [21].

Birdsong from different species can vary in several dimensions – frequency range, speed of pitch modulation, vocabulary size, syllable duration and song duration. Species cannot be uniquely distinguished by looking only one of these dimensions. For example, some species can have song patterns with same duration and other species can sing in the same frequency range. Therefore, it is necessary to look different dimensions of the signal, in order to distinguish the species from song [21].

The variation in bird sounds occurs also between the individuals from one species. Repertoire of an individual is dependent on gene heritage, but also on the condition during development of a young individual or on adult body conditions. In addition, the songs of individuals are affected by their age, population size and density, geographical location and social interactions. The effect of song sharing between neighboring birds leads to local dialects, also called microdialects, which means that birds from the same species in one area can have quite a different songs compared to some other geographical location [22].

Properties of Great Tit Song

Great Tit (*Parus Major*) is one of the most common songbirds in Estonia [23] and most recorded Estonian bird species in Xeno-canto[1]. Because of the bird's readily available sounds and wide spread in nature, Great Tit sound is chosen as the main bird sound to be classified in this work.

Great Tit song frequency range is usually between 3 - 7 kHz, while the most common frequency is near the 3.5 - 4.5 kHz [24]. It has been observed that the frequency range and patterns of the Great Tit song are influenced by the surrounding environment, especially in a noisy urban settings [25]. The distribution of frequencies from all the Great Tit's audio recordings, being used in this work, can be seen in figure 2.

The repertoire of Great Tits consists of one to eight distinct phrase versions, each containing one to more than five elements. Usually one to 20 phrases are repeated in one version of a song. The strophe of a Great Tit is mostly between one and five seconds long, with the pause of a few seconds between strophes. Up to 100 strophes of a single phrase version are usually sung in a bout. After a bout, the Great Tit can stop singing or start a bout of another phrase version [24].

[1]Xeno-Canto is crowd-sourced database of bird sounds: `https://www.xeno-canto.org/`

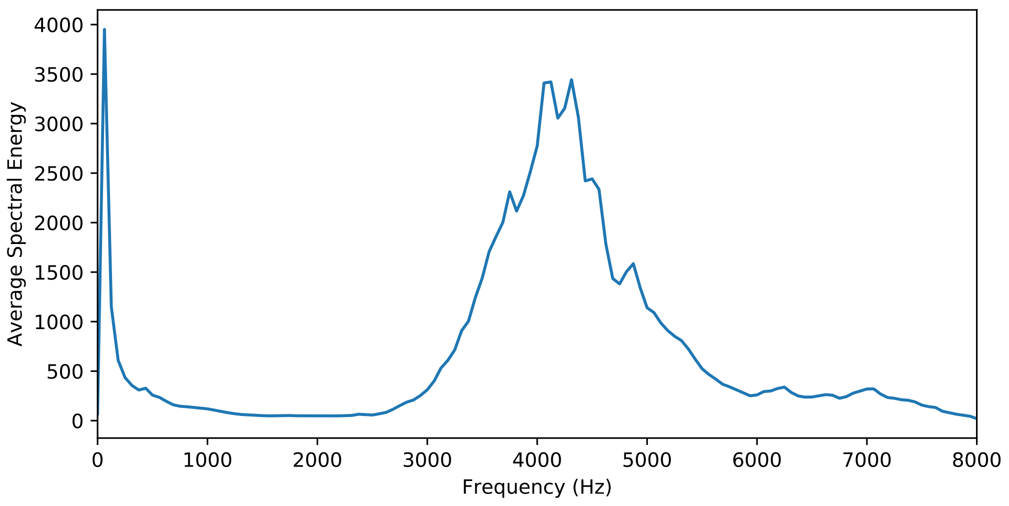


Figure 2. Distribution of frequencies in the recordings of Great Tits used in this work.

2.1.2 Machine Learning Methods for Bird Classification

Automatic bird classification is not a new research area and has been the topic of interest for many research groups. The summary of the current state of the art and trends in the field is given by describing the results of automatic bird classification competitions. This is followed by looking at works, which concentrate on applying bird song classification techniques to embedded system.

Overview of Bird Classification Competition Results

Since 2013, there has been several competitions, where different solutions of automatic bird classification were compared on common datasets in determining the best approaches. These competitions offer a great overview about the state of the art of bird classification during the last five years. Datasets used in the competitions vary from challenge to challenge, making it hard to compare the results from different years.

The tasks and datasets were different for MLSP 2013 Bird Classification [26] and NIPS4B 2013 Multi-Label Bird Species Classification [27] challenges, but the presented works were very similar to each other. For both challenges, the most popular machine learning method was ensembled random forests, which used features extracted by applying computation-heavy image processing techniques to spectrograms. It is worth mentioning that in MLSP 2013 challenge, there was one work which used a Convolutional Neural

Network (CNN) and achieved 4th place [26]. Since 2014, there has been annual LifeCLEF competition, which has also a bird classification task called BirdCLEF. 2014's [28] and 2015's [29] competitions had similar solutions to the MLSP 2013 and NIPS4B challenges, where Mel-Frequency Cepstral Coefficients (MFCCs) were often used, together with other features, as an inputs to random forests algorithms.

BirdCLEF 2016 [30] can be seen as a turning point, where CNNs started to significantly outperform works which mostly invested in feature engineering. Results of BirdCLEF 2017 [15] reinforced the success of CNNs, where state of the art architectures from image classification tasks were applied on spectrograms of bird sounds. BirdCLEF 2018 [17] introduced further improvements, using state of the art image classification solutions, while also showing that smaller CNN architectures can achieve comparable results to the complex architectures.

The overview of these competitions showed how Artificial Neural Networks (ANNs), and CNNs in particular, have started to dominate the competitions within the last five years. The solutions introduced in these competitions, however, are quite complex and require lots of computing power and memory, making them unsuitable for embedded systems.

Bird Sound Classification on Embedded Systems

The importance of having autonomous devices for classifying birds is discussed in large portion of the research done in the field of bird sound classification. Different approaches for this have been suggested in the literature. Most of the resources concentrate on having autonomous low power recording devices on the field and later processing the results off-line on more capable computers.

The work of [31] introduced using a digital recording device with microcontroller which turned it on for seven minutes in each hour. Their following work then included automatated classification of bird and amphibian species from these recordings [32]. ARBIMON system was developed by [33], which combined the recording and off-line classification into a single system. It included recording audio clips with iPod, from where it was sent to field base station for preprocessing and storage, before sending the data over Internet to the server for classification. Solution named AMBIO was developed in [34], where sounds, together with environmental information, such as temperature and wind, were recorded and sent to a central processing through 3G network.

More compact and scalable solution was provided in [35], where Wireless Acoustic Sensor Network (WASN), with ARM Cortex-M nodes, was deployed. There was a central node which recorded the data for later off-line processing. Similar study was done in [36], with the exception of adding classification capabilities to the central node. The work proposed a centralized WASN architecture where nodes send microphone signals through gateway to a Field Programmable Gate Array (FPGA), which performs feature extraction and classification on site.

The work in [37] further improved the cost and scalability of a WASN, by moving classification into the nodes themselves. Texas Instrument's TMS320F2812 Digital Signal Processor (DSP) was used to classify bird songs in real-time with a statistical model which used multivariate Gaussian distributions with covariance matrix on power spectral values of the signal. Liu et. al. [38] used FPGA as a sensor node for classifying bird sounds real-time. Template matching was used together with linear predictive cepstral coefficients extraction and dynamic time warping as a preprocessing step. Wei et. al. [39] used ARM Cortex-A9 as sensor nodes for classifying bird songs in real-time using sparse approximation based classification. Minimum distance classifiers was used in [14] on MFCCs features on ARM Cortex-M4F microcontroller, while also suggesting a possibility of using an ANN as a classifier instead.

The overview of bird classification efforts on embedded systems show the importance of using WASNs. Based on the overview, there seems to be a trend of moving classification towards edge, as embedded hardware becomes more powerful and efficient. This helps to improve the cost and scalability of the developed solutions, allowing to monitor larger areas. Overview of the research showed that, while CNNs have become state of the art in bird classification competitions, there is still lack of research about applying these networks to embedded systems.

2.2 Machine Learning

Machine learning is a field of computer science, which covers algorithms able to learn from data, without being explicitly programmed. It involves designing and studying algorithms for solving certain tasks by learning from data. Machine learning finds use in solving tasks where explicit programming is unfeasible, such as computer vision, speech recognition and spam filtering [40].

There is a large variety of tasks which machine learning covers. This section concentrates only on classification tasks and covers the theory behind the relevant topics for this thesis. Classification task involves identifying which category a new data sample belongs to. These tasks require to have a set of samples, where the category for each sample is known. These categories are often called labels or classes in machine learning literature. To classify new samples, without the known category, a classifier needs to be trained. During the training phase, samples paired with corresponding labels are presented to a machine learning algorithm. Well designed and properly used machine learning algorithm should then find the underlying features and properties of the data which are important in determining the label of the sample. The result of the training phase is the machine learning model, which can be used for classifying new samples. To actually understand how well that model is performing, it needs to be evaluated and tested with data it has not seen yet [40].

Rest of this section will cover the theory behind machine learning in more details. This includes overview of how data is used in machine learning, how it is prepared and preprocessed. Relevant machine learning algorithms are described, together with how to validate trained models and how to find suitable hyperparameters for the algorithms. The section also covers ways to compress machine learning models, so they could be run on microcontrollers.

2.2.1 Data in Machine Learning

The set of data used in machine learning is called a dataset and it includes the samples and corresponding labels. Before starting to use machine learning algorithms, the data should be split into three non-overlapping sets – training set, validation set and testing set [40].

Machine learning algorithm learns from training data to solve a given task. Developing good machine learning models takes lots of training iterations, often involving testing out different preprocessing and machine learning algorithm hyperparameters[1]. Evaluation set is used to measure the changes in model performance under these changes, as using training data for this purpose can cause the model to overfit. Overfitting can be looked as a situation where machine learning algorithm memorizes the seen data and looses the ability to generalize on unseen data. Before deploying the model or reporting about the

[1]The word *hyperparameter* is used to distinguish the parameters defined before the training phase, as word *parameter* is used for the values which are being refined and fitted to the data during the training phase

performance of it, the model should also be evaluated on a testing set. This is needed, as we might have chosen the hyperparameters in a way, that best suited the data in the evaluation set, but might not perform as well on unseen samples [40].

Complex problems with noisy data often require a lot of training samples. Collecting additional training data is usually very costly, if not impossible. Therefore data augmentation and synthesis is often used to generate new data, so the machine learning model can learn to generalize better on unseen samples. For example, in sound classification, data augmentation might include stretching time, shifting pitch or adding background noise [41].

2.2.2 Preprocessing and Feature Engineering

Before feeding data to the machine learning algorithm, it needs to be preprocessed into suitable format. Most machine learning algorithms require data to be in a tabular form, where each row is a single sample and each column corresponds to a feature [42]. Convolutional neural networks however, allow each sample to be a matrix or even several stacked ones, such as images with red, green and blue channels [40].

Feature engineering is the practice of applying domain knowledge for extracting properties from data which are useful for learning purposes and improve the accuracy of machine learning algorithms. The process of feature engineering is often very time consuming and requires good understanding of the problem domain. ANN algorithms have the power to learn the features themselves, but this will usually come with the cost of larger and more complex models. For ANNs, the work is shifted from feature engineering to finding suitable network structure and other hyperparameters [40].

Overview in section 2.1.2 showed that most works in classifying bird sounds transformed the data from time domain to frequency domain during the preprocessing, which was often followed by a feature extraction step. Some of the relevant feature extraction methods are further described in this section.

Mel-Frequency Cepstral Coefficients (MFCCs)

MFCCs are features which are often used when classifying different sounds and which were engineered to mimic how humans perceive sounds. MFCCs feature extraction takes the power spectrum of frequencies as an input, which involves taking the absolute value of the squared complex numbers from the Fourier transform. MFCCs are calculated separately for each slice of the Short-Time Fourier Transform (STFT). Output of MFCCs feature extraction, from a slice of STFT, is an array, usually having less than 26 elements [43].

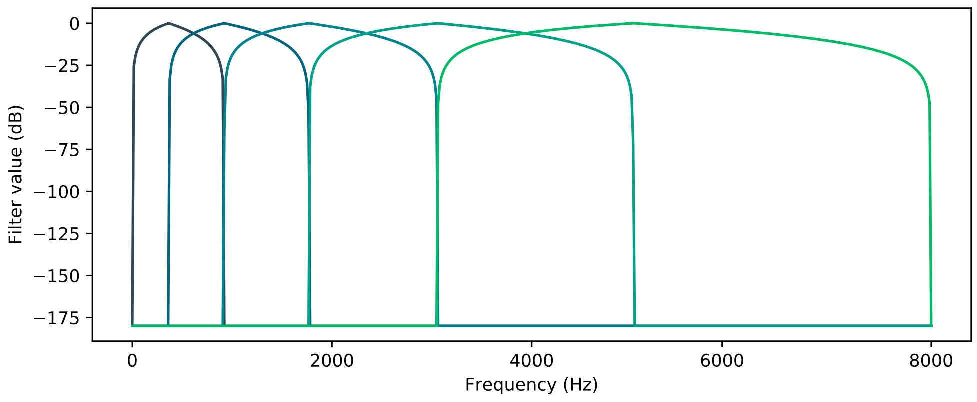
The human cochlea can not differentiate between two close frequencies and the effect grows as frequencies get higher. Mel filterbank is used to simulate this effect by applying filters and summing up close-by power spectrum bins in order to understand how much energy exists in different frequency regions. Mel filterbank consists of user-defined number of triangular overlapping filters. The illustration of these filters in logarithmic scale can be seen on figure 3a. The first filters are quite narrow, while getting much wider for higher frequencies. Mel scale is a function which helps to calculate the width and spacing of these filters. Each filter in the filterbank is presented as an array with the same number of elements as the power spectrum array. The filterbank vector elements are multiplied by the corresponding power spectrum elements and summed together, which describes how much energy is in each filter [43].

Humans do not perceive loudness on liner scale and therefore the logarithm is taken from the filterbank energies. The Mel filterbanks are overlapping and therefore correlated. The Discrete Cosine Transform (DCT) is therefore applied to the logarithms of filterbank energies to decorrelate these energies. This results in an array with the same number of elements as in filterbank energy array. More information and slower changes in filterbank energies are packed to the first DCT coefficients. Higher coefficients involve faster changes in filterbank energies and might even degrade the classification results. A user-defined number, which is often half of the number of filterbank filters, is therefore taken as an input to classification step. These chosen coefficients are called MFCCs [43].

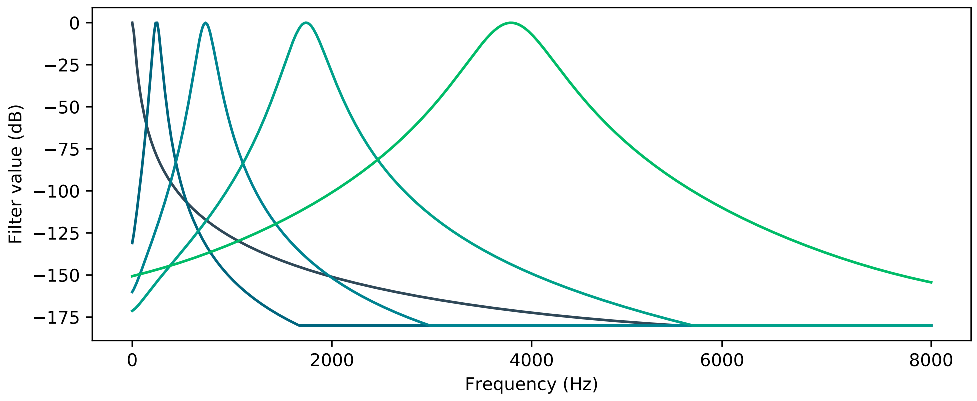
Gammatone Frequency Cepstral Coefficients (GFCC)

Several studies show that MFCCs work well with clean signal, but when noise is introduced, the classification accuracy will decrease drastically [44]–[48]. The usage of Gammatone Frequency Cepstral Coefficients (GFCCs), instead of MFCCs, has shown improvements with regards of noise robustness in several works [49]–[51].

Computing GFCCs follows similar steps as finding MFCCs. One key difference is the



(a) Five Mel filterbank filters.



(b) Five gammatone filterbank filters.

Figure 3. Filterbank filters for MFCCs (a) and GFCCs (b).

filterbanks which are used in the process, as Mel filterbank is replaced with gammatone auditory filterbank [48]. The difference between the filterbanks is presented on figure 3.

Another difference between GFCCs and MFCCs is what type of compression is used for the calculated filterbank energy coefficients. MFCCs used logarithmic compression, which is also used in some implementations [48] of GFCCs, while other implementations use cubic root compression [50].

2.2.3 Machine Learning Algorithms

There are many machine learning algorithms for classification tasks. The overview of research given in section 2.1.2 showed that CNNs have become state of the art in the field of bird sound classification. This section starts by describing the general theory behind

ANNs and then introduces CNNs. The section finishes by describing Depthwise Separable Convolutional Neural Network (DS-CNN) architecture, which reduces the complexity of CNNs and makes it possible to run ANNs on embedded systems.

Artificial Neural Networks

Artificial Neural Network (ANN) is a machine learning algorithm loosely modeled after how neurons in biological brains work. It contains large collection of connected nonlinear elements called neurons. Each neuron has summation function combining all the values of its inputs together. ANNs usually consist of multiple layers, where neurons from one layer are connected to the neurons on the next layer. These type of layers are called fully connected layers. ANNs with several hidden layers are called deep neural networks and the process of training these networks is called deep learning. Deep learning models represent the information as a nested hierarchy of concepts, where each concept is defined in relation to simpler concepts from previous network layers [40].

The first layer of ANN is called input layer, while the last is called output layer. Hidden layers are all the layers between the input and output layer. Each connection between two neurons has a weight. The value of the neurons in hidden and output layers are found by adding together the product of neurons from the previous layer, and weights for the corresponding connections, plus the bias for that neuron, followed by applying activation function on the result. The goal of activation function is to introduce non-linearity to the mode, which allows to solve more complex tasks [40].

ANN training involves optimization algorithm which changes the weights and biases in order to minimize the value of a loss function. Output of the loss function represents the difference between the classified labels and true labels of samples. One of the hyperparameters of optimization algorithms is the learning rate, which defines how much to change parameters during each update step. If the learning rate is too small, the learning might be too slow, if it is too large, it might result in fluctuations around the minimum, without ever reaching it [52].

Convolutional Neural Networks

Convolutional Neural Network (CNN) is a type of Artificial Neural Network which uses convolution operation in at least one of its layers, in place of general matrix multiplication, as in fully connected layers. The main benefit of convolutional layer is reducing the amount of parameters needed in learning process. CNNs also allow the use of multi-dimensional arrays as an input and are therefore well suited for image classification tasks [40].

Parameters of convolutional layers are made up of learnable kernels, which are smaller in width D_{kx} and height D_{ky} than its input, D_{ix} and D_{iy} respectively, while using the full depth M of the input to that layer. Each kernel produces a 2-dimensional feature map by convolving across the width and height of the input. Output of the convolution layer is achieved by stacking together N feature maps generated by N kernels, as shown in figure 4. The number of these kernels, sometimes also called number of filters, is an important hyperparameter of convolutional layer [40].

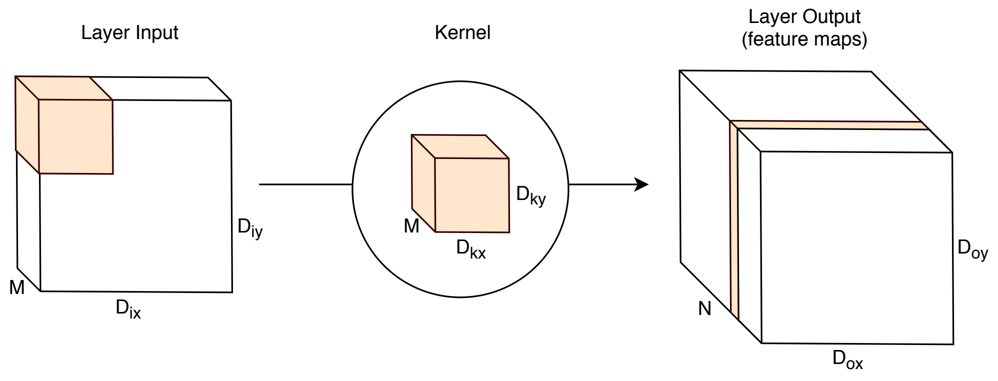


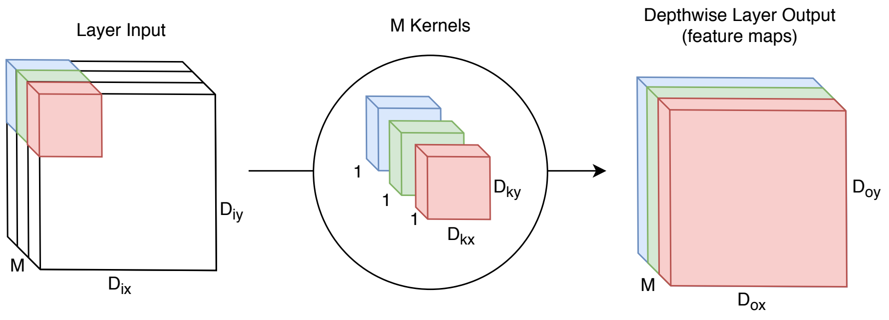
Figure 4. Kernel producing a feature map from input in a CNN layer.

Pooling layers are often used after convolutional layers. The goal of a pooling layers is to reduce the number of parameters by reducing the spatial size of the input layer, resulting in reducing the amount of needed memory and number of computations. While convolutional layer used full depth of the input to calculate the feature map, the pooling layer works independently on each feature map layer. Pooling layer also uses rectangular kernel to stride over the input layers, and finds one number as an output per stride. The result of a pooling layer has same depth (number of feature maps) but has decreased width and height [40].

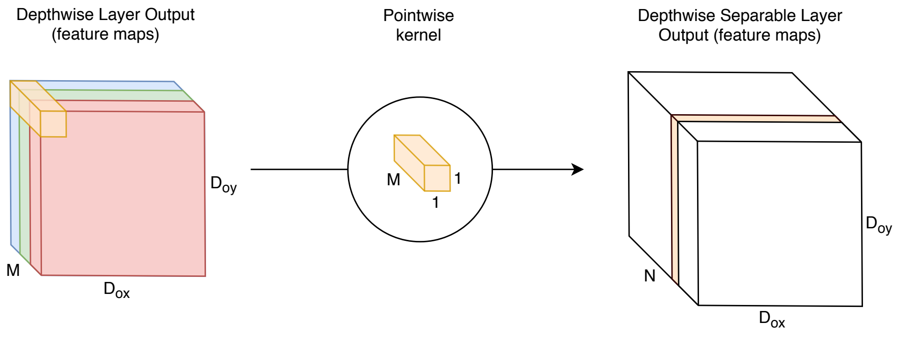
The last layers of CNNs are usually fully connected layers. While convolutional layers work as finding features from input data, the fully connected layer(s) perform the actual classification. The data from convolutional layer must be reshaped into a one dimensional vector before giving it as an input to the fully connected layer [40].

Depthwise Separable Convolutional Neural Networks

DS-CNN was introduced to make CNNs more efficient. DS-CNN divides regular convolution layer into depthwise and pointwise convolutions, which together form a depthwise separable convolution layer. This allows to reduce number of learnable parameters and therefore reduces also the number of required computations [53]. The depthwise and pointwise convolutions are illustrated on figure 5.



(a) Depthwise convolution, where input depth $M = 3$.


Depthwise Layer Output (feature maps)
Pointwise kernel
Depthwise Separable Layer Output (feature maps)
D_{oy}
M
D_{ox}
M
1
1
D_{oy}
N
D_{ox}

(b) Pointwise convolution, where input depth $M = 3$.

Figure 5. Depthwise separable convolution layer consisting of depthwise (a) and pointwise (b) convolutional layers.

Depthwise convolution uses one kernel for each of the M depth layers of the input, producing a feature map which has the same depth M as input. This is followed by pointwise convolution, where the feature map from the depthwise convolution is convolved with 1x1 kernels. These kernels have the same depth M as the input feature map. The height D_{oy} and width D_{ox} of the output of pointwise convolution is defined by the height and width of its input, while depth N is defined by the number of used pointwise kernels [53].

CNN and DS-CNN layers have same input and output dimensions, allowing to easily replace one type of layer with another. The efficiency win achieved by replacing classical convolution with depthwise separable convolution is dependent on the number of kernels N, the kernel width D_{kx} and kernel height D_{ky}. Depthwise separable convolution layer uses approximately $\frac{1}{N} + \frac{1}{D_{kx} \times D_{ky}}$ times the memory and computations required by a classical convolution layer [6].

2.2.4 Evaluation Metrics

Understanding how well the trained machine learning model is working requires evaluating it. For different tasks and use cases, different types of evaluation metrics should be used. Accuracy is used throughout this work for determining the classification power of an ANN.

In machine learning, accuracy is a performance measure defined as the ratio between the number of correctly classified samples and the total number of samples. Accuracy is not a good metric for problems with unbalanced data, where most of the samples belong to one class. This can lead to a situation called accuracy paradox where all the classifications done by a model belong to one class, while still showing good accuracy. When dealing with unbalanced data, other metrics, such as F1 score could be used which is the harmonic mean of precision and recall [54].

2.2.5 Hyperparameter Optimization

Hyperparameters are the parameters which affect the learning process of machine learning algorithms and which are not learned during the training process [55]. These hyperparameters can include things like the structure of the ANN, its learning rate, optimization algorithm and batch size. Finding suitable hyperparameters can be done manually, based on the experience and knowledge of the experimenter, but this requires lots of trials and errors. To automate this process, several algorithms can be used. Grid and random search are simple algorithms, which allow to test out different hyperparameters from a user-defined search space.

Grid search and random search both suffer from the disadvantage of not taking into account the results of the previous iterations. As training procedure usually takes relatively long time to run, this wastes computing resources and time. Bayesian model-based algorithms, such as Tree of Parzen Estimators (TPE), can be used to find better hyperparameters faster. Bayesian hyperparameter optimization builds probability models of the objective function (usually evaluation error), which are the basis for choosing next set of hyperparameters for training and validation procedure. After each experiment, Bayesian probability models are updated based on the used hyperparameters and the resulting value of the objective function [56].

2.2.6 Quantization

Deep Learning frameworks mostly deal with 32-bit floating point number format, which offer good accuracy. The drawback of this approach is the memory usage and computational complexity when dealing with floating point numbers. This is especially important when running ANNs on embedded devices. The memory usage and computational complexity can be reduced by converting 32-bit floating point numbers to fixed point numbers with lower bit depth. This process is called quantization. It has been shown that quantization can be successfully used to run neural networks on embedded devices [57]. There are various ways to perform quantization. This section concentrates on how it is done in Tensorflow and CMSIS-NN, as these libraries are used throughout this work.

Fixed point numbers allow to represent real numbers, with limited resolution, while storing them as integers in the memory. Q-notation is often used to describe the schema how these integers can be translated to real values or the opposite way around. In this work we will use the notation, as it is used in CMSIS library [58], where Qx.y specifies that x bits will be used for integer part of the number (also includes sign bit) and y is used for decimal part. For example, Q4.4 shows that one bit will be used for sign, three bits for integer part and four bits for decimal part of the real number. The number is stored in memory as an 8-bit integer, having values between -128 and 127. Q4.4 format allows the possible values to range between -8 and 7.9375, while having the resolution of 0.0625. In general, the value ranges for Q notation can be found to be $[-2^{m-1}, 2^{m-1} - 2^{-n}]$, while resolution is 2^{-n}. It can be seen that the value ranges are power of twos [59].

One of the possible methods of quantization is called post-training quantization. This method is simple to implement, as it does not need modifications to the training process. Post-training quantization can be divided into two parts – quantization of learnable parameters (weights and biases) and quantization of activations. Quantizing weights and biases is straightforward, as these parameters are fixed after training and suitable quantization range can easily be found with the equations shown in previous paragraph [6].

Quantization of activations, however, is dependent on the input data, as seen on figure 6. It is necessary to use some representative set of data to get the possible value range for activations. In [60], it is stated that about 100 mini-batches of data would be needed for this, without specifying batch size. In Tensorflow, post quantization is done by fake quantization operation[1] which is introduced after adding biases to the results of the convolution operation. Input arguments to the fake quantization operation include the range of values and the number of bits to which the values should be quantized. It then simulates the fixed point numbers and the bit-depth, by rounding the values based on the resolution from the bit-depth, and clipping the values to the allowed range. Post quantization involves testing fake quantization operations with different value ranges, starting from the ANN's input layer, to see which value ranges yield the best accuracy. This allows to find suitable ranges, and therefore quantization criteria and suitable Q format for each activation layer [60].

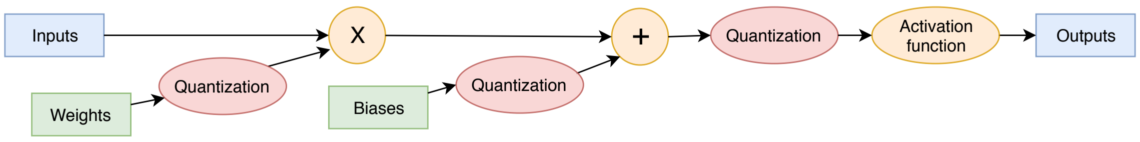


Figure 6. Quantization of weights, biases and activations in an ANN layer.

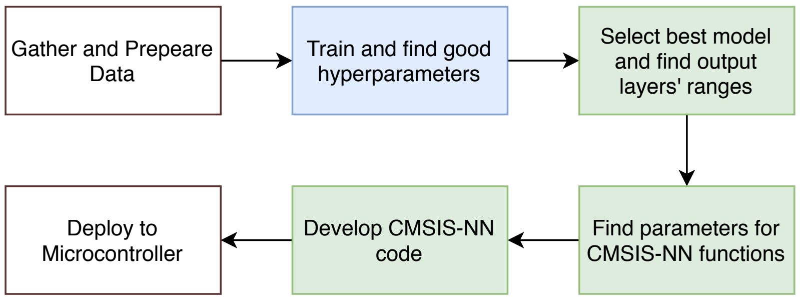
Activations quantization is needed to avoid de-quantization of weights and operations with floating points between layers [59]. The flow of quantized data through a convolutional layer, illustrated on figure 6, helps to explain this concept. It can be seen that layer inputs (quantized in previous layer) are multiplied with quantized weights during the convolutions, which in CMSIS-NN library yields a number with two times higher bit-depth [59]. For example, if input and weights are stored as 8-bit fixed point numbers, the output of their multiplication will be 16-bit fixed point number. This is required to keep the accuracy while multiplying the two numbers. As biases are added to the product of inputs and weights, these also have to be transformed into the same Q format. This requires changing the number of bits reserved for integer and decimal parts and is one of the parameters required by CMSIS-NN library [59]. This 16-bit number will then be converted back to 8-bit format, which again requires converting the number format and is a parameter required by CMSIS-NN functions. The format that it needs to be converted to is determined by the best range found with fake quantization operation described in previous paragraph.

[1]`https://www.tensorflow.org/api_docs/python/tf/quantization/fake_quant_with_min_max_vars`

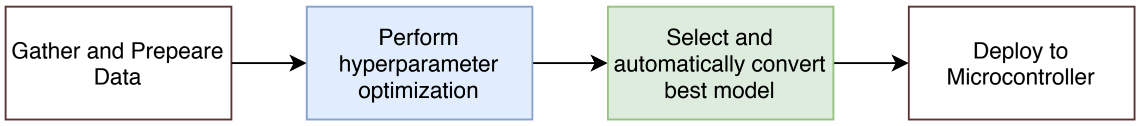
3 Framework for Creating ANNs for Microcontrollers

Running Artificial Neural Networks (ANNs) on microcontrollers is challenging in many aspects. Training a suitable ANN for microcontrollers requires an approach which takes into account the limitations of embedded systems. The training is followed by converting the learnable parameters into correct format and developing the code for microcontrollers. The process can be time consuming, prone to errors and requires knowledge about ANNs and embedded systems development. The goal of this section is to describe a framework which simplifies and speeds up the process of training an efficient ANN for audio classification task and automates the process of converting the trained model into a suitable format for Cortex-M microcontroller.

ARM has a project called Hello Edge [6] for training ANNs for audio classification tasks, which includes tutorial and description of the process, together with code examples, for running the trained models on Cortex-M microcontrollers. Hello Edge work simplifies the process of developing ANNs for microcontrollers, but does not automate or describe in enough detail some parts of the process. The work in this section describes, investigates and improves the process and framework used in Hello Edge project. Improved framework can be seen in figure 7b, while original Hello Edge process is described on figure 7a.


Gather and Prepeare Data
Train and find good hyperparameters
Select best model and find output layers' ranges
Find parameters for CMSIS-NN functions
Develop CMSIS-NN code
Deploy to Microcontroller

(a) Hello Edge workflow.


Gather and Prepeare Data
Perform hyperparameter optimization
Select and automatically convert best model
Deploy to Microcontroller

(b) Improved workflow.

Figure 7. Comparison of original (a) and improved (b) worklow for creating ANNs for microcontrollers.

The main changes introduced by this thesis include improvements in data preparation and preprocessing, introduction of hyperparameter optimization and automatic code creation from trained Tensorflow model, which can then run on Cortex-M microcontrollers. The introduced changes help to reduce development time and introduce ways which could improve accuracy of audio classification on embedded systems. The suitability of the Hello Edge project for being a basis of this work, together with frameworks and tools introduced by it, are analyzed in section 3.1.

The work is done by going through the whole process introduced by Hello Edge project and by taking bird sound classification task as an example for finding ways to improve the original work. A custom dataset is gathered and prepared for this, as described in section 3.2, with the goal of finding out whether the process introduced in Hello Edge work is also suitable for other audio classification domains.

Preprocessing and feature extraction is an important part in the process of developing ANN solutions and can have great impact on classification accuracy. As the data used in training is often not recorded by the embedded system, it might have different characteristics. The preprocessing phase should take this into account. Ways how Hello Edge's preprocessing and feature extraction could be improved are described in section 3.3.

Finding suitable hyperparameters in the training phase is one of the most time consuming parts of developing ANN solutions. Training phase usually takes several hours and search for good hyperparameters requires a lot of experimentation. This thesis introduces hyperparameter optimization as a way to partly automate this process, as described in section 3.4.

The training phase is followed by compressing learnable parameters of the trained model, finding the needed parameters of this model and developing code which would represent the same model on the microcontroller. In Hello Edge work, large part of this process needs to be done manually. The thesis introduces a way to automate the process of creating a model for microcontroller, which corresponds to the one trained on high performance computer. The automated process is described in section 3.5.

Previously mentioned improvements to the framework concentrated on developing an ANN solution on high performance computer and converting it to a model for microcontroller. To get this model working on microcontroller, audio retrieval, preprocessing and feature extraction also need to be efficiently implemented on the microcontroller. Hello

Edge project includes examples for implementing these parts on Cortex-M microcontrollers. The original work and changes introduced by this thesis are described in section 3.6.

3.1 Libraries and Tools

Libraries, frameworks and tools used in this work are described and analyzed in this section. The reason of taking Hello Edge project as a basis for this thesis is explained, together with the description of this project. This also includes describing and analyzing the suitability of CMSIS-NN and Tensorflow frameworks, used by the Hello Edge project, for developing ANN solutions for microcontrollers. In addition, other libraries used throughout the work are analyzed and described.

Hello Edge - Keyword Spotting on Microcontrollers

Hello Edge - Keyword Spotting on Microcontrollers [6] is a research paper, tutorial and example code by Arm about running ANNs on Arm Cortex-M microcontrollers for Keyword Spotting (KWS) task. KWS task involves recognizing short keywords, such as "stop" or "open". Hello Edge paper looks at different ANN architectures often used for KWS tasks, by exploring and evaluating these architectures from the perspective of running the ANNs on microcontrollers. Mel-Frequency Cepstral Coefficients (MFCCs) are used as features for these ANNs.

Hello Edge work included training various ANN structures with different complexities. The accuracy, memory and computational requirements of the trained ANNs were then compared and divided into three groups for microcontrollers with different capabilities. The article mainly concentrated on comparing Depthwise Separable Convolutional Neural Network (DS-CNN) with other types of ANNs, while also analyzing quantization's impact on classification accuracy.

The paper was accompanied by code[1] for preprocessing audio files, training ANNs and also examples of how some of these ANNs can be run Cortex-M microcontrollers. The work used Tensorflow for training the networks and CMSIS-NN to run the trained models on a Cortex-M microcontroller. The work also introduced code examples for quantization

[1]`https://github.com/ARM-software/ML-KWS-for-MCU`

of Tensorflow models and a way to find suitable parameters for CMSIS-NN functions. Hello Edge work is one of the few resources available for running ANNs on Cortex-M microcontrollers. The work is referenced and brought as an example throughout different ARM white papers [61], [62], community discussion boards and Github issues related to CMSIS-NN or generally running ANNs on microcontrollers. Therefore, the project was taken as a basis for this thesis. It should be also noted that there is an ARM NN SDK [1] which allows to easily transform ANNs to a code for ARM processors, but this tool concentrates on mobile devices and high performance embedded system, while not supporting Cortex-M microcontrollers.

CMSIS-NN

Cortex Microcontroller Software Interface Standard (CMSIS)[2] is a hardware abstraction layer for Cortex-M processors, enabling simple interfaces to the processor and other peripherals. CMSIS consists of different components, such as CMSIS-DSP and CMSIS-CORE. CMSIS-DSP[3] is a library for digital signal processing tasks and it includes over 60 functions for various data types, which are optimized for the Single Instruction, Multiple Data (SIMD) instruction set. CMSIS-CORE[4] is a component offering standardized API for the Cortex-M processor core and peripherals.

CMSIS-NN [59] is a component of a CMSIS and it is a collection of ANN kernels designed for Cortex-M processor cores to maximize performance and minimize the memory usage. It includes fully connected and convolutional ANN functions, while also providing functions for activations and pooling. CMSIS-NN also includes functions for depthwise separable and pointwise convolutions, allowing to easily create DS-CNN models. Functions in CMSIS-NN use either *q7_t* or *q15_t* data types, meaning that network weights and biases should be quantized to 8-bit or 16-bit integers. This also requires quantization of activations, as the functions require bias left shift and output right shift arguments as inputs.

There are few alternatives for CMSIS-NN, such as uNeural[5] and uTensor [6]. Both of

[1]https://developer.arm.com/ip-products/processors/machine-learning/arm-nn
[2]http://www.keil.com/pack/doc/CMSIS/General/html/index.html
[3]http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html
[4]http://www.keil.com/pack/doc/CMSIS/Core/html/index.html
[5]https://github.com/Jeff-Ciesielski/libuneural
[6]https://github.com/uTensor/uTensor

these libraries are currently limited to fully connected ANNs. While uNeural seems to have stopped development, uTensor is still under active development, promising support for convolutional layers and integration with CMSIS-NN in future. Due to limited functionality of the alternative libraries and the ARM supported development and provided ecosystem for CMSIS-NN, it is most suitable for the purposes of this work. The usage of uTensor should be considered in future works, when the promised features are rolled out.

Tensorflow

TensorFlow [63] is an open source platform for machine learning developed by Google. While it is possible to run other operations and machine learning algorithms with Tensorflow, it is mainly used for deep learning research and solutions. A comparison of TensorFlow and other popular deep learning libraries is given in table 1. It should be noted that Keras [64], another highly popular deep learning framework, uses TensorFlow as one of its possible backends. In the upcoming release of TensorFlow 2.0, Keras will be included into the TensorFlow library.

Table 1. Comparison of deep learning frameworks in Python based on GitHub statistics describing the activity and popularity of projects.

Framework	**Stars**	**Forks**	**Watchers**	**Contributors**
Tensorflow [65]	126 039	79 983	8 602	1 953
Keras [66]	40 400	15 342	2 004	790
Caffee [67]	27 864	16 789	2 243	266
PyTorch [68]	27 151	6 469	1 235	1 014
MXNET [69]	16 730	5 959	1 175	682
CNTK [70]	16 056	4 249	1 385	199

Based on comparison in table 1, it can be stated that Tensorflow is one of the most popular deep learning libraries. The popularity, flexibility and functionality of Tensorflow makes it a good framework for building ANNs. It is therefore used in this work for training and evaluating ANNs.

Other Used Libraries

There are several hyperparameter optimization libraries available, which support Bayesian optimization, such as Sigopt[1], Scikit-Optimize[2], Bayesian Optimization[3] and Hyperopt [71]. Sigopt is a service with good features and easy to use interface, but it is not open source and requires subscription. Scikit-Optimize has good features and is easy to use, but is still under heavy development. Bayesian Optimization is well developed, but subjectively harder to use than other libraries. Hyperopt is easy to use, supports flexible search spaces definition and is well developed, while development of new methods and features are still under a way. Based on Github statistics it's the most popular library of the discussed ones. For these reasons, it is chosen as a library for performing hyperparameter optimization in this work. Hyperopt library allows to optimize hyperparameters with search spaces that include real-valued, discrete and conditional dimensions. The library has currently implementations for random search and Tree of Parzen Estimators (TPE) algorithms.

Only two Python libraries were found which supported extracting Gammatone Frequency Cepstral Coefficients (GFCCs). Speech Feature Extractor[4] library is a small project offering different audio feature extraction methods, including GFCCs. Essentia [72] is a much larger and well developed open-source C++ library, which has Python interface, for audio-based music information retrieval. As Essentia is a larger project, offers better quality code and is more popular, it was used in this work to extract GFCCs. The code for extracting GFCCs was also used to implement a solution which can run on microcontrollers.

3.2 Gathering and Preparing the Dataset

Data gathering and preparing it for training phase involves downloading the data, converting downloaded recordings to wave files, splitting the recordings to smaller clips and dividing the data into training and testing set. The section describes different datasets, how these were combined and prepared, and how validation and test sets were created.

[1]`https://sigopt.com/`

[2]`https://github.com/scikit-optimize/scikit-optimize`

[3]`https://github.com/fmfn/BayesianOptimization`

[4]`https://github.com/ZhihaoDU/speech_feature_extractor`

3.2.1 Used Datasets

The dataset used in this work is a combination of five different data sources. The choice of bird sounds to be downloaded from Xeno-canto database was based on the work done in author's previous thesis [16] and included sounds from 20 most popular song birds in Estonia. Other datasets were prepared by various authors for scientific research. The idea was to combine bird sounds and environmental background sound into a single dataset, so it could be used to train a classifier which could classify certain birds in real-life environment. The overview of used datasets is shown in table 2.

Table 2. Different data sources used in creating a dataset for training an ANN.

Database / Dataset	**File Format**	**Clip Length (s)**	**Number of Recordings**	**Hours of Audio**
Xeno-Canto: Parus Major	MP3s with various sample rate and bit depth	Various	2 361	35.8
Xeno-Canto: other birds	MP3s with various sample rate and bit depth	Various	12 347	323.5
TUT	WAVs, 44.1 kHz, 24-bit	10	4 680	13.0
ESC-50	WAVs, 44.1 kHz, 16-bit	5	2 000	2.8
Urbansound 8K	WAVs with various sample rate and bit depth	≤ 4	8 732	8.78
Urbansound-SED	WAVs, 44.1 kHz, 16-bit	10	10 000	27.8

Xeno-Canto

Xeno-canto [1] is a crowd-sourced database for bird sounds. Everyone can upload their audio recordings of birds to this site. Xeno-canto database contains a lot of meta-data about the recordings, such as species in the foreground and background, location, time of recording and quality. It also offers Application Programming Interface (API) which enables to download specific recordings and relevant meta-data. Xeno-canto data has various quality, recording duration and background noise.

[1]`https://www.xeno-canto.org`

TUT Acoustic Scenes 2017

TUT Acoustic Scenes 2017 [73] dataset consists of recordings from 15 acoustic scenes: bus, cafe, car, city center, forest path, grocery store, home, lakeside beach, library, metro station, office, residential area, train, tram and urban park. From each scene 3-5 minute long audio recordings were captured and split into 10 second clips. This results in having 312 24-bit wave files with 44100 Hz sampling rate for each scene.

ESC-50

The ESC-50 [74] dataset consists of 2000 environmental audio recordings manually gathered from `freesound.org` database. It has 50 different semantical classes, each having 40 five second long recordings. These 50 classes are divided into five major categories: animals, natural soundscapes and water sounds, non-speech human sounds, interior/domestic sounds and exterior/urban sounds. It should be noted than one of the 50 classes contains bird sounds, which were left out of the dataset. The data is in 16-bit wave files at 44 100 Hz sampling frequency.

Urbansound 8K

Urbansound 8K [75] dataset consists of 8732 recordings of urban sounds, manually gathered from the `freesound.org` database. These recordings are divided into 10 classes: air conditioner, car horn, children playing, dog barking, drilling, engines idling, gun shots, jackhammer, siren and street music. The lenght of the recordings varies, but is less or equal to four seconds. Recordings are in wave format, but have same sampling rate, bit depth and number of channels as in `freesound.org`, meaning that these attributes will vary between different recordings.

Urbansound-SED

Urbansound-SED [76] is a dataset synthesized from Ubransound 8K dataset. This dataset has the same 10 classes, as in Urbansound 8K, but it consists of 10 000 augmented recordings. Each recording is 10 seconds long and has Brownian noise added as a background to resemble the typical hum heard in urban environments. In addition, some other techniques were used to modify original files in Urbansound 8K dataset, such as time stretching and pitch shifting. The files come in single channel 44 100 Hz, 16-bit wave format.

3.2.2 Data Preparation

The downloaded data is in different formats and therefore all the recordings are converted to wave files with same settings. The sampling frequency of 16 kHz is used for the converted wave files. This allows to analyze sounds up to 8 kHz, as proposed by Nyquist–Shannon sampling theorem [77], which covers the range of Great Tit song as illustrated in section 2.1.1 and on figure 2. The Analog-to-Digital Converter (ADC) used on the embedded system has resolution of 12 bits and therefore 16 bit Pulse-Code Modulation (PCM) was used for the converted wave files.

The converted wave files were then split into shorter clips, which represent the samples for training and testing the ANN. 20 percent of recordings were randomly chosen to be in the testing set. The clips were then balanced in a way that one third of samples would belong to *Parus Major*, one third to other birds and one third to environmental sounds. The balancing involved removing some of the clips from other bird sounds, in a way that resulted of having equal number of samples for *Parus Major* and other birds. During the training, it is specified how much recordings are taken for each training batch from *unknown* class, relative to the *Parus Major* glass, In our case, *unknown* class consists of other birds and environmental sound recordings. This approach allows to use more recordings from *unknown* class during the training phase, while giving same number of *Parus Major* and *unknown* class samples each training batch. The results of splitting recordings into shorter clips for training set can be seen in table 3.

Table 3. Created training set.

Original Dataset	**Number of Recordings**	**Hours**
Parus Major	78221	43.5
Other Birds	78221	43.5
TUT	29952	16.6
ESC-50	4119	2.3
Urbansound	5987	3.3
Urbansound-SED	32000	17.8

Embedded system restricts the maximum length of clips due to the limited memory. Training an ANN requires having clips with the same length as it can be retrieved on the embedded system. The length of each clip was set to be two seconds, as the experiments on the embedded system showed it would be the upper limit which could be used on that device. Different methods for splitting the recordings were applied for different data sources. Great Tit recordings were split by using a filter with sliding window of two seconds, which measured the energy of higher frequencies and compared it to energy from lower ones. If the ratio of higher frequency energy and lower frequency energy was large enough, the audio from that window was saved as an audio clip. This was done to improve the chances that silent part of the audio was not included in the Great Tit samples. For other datasets this was not important, as silence is also part of the *unknown* class and therefore data was split without the filter. Sliding window was also used with different step sizes, where the step size was bigger for the datasets containing more audio.

3.2.3 Recording Audio Through Embedded Device

The noise introduced by the embedded system might affect classification accuracy of the ANN. The peculiarities of the audio recorded by the embedded system should be taken into account when training the ANN. For this, the audio could be recorded through the embedded system. Recording audio clips in this manner allows to develop new ideas and solutions on high performance computers, without the need of testing each solution out on a microcontroller.

Recording sound through embedded system involved collecting ADC readings of the microphone through serial interface. The mean of the ADC readings was found, the signal was centered around this mean and converted to 16-bit PCM format, which could then be saved as a wave file. The wave file had same specifications as rest of the dataset. The recordings were performed indoors, while having different types of noises present, such as passing airplanes. As these types of noises can also occur in natural environment and were not very loud, having these noises was not considered as a problem. This is illustrated on figure 8.

Noise introduced by the embedded system might have an impact on the classification accuracy of the trained model. To counter this, the noise specific to the embedded system, can be added to the training samples. The background noise of the indoors environment was recorded for this purpose through embedded device. These recordings were then randomly added to the audio clips during the training phase, as described in section 3.3.

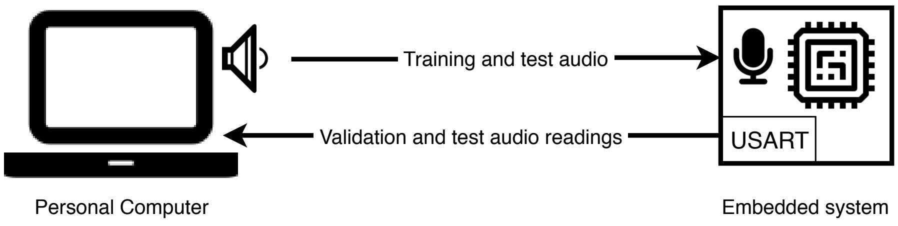


Figure 8. Validation and test set gathering through the embedded system.

Validation and testing sets were created in a similar manner. The key difference was that sounds were played through the speakers. The same script was used for playing audio and retrieving readings from the embedded device, as illustrated in figure 8. As the audio recording on embedded device and playing audio from regular computer were not synced, the same audio clip was played several times. This made sure that relevant part of the audio played through the speakers will be recorded through embedded device. The readings from ADC where discarded, if they contained parts where no audio was being played. With Great Tit song, a Root Mean Square (RMS) level was also measured and clips with low RMS were discarded. The resulting clips are characterized in table 4.

Table 4. The test set recorded through the embedded device.

Dataset Name	Number of Recordings	Hours
Parus Major	5 702	3.2
Other Birds	4 395	2.4
TUT	1 096	0.6
ESC-50	538	0.3
Urbansound	591	0.3
Urbansound-SED	2 170	1.2

The speakers can add additional noise to the signal, which is not present in natural environment. This should be considered, when researching how well the device would work in real-life. As this is out of the scope of this thesis, the characteristic introduced by the speakers can be considered as a part of the noise of the embedded system.

3.3 Preprocessing and Feature Extraction

Hello Edge [6] code offers a very good approach for preprocessing the data and is used as a basis of the work. However, there were some changes introduced by this work for improving the noise robustness of the ANNs solutions.

Original Hello Edge Implementation

The preprocessing of audio clips occurs when the data is requested during the training, validation or testing phase. Preprocessor allows to divide data into training, validation and testing set, which percentages are determined by input arguments. These different subsets are randomly chosen and determined by file name hashes, so the files will remain in the same subsets, even after adding new audio files. Preprocessor returns a batch of processed data and its labels for the needed subset, as requested by the training, testing or validation program. Preprocessing of testing and validation data differs from preprocessing of training data, as in the latter case the data is augmented by background noise and time shifts.

The preprocessor starts by loading the wave files and scaling these to have values between -1 and 1. If the data is requested from training phase, the samples will be randomly chosen from training set for each request. In training mode, the audio samples will be time shifted and background noise will be added. This process is called data augmentation, and it creates more training samples from existing ones to reduce overfitting and improve the generalization power of machine learning models. Time shifting moves the starting point of the audio clip and pads the signal in case it is not long enough. As training samples are used several times in training phase, this allows to introduce small changes in the used samples, which should help to improve classification accuracy.

Another way to augment data is by adding background noise, which takes a random part of an random audio clip from a specific folder and adds it to the training sample. If some elements of the sum of original signal and background signal end up being larger than 1 or less than -1, these elements will be clipped to 1 or -1. Frequency of training samples, which should include added background noise, and the maximum range of sound volume for background noise are input arguments of the preprocessor. The addition of background noise allows to use different background sounds for same samples, forcing the ANN to find the important properties of the signal.

Data loading and augmentation is followed by creating a spectrogram and extracting MFCCs from it. Both of these function are included in the Tensroflow library. Spectrogram takes window size and window stride as input arguments. These arguments define the number of time and frequency bins of the spectrogram. Function for MFCCs takes the number of returned MFCC coefficients as an input argument. The number of MFCC coefficients determines how many features there are for each time bin, while the number of time bins for MFCCs is equal to the number of spectrogram time bins. The MFCCs are then returned together with the ground truth labels to the training, testing or validation program.

Modified Implementation

The code from Hello Edge work offered good preprocessing methods, but it needed some modifications so it could be integrated to the proposed framework. The original code was not modular enough for the purposes of this work and therefore could not be used as an external library.

Original code included the possibility to add background noise to the training samples, but the sound volume of the added background noise was randomly chosen within some given range. The code was modified to allow adding the original volume of the background noise, as similar noise levels could be expected from the embedded system.

Preprocessing for Great Tit classification does not need to consider lower frequencies, as was described in section 2.1.1 and on figure 2. By dismissing lower frequencies, we can also filter out big portion of the noise, which could improve classification accuracy and also makes preprocessing more efficient. The suitable frequency range could be chosen when extracting MFCCs. This possibility is implemented in Tensorflow library, but Hello Edge code needed some modifications, so it would be possible to add these arguments as inputs for the preprocessor.

MFCCs are not very robust under noisy conditions, as described in section 2.2.2. Therefore, GFCCs were introduced, which perform better in noisy conditions. Essentia[1] library offers extraction of GFCCs and was used in this work. The number of GFCCs and number of filterbank filters used in the process were used as input arguments for the preprocessors. Lowering these numbers will reduce the required memory and number of computations.

[1]https://essentia.upf.edu/documentation/

Using GFCCs required some additional changes. The created coefficients had a lot of variation in them, having different ranges and mean values. For example, the values for the first coefficients of GFCCs were about 200 times larger than the values of last coefficients and not centered around zero. The quantization of these different values would result in loosing a lot of important information. Therefore, the minimum and maximum values were found for each coefficient over the training and validation data. These values were then used to scale the output of each coefficient, so all of the values would be varying within the same range. The comparison of MFCCs and GFCCs for the dataset used in this work is analyzed in section 4.3.

GFCCs require more preprocessing and memory than MFCCs, as each GFCC filter continuously covers all frequencies. To fit everything into the memory of the embedded system, the DCT matrix and GFCCs filter coefficient are created in Python and written into a C++ header file, as these matrices can be calculated without the need for input data. This approach allows to create constant variables, which will be loaded into read-only flash memory, instead of more valuable Random-Access Memory (RAM). Calculating DCT matrix and GFCCs filter coefficients required to rewrite original Essentia C++ code in Python.

3.4 Training and Hyperparameter Optimization

The original Hello Edge code for training a ANN is introduced in this section. Some minor modifications were introduced by the author to make the training phase more flexible and suitable for the wokflow proposed in this work. Hypeparameter optimization is introduced to simplify the process of finding suitable preprocessing parameters, ANN structure and training hyperparameters.

Original Implementation

The code for training ANNs from the Hello Edge work included defining the preprocessing parameters and hyperparameters for the ANN. These hyperparameters included the type of the ANN, its architecture and other parameters, such as how many training steps should be performed and which learning rates should be used. For each training step, a certain number of training data, specified by batch size parameter, is preprocessed and used for training the ANN. After a certain number of training steps, the accuracy of

ANN is found on validation set. Most accurate models throughout validation steps of the training phase are saved.

Modified Implementation

The original code allowed only to train by running the program from command line. To make it usable for hyperparameter optimization program some modifications were introduced. Command line arguments were replaced by training function's input arguments. To further improve the possibility of using hyperparameter optimization, a function was introduced which would parse all the parameters from a dictionary data type and translate them to input arguments for the training function.

Compared to the original training set, where noise from embedded system is added to the samples, the proposed validation set consists of recordings retrieved through embedded system. The data in validation set has better resemblance to the data expected during real life classification and gives better estimation of the classification accuracy on the end device. The original Hello Edge code required small modifications to use custom validation data. As it is good practice to use testing data only for the final validation, the evaluation on testing set, which occurred after the last training step, was removed from the training function. This allows to get final test accuracies separately, after we've chosen our best performing models, to avoid overfitting to test data during model selection.

Training an ANN requires a lot of experimentation. To automate this process, Bayesian optimization is used to narrow down the ranges for suitable preprocessing and ANN hyperparameters. The range for each hyperparameter can be defined and this forms a search space. The process of extracting a set of hyperparameters from the search space and training an ANN on these hyperparameters is done iteratively. Each iteration involves training an ANN and returning the best validation accuracy to the Bayesian optimization algorithm. Based on this feedback, a new set of hyperparameters are chosen and the process is repeated. Trained models are saved, together with their hyperparameters, complexity and classification accuracy, allowing to match each hyperparameter optimization iteration to a corresponding trained model. The hyperparameter optimization process is illustrated on figure 9. Hyperopt library is used with TPE algorithm to perform Bayesian optimization.

Hyperparameter optimization is done in several rounds, where the search space is manu-

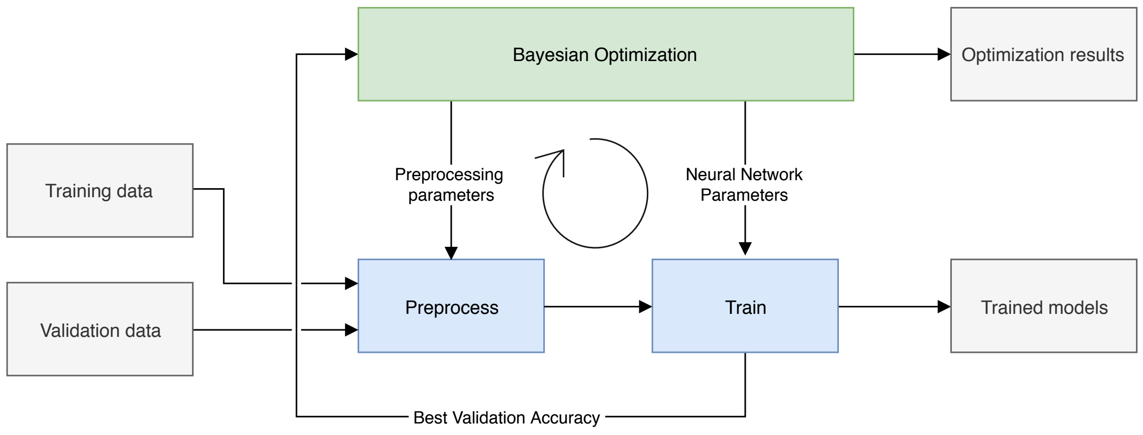


Figure 9. Implementation of hyperparameter optimization.

ally narrowed down each round. Changes to search space are introduced manually, based on new knowledge gained through previous rounds. In addition, experiments were performed with different trained models on the embedded system by finding the amount of CPU cycles it took to preprocess and classify an audio clip. The experiments gave additional knowledge how complex the model could be, so it could still run on embedded system in real time. Based on the gained knowledge, hyperparameter optimization search space and objective function was modified. Instead of only including the validation error in the objective function, another parameter was added, which penalized models with higher number of parameters. After performing hyperparameter optimization, a suitable model can be chosen based on it's complexity and classification accuracy.

3.5 Converting Tensorflow Model to CMSIS-NN Model

The largest contribution to improving the framework introduced by Hello Edge is automating the process of converting models trained in Tensorflow to a C++ code, which can then run on embedded device and which uses CMSIS-NN library. This approach will reduce development time and is less prone to human errors, which might occur when manually developing the code. The conversion process was partly done by Hello Edge work, but biggest part of this conversion required still manual work and calculations. In this section the original implementation is described together with the necessary modifications for automating the conversion process.

Original Implementation

The code from Hello Edge work introduced a way for reducing the resolution of learnable parameters, called quantization. Quantization is needed so the trained ANN could use less memory on the embedded system. Quantization of activations, weights and biases is needed. The quantization of weights and biases can be easily performed, as these values are known after training process. Quantizing activations requires a different approach, as it is dependent on the input data fed to an ANN. Quantization of activations involves loading a trained model to memory and introducing fake quantization layers for each ANN layer, including the input layer. It is then possible to test out different ranges (power of twos) for each of these fake quantization layers. Each fake quantization layer simulates the minimum and maximum values based on the range and its resolution. This is then tested on a subset of data to see how much the quantization changed the results. The original code also writes the quantized weights into a C++ header file which can then be deployed to the embedded system.

The Hello Edge process involves manually testing out different ranges for each layer of the ANN, including input layer. The work suggest starting from the first layer and finding the best range for that, by testing out which ranges (1, 2, 4, 8, 16, 32, 64, 128) give best accuracy on chosen set of data. When the range is found for the layer, the same should be done with the next one, until the ranges for all the layers are found. After the ranges are found, the bit shifts required by the CMSIS-NN framework, will have to be manually calculated. For a DS-CNN with four layers, this requires finding 10 ranges, which might need up to 80 runs of the script. In addition, creation of the corresponding ANN needs to be manually implemented in C++.

Modified Implementation

The modifications to the original Hello Edge code included introducing a way to fully automate model conversion and C++ code generation. The overview of the introduced framework is illustrated on figure 10.

Quantization of activations requires finding best activation ranges for each layer. This part was automated by introducing a simple for loop, that iterates over possible ranges, starting from the lowest possible. Each iteration, the range is increased and the effect of quantization within that range is measured on validation set. If the accuracy found is

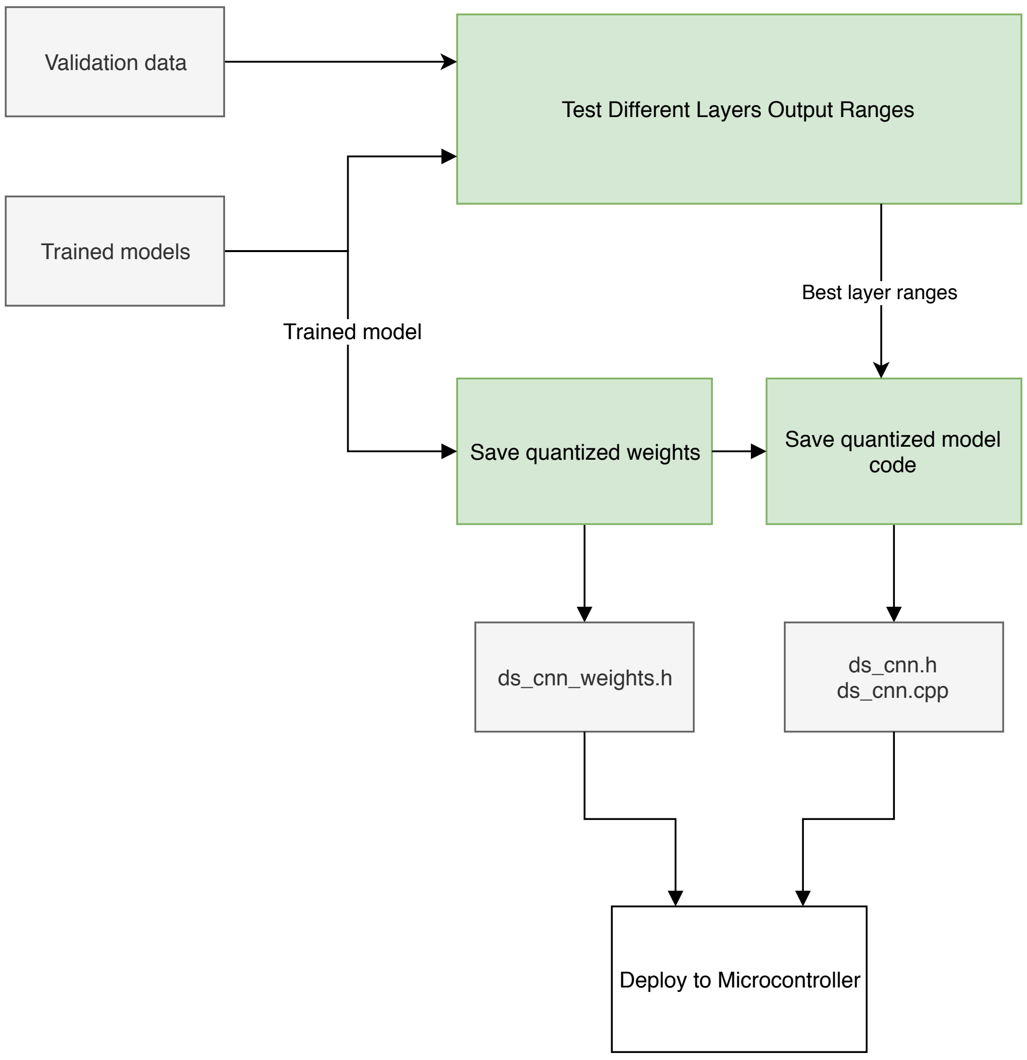


Figure 10. Process of converting Tensorflow model to CMSIS-NN model.

smaller than in previous iteration, it is concluded that the best range for that layer is the one from the previous iteration. The same process is then repeated with the next layer until the best ranges for each layer are found. The quantized weights, biases and activations define the quantized ANN model, which we can use on our testing set.

The CMSIS-NN library performs computations with 8-bit integers in Q-notation (described in section 2.2.6), so these integer values could represent real numbers. Based on quantization of activations, the suitable Q-notation was found for each layer. CMSIS-NN requires bias left shift and output right shift as inputs to neural network layers, so the Q-notation would match between arguments when performing multiplications and additions. The left and right shifts can be calculated from the ranges found while quantizing weights, biases and activations.

The hyperparameters used for training the ANN allow to automatically generate required C++ code (ds_cnn.cpp). The C++ file consists of sequential CMSIS-NN functions, which correspond to layers of the trained ANN. Certain parameters for these functions (including bias and output shifts) are held in ds_cnn.h file. This file is also automatically generated based on previous steps.

Original Hello Edge work required writing C++ source file manually, while also finding and calculating parameters for the header file (ds_cnn.h). The improvements introduced in this section allow to take the trained Tensorflow model and automatically convert it to a C++ code. The resulting code files can then be deployed to a Cortex-M microcontroller by simply copying the files to the project, compiling it and uploading compiled files to microcontroller.

3.6 Audio Classification on Cortex-M Microcontroller

The code for performing classification is generated automatically in the improved framework, but the embedded system also needs to read the audio signal and extract features from it. The section covers briefly the embedded systems used in this work and how the whole audio classification process is performed on that system.

Embedded system used in this work is produced by Thinnect and has Mighty Gecko EFR32MG12P432F1024GM48 microcontroller, which belongs to Cortex-M family. The microcontroller can run up to 38 MHz, has 1024 kB of flash memory and 256 kB of RAM. It has an 12-bit ADC, which can be used together with a microphone to retrieve audio. The microcontroller also has a Floating-Point Unit (FPU), which allows to efficiently do arithmetics with floating point numbers.

The process of audio classification includes ADC readings retrieval, normalization of the readings, feature extraction and classification, as illustrated on figure 11. Retrieving ADC readings and normalizing these was already implemented and provided by the supervisor of this thesis. Code for preprocessing and classification was taken from the Hello Edge [6] project. Integrating the audio retrieval with feature extraction and classification was done by the author of this thesis, as was the implementation of GFCCs feature extraction.

Audio retrieval was performed by storing ADC readings in two ping pong buffers, which

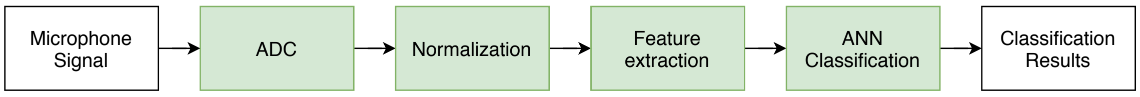


Figure 11. Audio classification process on the embedded system.

allow to preprocess the readings from one buffer, while the readings are recorded to the other. Both buffers save two seconds of data at 16 kHz sampling rate. This means that preprocessing and classification has to take less than two seconds, in order to work in real time. After one ping pong buffer is filled, the average of the ADC readings is found. This average is then subtracted from all the readings to center the values. The signal is then normalized to have values between -1 and 1.

The normalization of audio signal is followed by feature extraction. Two different feature extraction methods were used and analyzed. The extraction of MFCCs used the implementation provided by the Hello Edge work. The implementation of GFCCs was done by the author of this work. Developer can easily choose which feature extraction method is used, as the arguments, and input and output formats are the same for both feature extraction functions. Exploring audio filtering and noise reduction techniques is left out of the scope of this work.

The implementation of MFCCs extraction, provided by Hello Edge work, followed a similar process as described in section 2.2.2. The Short-Time Fourier Transform (STFT) was performed with the Fast Fourier Transform (FFT) function provided by CMSIS-DSP library. The starting and end frequency bins for each Mel filterbank filter was found, as the filters are triangular and have well defined start and end. This helped to save memory and computations, as there was no need to store and perform computations, where filter values were zero. After finding the MFCCs, the resulting floating point numbers were converted to *q7_t* format, based on the range found during quantization step for input layer.

GFCCs were implemented based on the Essentia GFCCs extraction and Hello Edge's MFCCs extraction code. GFCCs have continuous filters, instead of the triangular ones. This means that it requires more memory and computations, as it was necessary to store and compute with filter values, which have the same dimensions as output of the FFT. For example, having 10 filters, and FFT settings used in this work, would require storing and doing multiplications with $10 \times 513 = 5130$ floating point numbers. As filter coefficients and Discrete Cosine Transform (DCT) matrix could be calculated beforehand, when number of filters and FFT settings are known, these were defined as constant variables, so it could be loaded into larger read-only flash memory, instead of using RAM.

The features obtained from feature extraction phase are fed to an ANN for classification. Hello Edge work included code for fully connected neural network and DS-CNN models. This thesis uses only DS-CNN implementation. The implementation consists of using CMSIS-NN functions to create a identical model structure as the trained and quantized Tensorflow model. The layer and activation functions use two buffers which are being changed as inputs and outputs between the consecutive functions. The weights and biases for the DS-CNN are loaded to flash memory as constant variables from the header file, created during the quantization step. The concrete parameters for each layer, including each layer's dimensions, padding, stride, kernel size, bias left shifts and output right shifts, are also stored in a header file. The output of a classification is a vector of probabilities for each class in *q7_t* format.

Section 3 researched and improved the work of Hello Edge on the example of classifying bird sounds on a Cortex-M microcontroller. Libraries and tools used by Hello Edge work, and the ones introduced by this work, were analyzed to determine their suitability for the audio classification task. The work used a bird sound classification use-case, which required gathering data from different sources and consisted of bird sound and environmental sounds. GFCC feature extraction method was added to original work for improving noise robustness of the solution. This required implementing this feature extraction method for Cortex-M microcontrollers. The training process was improved by introducing hyperparameter optimization which allows to automate the process of finding hyperparameters for ANNs. The model trained in Tensorflow needs to be converted to a CMSIS-NN model, so it could be used on Cortex-M microcontroller. This process was automated, allowing to generate C++ code from a trained Tensorflow model. The implementation of this framework is in public domain and can be found on the link provided in appendix 1.

4 Method and Analysis

The main goal of this thesis work is to analyze the challenges of developing ANNs for Cortex-M microcontrollers and improve the development process. This involves training an ANN model on a regular computer and then converting the model to C++ code, suitable for running on Cortex-M microcontrollers.

Section 3 introduced the process of training and converting Tensorflow ANN model to CMSIS-NN model, while highlighting some of the changes and improvements needed for achieving better classification accuracy on an embedded system. In this section, the effects of these changes are analyzed. The analysis will be done on the example of classifying Great Tit sounds from other environmental sounds, including songs from other birds. This problem is complex enough for motivating the use of ANNs, while also having enough openly available training data. Although the research is done on bird sound classification, the results of this thesis can be applied, with slight modifications, to other domains related to sound and image classification.

Running an ANN model on microcontrollers requires combining several techniques for reducing complexity and computational footprint of the model. Some of these techniques are analyzed in this work with regards of how much computational and memory resources can be saved and how much it will affect the classification accuracy of the ANN model. Analyzing difference between using Convolutional Neural Network (CNN) and DS-CNN is done in Section 4.1. The effects of quantization are analyzed in Section 4.2.

The effect of preprocessing methods on the classification results will be also researched in this work. MFCCs and GFCCs are analyzed from the perspective of applying them on embedded systems. This involves comparing the computational requirements of extracting these features and robustness to noise introduced by the used embedded system. The difference in classification accuracy and resource usage is analyzed in Section 4.3.

Performing classifications on embedded systems often requires dealing with noise specific to that system. A solution of using background noise specific to the embedded system was introduced in this work. Section 4.4 describes the effects of adding background noise to the training samples.

The author's implementation of extracting GFCCs on embedded device requires verification that the results correspond to the ones used on regular computer. Similarly, the ANN model conversion from Tensorflow to CMSIS-NN framework, proposed in this work, should be validated. The analysis of how well the preprocessing and classification results match between the implementation on regular computer and embedded device is performed in Section 4.5.

The analysis is performed by using the hyperparameters found from hyperparameter optimization phase. The preprocessing and feature extraction phase transforms two second audio signal into features which have 61 time bins, where each time bin has five elements. The ANN consists of four convolutional layers and a fully connected layer. In case of DS-CNN, the first convolutional layer is the classical one, while rest of the three are depthwise separable convolutional layers. The CNN model has four classical convolutional layers, followed by a fully connected layer. In both cases, each convolutional layer creates 32 feature maps as outputs. The first convolutional layer has stride of five in x direction and stride of eight in y direction, while the kernel size is 5×5. Rest of the convolutional layers have kernel size of 3×3 and stride of one in both directions.

The accuracy is used as a metric throughout the analysis for describing the classification power of an ANN model. Accuracy is a simple and suitable metric for classification tasks with balanced datasets, as used in this work. However, using accuracy obtained from a single training run can be misleading in an analysis, where the effects of preprocessing and hyperparameters on classification accuracy is researched. This is because running several training runs with same hyperparameters and data, yields different models and classification accuracies, due to the random initialization of weights. Therefore, for each experiment the training should be repeated several times to be sure that the change in accuracy was caused by the change of hyperaparameters, not by the initial values of weights. The number of repeated training runs was limited to five, as each run can take up to a half day and the available computing time for this work was restricted. Confidence intervals are given for the average classification accuracy, to determine in what range the actual average accuracy could lie in. Confidence level of 95% is chosen for finding the confidence intervals. Selection of a confidence level is a subjective decision and 95% was chosen, as it is often used in research and lies between two other popular options – 90% and 99%.

4.1 Comparison of CNN and DS-CNN

Simplifications to ANNs make it possible to run deep learning models on embedded devices. One of those simplifications is using DS-CNN instead of CNN, as described in section 2.2.3. Analysis performed in this section concentrates on how much more efficient DS-CNN would be with regards of used memory and computations, while also analyzing how much classification accuracy is lost. In addition, it is compared whether the chosen architecture yields different results, when using different feature extraction techniques.

Method

Layers in classical CNN can be easily replaced with depthwise separable convolutional layers, without changing the output dimensions of layers. This makes it is easy to compare CNN and DS-CNN with regards of the used memory, number of computations and accuracy by training both network structures with same parameters. CNN and DS-CNN architectures are trained with MFCCs and GFCCs as features, with 10 filterbank filters. This will give insights on how dependent these architectures are on the features which are being used. Based on the five trainings for each of the preprocessing and model architecture set, we can find accuracy confidence intervals on the test set. If the accuracies overlap within the confidence interval it is not possible to state that one architecture has better performance than the other, as the difference might be due to the random initialization of weights.

The memory usage of the network can be estimated by the number of trainable parameters. In microcontroller, trainable parameters are stored in flash memory as 8-bit integers. Number of these parameters, however, also affects how much RAM memory is needed for buffers used in the classification process. The computational complexity can be measured in Floating Point Operations per Second (FLOPS). Tensorflow allows to easily find these values. The concrete steps to test the hypothesis can be seen below.

1. Train DS-CNN five times with MFCCs as features and five times with GFCCs as features
2. Train classical CNN, using the same hyperparameters five times with MFCCs as features and five times with GFCCs as features
3. Find accuracies for these models on testing set together with confidence intervals

4. Find number of trainable parameters for CNN and DS-CNN
5. Find number of FLOPS needed for giving a classification for CNN and DS-CNN
6. Compare accuracy, memory usage and FLOPS of trained CNN and DS-CNN models

Results

The results of the experiments are shown in table 5, which indicate that both architecture have approximately the same accuracy. CNN shows better performance with GFCCs as features, while it is slightly less accurate when MFCCs are used. With both features the difference is small and it can be said that the accuracy difference is not significant, as the confidence intervals overlap with each other. This means that we cannot be sure whether difference in accuracies is due to the difference of ANN architectures or because of the random initialization of weights.

Table 5. Comparison of CNN and DS-CNN models.

Model Architecture	**Trainable Parameters**	**FLOPS**	**Features**	**Accuracy**
CNN	28 770	5 306 438	MFCCs	74.86% ± 0.79%
			GFCCs	78.38% ± 0.51%
DS-CNN	5 250	904 844	MFCCs	75.29% ± 0.68%
			GFCCs	78.60% ± 0.76%

The memory usage and number of necessary computations for classification favors DS-CNN. It can be seen that memory-wise the DS-CNN analyzed in this work is over five times more efficient than CNN, while requiring almost six times less computations. The efficiency savings would be even larger with ANN structures having more layers.

Discussion and Conclusions

Compared with CNNs, DS-CNNs allow to create deeper ANNs under the same memory and computational restrictions. This allows to create ANNs with more classification power and therefore DS-CNN architecture should be preferred for embedded systems. When classification task is simpler, the DS-CNN could be made smaller and therefore faster. This allows to use overlapping sliding windows for classifications and increase the number of classifications made within a certain time period. The classifications could be then aggregated together for improving overall accuracy of the developed solution.

Results showed that DS-CNNs outperform CNNs with regards of memory usage and required computations, while the difference in accuracy was small. The experiments performed with GFCCs showed the possibility that the classification accuracy could even improve with DS-CNN architecture. Due to the small number of trained models, it was not possible to determine, whether this was caused by difference in network architecture or random initialization of weights. Current analysis could be improved by training more models for each of the four cases looked in this section.

4.2 Quantization Effects

Qunatization can impact classification accuracy, as the resolution of weights and biases is reduced. When using quantization, developers of ANNs need to analyze the effects of it, in order to determine whether the concrete method of quantization suits their task. Quantization's impact to classification accuracy can be analyzed by comparing accuracies of different trained models before and after quantization. The analysis in this section involves testing a hypothesis that quantization of weights and activations does not have significant impact on classification accuracy on the created bird sound dataset.

Method

The effects of quantization can be measured with different models and preprocessing techniques that are being used in this work. The accuracy for the trained models on test set can be found before and after quantization. Based on this, it is possible to validate how much the quantization affects classification accuracy. The change in classification accuracy is considered significant in this thesis, if the confidence intervals do not overlap for the accuracies calculated before and after quantization. Therefore, the definition of significant accuracy change in this work means that the difference in accuracies is due to quantization, not due to random initialization of weights. The concrete steps for testing the hypothesis are brought out below.

1. Train following models:
   - Five CNN models with MFCCs features having 10 filterbank filters
   - Five CNN models with GFCCs features having 10 filterbank filters

    - Five DS-CNN models with MFCCs features having 10 filterbank filters
    - Five DS-CNN models with MFCCs features having 10 filterbank filters and no added background noise to training samples
    - Five DS-CNN models with MFCCs features having 40 filterbank filters
    - Five DS-CNN models with GFCCs features having 10 filterbank filters
    - Five DS-CNN models with GFCCs features having 10 filterbank filters and no added background noise to training samples
    - Five DS-CNN models with MFCCs features having 40 filterbank filters
2. Get average accuracies found on test sets for these trained models with confidence intervals
3. Get average accuracies found on test sets for these models after quantization with confidence intervals
4. Compare the difference before and after quantization for different models.

Results

The results show that quantization tends to slightly decrease accuracy on the testing set, as shown in table 6. In general, the accuracy is decreased by less than one percentage point. In all the cases, the accuracy loss is not significant, as for all trained models the corresponding accuracy confidence intervals are overlapping. Therefore, it is not possible to say, whether changes in accuracy are due to quantization or randomness introduced by weight initialization.

Models using GFCCs tend to loose less accuracy due to quantization, than models which use MFCCs. MFCCs also show slightly higher variance in accuracies after quantization compared with GFCCs, while quantization also increased the variance for most of the models with GFCCs. DS-CNNs show slightly more decrease in accuracies after quantization than CNN. For most of the cases, the variance in classification accuracy increased after quantization.

Discussion and Conclusions

The experiments did not confirm that the decrease in accuracies was due to quantization. On the other hand, when looking at all eight experiments, each showed a small amount of accuracy decrease, suggesting that quantization might be the reason. In Hello Edge

Table 6. Effects of quantization on different ANNs and feature extraction methods.

Model Architecture	**Features**	**Filterbank Filters**	**Background Noise Added**	**Accuracy Before Quantization**	**Accuracy After Quantization**
CNN	MFCCs	10	True	74.86% ± 0.79%	73.81% ± 1.26%
	GFCCs	10	True	78.38% ± 0.51%	78.26% ± 0.48%
DS-CNN	MFCCs	10	True	75.29% ± 0.68%	72.66% ± 2.36%
		10	False	70.65% ± 1.25%	69.24% ± 1.62%
		40	True	71.96% ± 1.01%	70.05% ± 0.58%
	GFCCs	10	True	78.60% ± 0.76%	77.97% ± 0.78%
		10	False	78.01% ± 0.57%	77.25% ± 0.81%
		40	True	78.89% ± 1.08%	77.94% ± 1.90%

paper [6], the quantization showed slight increase in accuracies. Further research should be performed, whether the tendency of decreased accuracy, compared to the increase in Hello Edge work, is due to the specifics of used dataset and ANN architecture, differences between validation and testing set or small amount of experiments performed in Hello Edge paper.

Models with MFCCs showed more accuracy loss after quantization, compared to models with GFCCs. One of the reasons for this might be that GFCCs were scaled between -1 and 1. For MFCCs, the first coefficients might have larger values than last coefficients, as DCT is used for generating the outputs. This results in loosing more information due to linear quantization. There was also a small difference between variance of model accuracies before and after quantization. This could be due to increased randomness introduced by rounding the values, when decreasing the resolution of weights and biases. As MFCCs were not scaled, the introduced randomness is larger, resulting in larger variance after quantization, when compared with GFCCs.

The quantization showed a slight decrease in classification accuracy. At the same time, quantization allows to reduce memory usage up to four times, while also reducing computational requirements, as there is less overhead when doing computations with integers, instead of floating point numbers.The benefits from quantization outweight the possible reduction of accuracy, as reduced memory usage allow to use more complex models.

4.3 Comparison of MFCCs and GFCCs

Preprocessing methods have different properties with regards to computational cost, but also in regards of noise robustness. Affordable embedded systems might introduce noise, which should be taken into account when building ANN solutions. GFCCs were implemented in this work to see how it affects classification results under the introduced noise. The analysis is performed to determine how much classification accuracy is affected, when using GFCCs instead of MFCCs and how much the computational cost differs between these feature extraction methods. In addition, experiments are done to determine how does reducing the number of filterbank filters affect the classification results and computational complexity.

Method

A DS-CNN model is trained using GFCCs and a model with same hyperparameters is trained using MFCCs. With both feature extraction methods, the training was performed five times with 10 filterbank filters and five times with 40 filterbank filters. The accuracies for these trained models were then found on the test set.

The feature extraction methods were run on the embedded system, while measuring the number of Central Processing Unit (CPU) cycles it takes to extract features, in order to compare the difference in computations. The CPU cycles measured in embedded system through cycle counter are printed out through serial interface. To understand how much each additional filter adds computational cost, measurements were done with filters from 10 to 50, with a step of 10. The steps for finding the accuracies and CPU cycles are described below.

1. Train DS-CNN five times using GFCCs with 10 filterbank filters as features
2. Train DS-CNN five times using GFCCs with 40 filterbank filters as features
3. Train DS-CNN five times using MFCCs with 10 filterbank filters as features
4. Train DS-CNN five times using MFCCs with 40 filterbank filters as features
5. Find average accuracies with confidence intervals for the models on testing set
6. Run GFCCs extraction on embedded device with 10 to 50 filterbank filters (step of 10) and find the number of CPU cycles it takes
7. Run MFCCs extraction on embedded device with 10 and 50 filterbank filters (step of 10) and find the number of CPU cycles it takes

Results

The results, summarized in in table 7, show that using GFCCs, instead of MFCCs, as features yield significantly better accuracy on a noisy dataset. With 10 filterbank filters, a model using GFCCs, instead of MFCCs, shows around three percentage point improvement, while with 40 filterbank filters the accuracy improvement is around six percentage points.

Table 7. Comparison of classificiation accuracies when using GFCCs and MFCCs as features with the same DS-CNN structure.

Used Features	**Filterbank Filters**	**CPU Cycles**	**Accuracies**
GFCCs	10	45.44M	78.60% ± 0.76%
GFCCs	40	101.81M	78.89% ± 1.08%
MFCCs	10	31.68M	75.29% ± 0.67%
MFCCs	40	36.55M	72.64% ± 1.14%

GFCCs with 10 and 40 filters in filterbank, have small difference in classification accuracy, where 40 filters show slightly better, but not statistically significantly better results. MFCCs with 10 and 40 filters in filterbank, on the other hand, show statistically significant difference in accuracies. MFCCs with 40 filters have almost three percentage point lower classification accuracy than with 10 filters.

The measured number of CPU cycles it takes to extract features show that MFCCs are more efficient, compared with GFCCs. The number of CPU cycles was also measured with different number of filterbank features, summarized in table 8, to understand how increasing the number of filters increases computational complexity. Within the range of 10 to 50 filters, both feature extraction methods showed linear relationship between number of CPU cycles and number of filters. For GFCCs, adding one filter resulted approximately in additional 1.88 million cycles, while in case of MFCCs, it was only about 0.09 million additional CPU cycles.

The analysis shows that GFCCs outperform MFCCs on a noisy dataset, with regards of classification accuracy. On the other hand, GFCCs require more computational power, especially with higher number of filterbank features.

Table 8. The effect of the number of filterbank filters to the computational complexity of extracting MFCC and GFCC features.

Filterbank Filters	CPU Cycles (MFCCs)	CPU Cycles (GFCCs)
10	31.68M	45.44M
20	32.85M	64.23M
30	34.73M	83.02M
40	36.55M	101.81M
50	39.97M	120.60M

Discussion and Conclusions

The reason why GFCCs require more computations is due to the continuous functions used in GFCCs filters, where each filter has non-zero values along all frequency bins. The efficiency of calculating GFCCs on embedded systems could be improved by turning small values of the filters to zero and not performing multiplications with these values. This would require finding out suitable threshold values for turning values into zeroes and validating the accuracy loss caused by this.

The trade-off between increased computational requirements and accuracy should be taken into account when developing choosing feature extraction methods for ANNs. As embedded systems are often in noisy environments, using GFCCs instead of MFCCs is engouraged, when there is enough computational power available. While not researched in this thesis, using GFCCs could also improve the accuracy when other sources of noise are present, such as wind or rain.

MFCCs should be used, when there is not enough computational power available or when there is not a lot of noise present. Otherwise, using GFCCs should be preferred. On this specific dataset, GFCCs with 10 filters are preferred, as these show better results on the specific dataset and are fast enough on the embedded device to achieve real-time classifications.

4.4 Effects of Adding Background Noise to Training Samples

Adding background noise to the original audio clips during the training phase, as described in Section 3.3, should improve the accuracy on the test set recorded through the embedded device. To validate this, a hypothesis is proposed that models trained with added background noise show better performance on the test set. Performing this analysis with both MFCCs and GFCCs allows us to see how much the added background noise would help to increase accuracy. As GFCCs are more robust to noise, the expected result is that the difference in accuracy is smaller in case of GFCCs, than with MFCCs

Method

A DS-CNN is trained five times with the added background noise and five times without the added background noise. This is done with both MFCC and GFCC features. The average accuracies, together with confidence intervals, are found for these models on the test set. If there is no overlap between these intervals, it can be said that adding background noise to the training samples showed significant difference in test accuracy. The significant difference therefore means that the change in accuracy is due to the added background noise, not other aspects, such as random initialization of weights.

Results

The results show that adding background noise to the training samples tends to improve the classification accuracy. The results of the experiments are described in Table 9. ANN which uses GFCCs, shows around 0.6 percentage point improvement, when background noise is added to training samples. The effect, however, is not significant and it is not possible to say whether the difference in accuracy is due to added background noise or other aspects.

MFCCs show greater improvement in accuracy with added background noise. The difference is almost five percentage points and is significant. It can therefore be said, that adding background noise to training samples, when using MFCCs as features, improved the classification accuracy of the trained DS-CNN on the created bird sound dataset.

Table 9. The effect of adding background sounds from the embedded device to training samples.

Features	**Accuracy with added background noise**	**Accuracy without added background noise**
MFCC	75.29% +/- 0.68%	70.65% +/- 1.25%
GFCC	78.60% +/- 0.76%	78.01% +/- 0.57%

Discussion and Conclusions

The effect of adding background noise to training samples when using GFCCs was small and it was not possible to determine whether the increase in classification accuracy was due to added noise. The small impact of adding background noise to GFCCs demonstrated the noise robustness of this feature extraction method.

Results showed that using background noise during the training phase is especially important for features which are sensitive to noise, such as MFCCs. Adding background sound from the embedded system to the training samples is therefore encouraged for audio classification tasks, when MFCCs are being used. This solution can be used when the device does not have enough computing power for performing GFCC extraction.

4.5 Verification of Classification Process on Embedded Device

The framework proposed in this work includes automatic C++ code generation, as described in Section 3.5. The solution quantizes weights, biases and activation of a trained Tensorflow model and creates code files which could then be deployed to the Cortex-M microcontroller. To make sure that the proposed solution works as expected it is necessary to verify that the created CMSIS-NN model on embedded system works the same way as the model in Tensorflow. Verification should also include preprocessing and feature extraction process. This verification could be formulated as a hypothesis – Tensorflow model and CMSIS-NN model with same parameters will give same classification results for same input data.

Method

Understanding whether the created model in CMSIS-NN works the same way as the model in Tensorflow needs experimentation on whether same inputs produce same outputs. Testing this is separated into two parts. As the first step, quantized Tensorflow model classification results are saved together with the corresponding preprocessed inputs. Feeding preprocessed inputs to the CMSIS-NN model assures that the possible differences in preprocessing are not affecting the results. The classifications of CMSIS-NN can then be compared to the classifications of the Tensorflow model trained on regular computer. We can say that match can be found, when the two models produce same outputs for the same input. As the second step, after we have verified that CMSIS-NN model gives same predictions as Tensorflow model, the preprocessing is added to the verification process and validation is repeated. For this, a set of sound samples is randomly selected from the overall test set. Each sample is loaded as static data on a microcontroller and the embedded system then performs preprocessing and classification. The results of the classifications on quantized Tensorflow model are then compared to the ones on embedded device.

The comparison is done with 100 randomly chosen wave files from the testing set, which includes files recorded through embedded system. GFCC feature extraction with 10 filterbank filters was used. If all hundred samples are classified the same way by the quantized Tensorflow model and CMSIS-NN model, it can be said that the preprocessing and classification parts are similar enough and the possible differences (e.g. due to round-off errors) are not significant. The whole verification process is described below.

1. Create features from the 100 randomly chosen recordings
2. Get predictions for these 100 feature sets from Tensorflow model
3. Get predictions for these 100 feature sets from CMSIS-NN model through serial interface
4. Find the number of matches between the Tensorflow and CMSIS-NN model outputs on these 100 feature sets
5. Save the content of the 100 randomly chosen recordings to the embedded device
6. Get predictions for these 100 recordings through serial interface
7. Find the number of matches between the Tensorflow and CMSIS-NN model outputs on these 100 recordings

Results

The classifications of Tensorflow and CMSIS-NN models were compared on the saved features. The results showed that the classifications matched in $92.00\% \pm 5.32\%$ of cases. Reason for these mismatches was found from Hello Edge source code and fixed. Same procedure was then repeated with the fixed solution.

The unexpected behaviour was researched by comparing the layer outputs of Tensorflow and CMSIS-NN models for each of the layers. The comparison showed that difference came from average pooling layer. Further research into the Hello Edge code revealed couple of programming bugs. The function for average pooling in Hello Edge used *int* type for finding elements of an output array. Output array, however, was of type *q7_t*, allowing values between -128 and 127. This meant that values smaller than -128 or larger than 127 were converted incorrectly to *q7_t* format. For example, *int* with a value of 143 would end up being -113 when converted to *q7_t* format directly. The expected behaviour would be to clip the values which are too small or large. In the previous example, 143 should be converted to 127. As original average pooling function used integers, but finding output elements involved also division, it introduced small rounding errors, where all the values were rounded down. Second bug related to average pooling, was that the output left shift parameter was hardcoded, while its value actually depended on the results of quantization. Both of the issues were reported[1] to the Hello Edge project.

The usage of integers for finding output elements of the average pooling was replaced with floating point numbers, so division would yield correct results. The floating point value was then rounded and converted to *q7_t* format, while also clipping values smaller than -128 or values larger than 127. The hardcoded left shift variable in the average pooling function call was replaced with a variable which value was found during the model conversion process. After the fixes were introduced, all the classifications from both models matched exactly for the 100 tested samples.

When the correctness of developed CMSIS-NN model was verified, the correctness of preprocessing and feature extraction was tested. The content from 100 wave recordings was then used to extract GFCCs and perform classification on the embedded system. These experiments showed that all 100 samples gave same classification results on Tensorflow model and on embedded system. It can therefore be said, that the automatic code

[1]`https://github.com/ARM-software/ML-KWS-for-MCU/issues/103`

generation and GFCCs feature extraction implementation was verified to have same results on the embedded system as in Python.

Discussion and Conclusions

The correctness of developed GFCCs feature extraction solution and automatic CMSIS-NN model generation was verified. It involved looking for programming errors from Hello Edge work and fixing these errors. The verified framework for training DS-CNN and deploying these to Cortex-M microcontrollers allows researcher to concentrate on improving the classification accuracy, without worrying whether the model was correctly converted from Tensorflow to CMSIS-NN framework.

Section 4 analyzed different aspects of a framework, introduced in section 3, for training and converting Tensorflow ANN models to CMSIS-NN models. The analysis was performed on the example of classifying Great Tit sounds from other environmental sounds. The results showed the benefits of using DS-CNNs instead of CNNs, as the memory usage was reduced over five times and number of required CPU cycles was reduced almost six times, without significant loss of classification accuracy. Quantization showed further reduction of memory, without significant accuracy loss. Adding background noise to training samples improved accuracy for models using MFCC features, while GFCCs were able to outperform these results. The drawback of GFCCs was increased processing time and memory usage.

5 Summary

The thesis researched challenges of implementing ANNs on microcontrollers for audio classification tasks. The Hello Edge project, created by ARM, was taken as a basis for this work, which offered a framework for feature extraction, ANN model training and deployment of this model to Cortex-M microcontrollers. The work performed in this thesis analyzed and introduced ways to improve and automate parts of the Hello Edge process, while also suggesting methods to increase classification accuracy under noisy conditions.

The research was done by the use-case of classifying Great Tit sounds from other environmental sounds, including sounds from other birds. A dataset was created for this purpose, which consisted of bird sounds from different species, but also environmental sounds. The creation of dataset also included recording sounds through the embedded device to analyze how noise specific to that device can affect the classification accuracy.

The work used different techniques to reduce the size and complexity of the trained ANN and methods for improving the classification accuracy under the noise created by the embedded system. One technique for reducing the size of the ANN was using DS-CNN architecture instead of CNN architecture. Memory usage of the ANN was further improved by quantization, which reduced the resolution of weights and biases. Classification accuracy was improved by using GFCC feature extraction method instead of MFCCs and adding background noise recorded through embedded system to training samples.

The usage of DS-CNNs instead of CNNs was analyzed to give better understanding how much more computationally efficient DS-CNNs are when compared with CNNs. This involved comparing classification accuracies on the created dataset. The results showed little difference in classification accuracy, while DS-CNN was over five times more efficient with regards of computations and memory usage.

The effects of ANN quantization, which reduces memory usage by scaling down the resolution of weights and biases, was analyzed. The results showed, that quantization step decreased classification accuracy by a small amount, but also reduced memory usage up to four times.

GFCC and MFCC feature extraction methods were compared to see which performs better on microcontrollers. The analysis concentrated on how well these features work under noisy conditions and how much these features differ with regards to computational requirements. The analysis showed that GFCCs are much more robust to noise than MFCCs, but using GFCCs requires also more memory and computational power. Experiments showed that using noise specific to the embedded device as a background for training samples, can improve the accuracy, when using MFCCs. One of the results of this work is the implementation of GFCC feature extraction for microcontrollers, as there was no such library available before. The correctness of the implementation was checked by comparing the feature extraction steps on microcontroller to the ones found on regular computer, which used existing feature extraction library.

Another contribution of the thesis was improving the way how Tensorflow models, trained on regular computers, can be automatically converted to CMSIS-NN models, which can be then run on microcontrollers. Specifically, this included automating the process of finding activation quantization parameters for DS-CNN models. The output of this process is readily deployable C++ code for Cortex-M microcontrollers, that define the DS-CNN model on the microcontroller. The work was validated by comparing if model outputs on the same data using quantized Tensorflow model matched the classifications of the CMSIS-NN model on embedded device. During the process, programming bugs were found in Hello Edge code, which were fixed and reported to ARM developers.

The results of this work can be applied to sound classification problems, which are run on Cortex-M microcontrollers. The introduction of recording background noise with the embedded device and adding this as background to the training samples allows to improve the classification accuracy when using MFCCs as features. The implementation of GFCC feature extraction gives researchers and data scientists a way to use their ANN models on embedded devices with better robustness to noise.

The general way of automating the whole model conversion process has several benefits. The proposed process reduces the amount of work needed for deploying the solutions and shortens the feedback loop for acquiring new knowledge. It also reduces possible human errors and the amount of necessary knowledge about embedded systems. The automated model conversion allows researchers to concentrate on improving the ANN models and feature extraction techniques, while worrying less about the deployment process. With slight modifications in the preprocessing part, the solution can be generalized to image classification tasks.

5.1 Future Work

The work concentrated only on DS-CNNs models using post-training quantization. Further work could be done on implementing other ANN architectures. This might also include introducing new architectures and layer types to CMSIS-NN library. There are also numerous ways for quantizing ANNs, which could be researched in future works.

The implementation for extracting MFCCs and GFCCs used floating point numbers. Future research could improve these methods by using fixed point numbers instead, making the algorithms more suitable for devices without a FPU. Future research could also improve the computational efficiency of GFCCs, in order to make these more suitable to microcontrollers.

The noise introduced by the embedded system showed to decrease classification accuracy. Further research on this could be performed, to find suitable noise reduction and filtering techniques for improving accuracy of the solution. The research should also include analysis of computational requirements for different methods.

References

[1] D. Evans, “The internet of things: How the next evolution of the internet is changing everything”, *Cisco Internet Business Solutions Group (IBSG)*, vol. 1, pp. 1–11, Jan. 2011.

[2] M. A. Al-Garadi, A. Mohamed, A. K. Al-Ali, X. Du, and M. Guizani, “A survey of machine and deep learning methods for internet of things (iot) security”, *CoRR*, vol. abs/1807.11023, 2018.

[3] A. V. Dastjerdi and R. Buyya, “Fog Computing: Helping the Internet of Things Realize Its Potential”, *Computer*, vol. 49, no. 8, pp. 112–116, Aug. 2016.

[4] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, “Machine learning for internet of things data analysis: A survey”, *Digital Communications and Networks*, vol. 4, no. 3, pp. 161–175, Aug. 2018.

[5] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, “Squeezing deep learning into mobile and embedded devices”, *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.

[6] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello Edge: Keyword Spotting on Microcontrollers”, *CoRR*, vol. abs/1711.07128, 2017.

[7] J. Yiu and A. Frame, “Cortex-M Processors and the Internet of Things (IoT)”, ARM Ltd, Tech. Rep., 2013.

[8] A. Amara, F. Amiel, and T. Ea, “FPGA vs. ASIC for low power applications”, *Microelectronics Journal*, vol. 37, no. 8, pp. 669–677, 2006.

[9] O. Andersson, M. Wzorek, and P. Doherty, “Deep learning quadcopter control via risk-aware active learning”, in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[10] N. J. Cotton, B. M. Wilamowski, and G. Dundar, “A neural network implementation on an inexpensive eight bit microcontroller”, in *2008 International Conference on Intelligent Engineering Systems*, 2008, pp. 109–114.

[11] F. Mancilla-David, F. Riganti-Fulginei, A. Laudani, and A. Salvini, “A neural network-based low-cost solar irradiance sensor”, *IEEE Transactions on Instrumentation and Measurement*, vol. 63, no. 3, pp. 583–591, 2014.

[12] U. Farooq, M. Amar, E. ul Haq, M. U. Asad, and H. M. Atiq, “Microcontroller based neural network controlled low cost autonomous vehicle”, in *2010 Second International Conference on Machine Learning and Computing*, 2010, pp. 96–100.

[13] P. Petchjatuporn, W. Ngamkham, N. Khaehintung, P. Sirisuk, W. Kiranon, and A. Kunakorn, “A solar-powered battery charger with neural network maximum power point tracking implemented on a low-cost pic-microcontroller”, in *TENCON 2005 - 2005 IEEE Region 10 Conference*, 2005, pp. 1–4.

[14] O. Kücüktopcu, E. Masazade, C. Ünsalan, and P. K. Varshney, “A real-time bird sound recognition system using a low-cost microcontroller”, *Applied Acoustics*, vol. 148, pp. 194–201, May 2019.

[15] H. Goeau, H. Glotin, W.-P. Vellinga, R. Planqué, and A. Joly, “LifeCLEF Bird Identification Task 2017”, in *CLEF 2017 - Conference and Labs of the Evaluation Forum*, Dublin, Ireland, Sep. 2017, pp. 1–9.

[16] T. Peet, “Automatic Classification of Bird Vocalizations Using Convolutional Neural Networks”, Diploma Thesis, Estonian Information Technology College, 2017.

[17] A. Joly, H. Goëau, C. Botella, H. Glotin, P. Bonnet, W.-P. Vellinga, R. Planqué, and H. Müller, “Overview of LifeCLEF 2018: A Large-Scale Evaluation of Species Identification and Recommendation Algorithms in the Era of AI”, in *Experimental IR Meets Multilinguality, Multimodality, and Interaction*, vol. 11018, Cham: Springer International Publishing, 2018, pp. 247–266.

[18] R. Suthers, “Signal Production and Amplification in Birds”, in *Encyclopedia of Neuroscience*, Elsevier, 2009, pp. 805–815.

[19] G. F. Ball and S. H. Hulse, “Birdsong.”, *American Psychologist*, vol. 53, no. 1, pp. 37–58, 1998.

[20] C. K. Catchpole and P. J. B. Slater, *Bird Song Biological Themes And Variations*, 2nd edition. Cambridge University Press, 2008.

[21] D. Stowell and M. D. Plumbley, “Birdsong and C4dm: A survey of UK birdsong and machine recognition for music researchers”, Centre for Digital Music, University of London, Tech. Rep. C4DM-TR-09-12, v1.2, 2011.

[22] E. Briefer, T. S. Osiejuk, F. Rybak, and T. Aubin, “Are bird song complexity and song sharing shaped by habitat structure? An information theory and statistical approach”, *Journal of Theoretical Biology*, vol. 262, no. 1, pp. 151–164, Jan. 2010.

[23] J. Elts, A. Leito, A. Leivits, L. Luigujõe, R. Nellis, R. Nellis, M. Ots, and H. Pehlak, “Eesti lindude staatus, pesitsusaegna ja talvine arvukus 2008–2012”, et, Eesti Ornitoloogiaühing, Report, 2013.

[24] M. M. Lambrechts, “Song frequency plasticity and composition of phrase versions in great tits Parus major”, *Ardea*, vol. 85, no. 1, pp. 99–109, 1997.

[25] L. Lehtonen, “The changing song patterns of the Great Tit Parus major”, *Ornis Fenn*, vol. 60, pp. 16–21, 1983.

[26] F. Briggs, Y. Huang, R. Raich, K. Eftaxias, Z. Lei, W. Cukierski, S. F. Hadley, A. Hadley, M. Betts, X. Z. Fern, J. Irvine, L. Neal, A. Thomas, G. Fodor, G. Tsoumakas, H. W. Ng, T. N. T. Nguyen, H. Huttunen, P. Ruusuvuori, T. Manninen, A. Diment, T. Virtanen, J. Marzat, J. Defretin, D. Callender, C. Hurlburt, K. Larrey, and M. Milakov, “The 9th annual MLSP competition: New methods for acoustic classification of multiple simultaneous bird species in a noisy environment”, in *2013 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, Southampton, United Kingdom: IEEE, Sep. 2013, pp. 1–8.

[27] H. Glotin, Y. LeCun, T. Artieres, S. Mallat, O. Tchernichovski, and X. Halkias, “Neural information processing scaled for bioacoustics, from neurons to big data”, in *Proceedings of NIPS4B, international workshop joint to NIPS*, USA, 2013.

[28] A. Joly, H. Goeau, H. Glotin, C. Spampinato, W.-P. Vellinga, R. Planque, A. Rauber, B. Fisher, and H. Muller, “LifeCLEF 2014: Multimedia Life Species Identification Challenges”, in *Information Access Evaluation. Multilinguality, Multimodality, and Interaction*, Springer International Publishing, 2014, pp. 229–249.

[29] H. Goëau, H. Glotin, W.-P. Vellinga, R. Planqué, A. Rauber, and A. Joly, “LifeCLEF Bird Identification Task 2015”, in *CLEF: Conference and Labs of the Evaluation forum*, ser. CLEF2015 working notes, vol. 1391, 2015.

[30] H. Goëau, H. Glotin, W.-P. Vellinga, R. Planqué, and A. Joly, “LifeCLEF Bird Identification Task 2016”, in *Working Notes of CLEF 2016 - Conference and Labs of the Evaluation forum*, 2016, pp. 440–449.

[31] M. A. Acevedo and L. J. Villanueva-Rivera, “Using Automated Digital Recording Systems as Effective Tools for the Monitoring of Birds and Amphibians”, *Wildlife Society Bulletin*, vol. 34, no. 1, pp. 211–214, Mar. 2006.

[32] M. A. Acevedo, C. J. Corrada-Bravo, H. Corrada-Bravo, L. J. Villanueva-Rivera, and T. M. Aide, “Automated classification of bird and amphibian calls using machine learning: A comparison of methods”, *Ecological Informatics*, vol. 4, no. 4, pp. 206–214, Sep. 2009.

[33] T. M. Aide, C. Corrada-Bravo, M. Campos-Cerqueira, C. Milan, G. Vega, and R. Alvarez, “Real-time bioacoustics monitoring and automated species identification”, *PeerJ*, vol. 1, e103, Jul. 2013.

[34] C. Tsimpouris, T. Kostoulas, O. Jahn, P. Grobe, and K. Riede, “The AMIBIO database: Management tool and interface”, Amibio, Newsletter, Apr. 2012.

[35] M. Hodon, P. Sarafin, and P. Sevcik, “Monitoring and recognition of bird population in protected bird territory”, in *2015 IEEE Symposium on Computers and Communication (ISCC)*, Larnaca: IEEE, Jul. 2015, pp. 198–203.

[36] Marcos Hervás, Rosa Alsina-Pagès, Francesc Alías, and Martí Salvador, “An FPGA-Based WASN for Remote Real-Time Monitoring of Endangered Species: A Case Study on the Birdsong Recognition of Botaurus stellaris”, *Sensors*, vol. 17, no. 6, Jun. 2017.

[37] D. Moore, “Demonstration of bird species detection using an acoustic wireless sensor network”, in *2008 33rd IEEE Conference on Local Computer Networks (LCN)*, Montreal, QB, Canada: IEEE, Oct. 2008, pp. 730–731.

[38] Hongzhi Liu and N. W. Bergmann, “An FPGA softcore based implementation of a bird call recognition system for sensor networks”, in *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Edinburgh, United Kingdom: IEEE, Oct. 2010, pp. 1–6.

[39] B. Wei, M. Yang, Y. Shen, R. Rana, C. T. Chou, and W. Hu, “Real-time classification via sparse representation in acoustic sensor networks”, in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems - SenSys ’13*, Roma, Italy: ACM Press, 2013, pp. 1–14.

[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[41] J. Salamon and J. P. Bello, “Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification”, *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, Mar. 2017.

[42] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[43] M. A. Hossan, S. Memon, and M. A. Gregory, “A novel approach for MFCC feature extraction”, in *2010 4th International Conference on Signal Processing and Communication Systems*, Gold Coast, Australia: IEEE, Dec. 2010, pp. 1–5.

[44] Y. Pan and A. Waibel, “The Effects of Room Acoustics on MFCC Speech Parameter”, in *Sixth International Conference on Spoken Language Processing*, 2000.

[45] M. L. Narayana and S. K. Kopparapu, “Effect of noise-in-speech on mfcc parameters”, in *Proceedings of the 9th WSEAS International Conference on Signal, Speech*, ser. SSIP ’09/MIV’09, Budapest, Hungary: World Scientific, Engineering Academy, and Society (WSEAS), 2009, pp. 39–43.

[46] Jinfu Xu and Gang Wei, “Noise-robust speech recognition based on difference of power spectrum”, *Electronics Letters*, vol. 36, no. 14, pp. 1247–1248, 2000.

[47] B. Nasersharif and A. Akbari, “SNR-dependent compression of enhanced Mel sub-band energies for compensation of noise effects on MFCC features”, *Pattern Recognition Letters*, vol. 28, no. 11, pp. 1320–1326, Aug. 2007.

[48] O. Cheng, W. Abdulla, and Z. Salcic, “Performance Evaluation of Front-end Processing for Speech Recognition Systems”, School of Engineering, The University of Auckland, Tech. Rep. No. 621, 2005.

[49] M. A. Islam, W. A. Jassim, N. S. Cheok, and M. S. A. Zilany, “A Robust Speaker Identification System Using the Responses from a Model of the Auditory Periphery”, *PLOS ONE*, vol. 11, no. 7, Jul. 2016.

[50] X. Zhao and D. Wang, “Analyzing noise robustness of MFCC and GFCC features in speaker identification”, in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, Vancouver, BC, Canada: IEEE, May 2013, pp. 7204–7208.

[51] J. Qi, D. Wang, J. Xu, and J. Tejedor, “Bottleneck Features based on Gammatone Frequency Cepstral Coefficients”, in *Interspeech*, 2013, pp. 1751–1755.

[52] S. Ruder, “An overview of gradient descent optimization algorithms”, *CoRR*, 2016.

[53] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”, *arXiv:1704.04861 [cs]*, Apr. 2017, arXiv: 1704.04861.

[54] N. Japkowicz and M. Shah, *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[55] D. Maclaurin, D. Duvenaud, and R. P. Adams, “Gradient-based hyperparameter optimization through reversible learning”, in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15, Lille, France: JMLR.org, 2015, pp. 2113–2122.

[56] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization”, in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, ser. NIPS’11, Granada, Spain: Curran Associates Inc., 2011, pp. 2546–2554.

[57] P. Gysel, M. Motamedi, and S. Ghiasi, “Hardware-oriented Approximation of Convolutional Neural Networks”, *arXiv:1604.03168 [cs]*, Apr. 2016.

[58] ARM, “ARM Developer Suite AXD and armsd Debuggers Guide”, ARM Ltd., Tech. Rep. DUI 0066D, 2001.

[59] L. Lai, N. Suda, and V. Chandra, “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs”, *arXiv:1801.06601 [cs]*, Jan. 2018, arXiv: 1801.06601.

[60] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper”, *arXiv:1806.08342 [cs, stat]*, Jun. 2018, arXiv: 1806.08342.

[61] N. Suda and D. Loh, “Machine Learning on Arm Cortex-M Microcontrollers”, ARM Ltd., whitepaper, 2019.

[62] Y. Zhang, L. Lai, and V. Chandra, “The Power of Speech: Supporting Voice- Driven Commands in Small, Low-Power Microcontrollers”, ARM Ltd., Whitepaper, 2018.

[63] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems”, Google, whitepaper, 2015.

[64] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[65] Google. (2019). An Open Source Machine Learning Framework for Everyone: Tensorflow/tensorflow, [Online]. Available: `https://github.com/tensorflow/tensorflow` (visited on 2019-05-02).

[66] Keras. (2019). Deep Learning for humans: Keras-team/keras, [Online]. Available: `https://github.com/keras-team/keras` (visited on 2019-05-02).

[67] The Berkeley Vision and Learning Center. (2019). Caffe: A fast open framework for deep learning: BVLC/caffe, [Online]. Available: `https://github.com/BVLC/caffe` (visited on 2019-05-02).

[68] Pytorch. (2019). Tensors and Dynamic neural networks in Python with strong GPU acceleration: Pytorch/pytorch, [Online]. Available: `https://github.com/pytorch/pytorch` (visited on 2019-05-02).

[69] Apache. (2019). Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Scala, Go, Javascript and more: Apache/incubator-mxnet, [Online]. Available: `https://github.com/apache/incubator-mxnet` (visited on 2019-05-02).

[70] Microsoft. (2019). Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit: Microsoft/CNTK, [Online]. Available: `https://github.com/microsoft/CNTK` (visited on 2019-05-02).

[71] J. Bergstra, D. Yamins, and D. D. Cox, “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures”, in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML’13, Atlanta, GA, USA: JMLR.org, 2013, pp. I-115–I-123.

[72] D. Bogdanov, N. Wack, E. Gómez, S. Gulati, P. Herrera, O. Mayor, G. Roma, J. Salamon, J. R. Zapata, and X. Serra, “Essentia: An audio analysis library for music information retrieval”, in *International Society for Music Information Retrieval Conference (ISMIR’13)*, Curitiba, Brazil, 2013, pp. 493–498.

[73] A. Mesaros, T. Heittola, and T. Virtanen, “TUT database for acoustic scene classification and sound event detection”, in *2016 24th European Signal Processing Conference (EUSIPCO)*, Budapest, Hungary: IEEE, Aug. 2016, pp. 1128–1132.

[74] K. J. Piczak, “ESC: Dataset for Environmental Sound Classification”, in *Proceedings of the 23rd ACM international conference on Multimedia - MM ’15*, Brisbane, Australia: ACM Press, 2015, pp. 1015–1018.

[75] J. Salamon, C. Jacoby, and J. P. Bello, “A Dataset and Taxonomy for Urban Sound Research”, in *Proceedings of the ACM International Conference on Multimedia - MM ’14*, Orlando, Florida, USA: ACM Press, 2014, pp. 1041–1044.

[76] J. Salamon, D. MacConnell, M. Cartwright, P. Li, and J. P. Bello, “Scaper: A library for soundscape synthesis and augmentation”, in *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, New Paltz, NY: IEEE, Oct. 2017, pp. 344–348.

[77] H. Nyquist, “Certain topics in telegraph transmission theory”, *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

Appendix 1 – Software Project for Improved Framework

The code for the improvements introduced in section 3 can be found on `https://github.com/tpeet/ML-KWS-for-MCU`.

The code was built on top of the Hello Edge project, as it improves open source collaboration and gives developers, who already use Hello Edge work, a familiar code structure. The code includes Python scripts for developing an DS-CNN Tensorflow model for audio classification task and generating C++ code from the trained model. The project also includes C++ code for Cortex-M microcontrollers, which can use automatically generated CMSIS-NN model.