TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jonathan Räni 201430IVSM

Implementation of Urban Environment Noise Classification Application on a Low Power Microcontroller

Master's thesis

Supervisors: Jaanus Kaugerand
Phd
Konstantin Bilozor
MSc

Tallinn 2022

Author’s declaration of originality

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

Author: Jonathan Räni

09.05.2022

Abstract

As more people keep moving to urban areas then noise pollution is a growing problem. Noise can contribute to health issues and therefore cities are looking for ways how to analyse and mitigate noise issues. Currently in Tallinn there are hundreds of low power devices that measure sound pressure levels, that is used to analyse noise issues all over the city.

This thesis aims to implement a solution on a low power microcontroller that could provide information on what type of noise was measured. Different noise types include traffic, industrial, aircraft and human noise. To do that TensorFlow Micro Speech application is integrated to a low power microcontroller. Two pre-processing algorithms are described and implemented. Suitable machine learning model is trained for a low power microcontroller. Conclusions are drawn whether Micro or MFCC algorithm is better for pre-processing the noise. Results and analysis are provided to show how feasible it is to run this application on a low power microcontroller.

This thesis is written in English and is 84 pages long, including 4 chapters, 35 figures and 25 tables.

Annotatsioon

Linnakeskkonna müraklassifitseerimise rakenduse implementeerimine madala voolutarbega mikrokontrollerile

Müra reostus on üha suurenev probleem linnades, kuna aina rohkem inimesi kolib linnadesse. Mürarikkas keskkonnas elamine võib põhjustada tervisehädasid, mistõttu linnad otsivad mooduseid kuidas analüüsida ja lahendada müraprobleeme. Tallinnas on hetkel mitusada madala voolutarbega mikrokontrollerit, mis mõõdavad müratasemeid, mida kasutatakse müraprobleemide analüüsimiseks.

Käesoleva töö eesmärgiks on luua rakendus madala voolutarbega mikrokontrolleril, mis suudaks tuvastada mis tüüpi müra mõõdeti. Erinevad müratüübid mida klassifitseeritakse on liiklusmüra, tööstusmüra, lennukimüra ja inimmüra. Selle eesmärgi saavutamiseks integreeritakse TensorFlow Micro Speech aplikatsioon mikrokontrollerile. Uuritakse kahte erinevat heli eeltöötlus algoritmi. Treenitakse sobilikud masinõppe mudelid, mida jooksutatakse mikrokontrolleril. Tehakse järeldused kas Micro või MFCC algoritm on sobilikum müra eeltöötluseks. Tulemused ja analüüsid näitavad kas ja kui realistlik on sellist rakendust madala voolutarbega mikrokontrolleril jooksutada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 84 leheküljel, 4 peatükki, 35 joonist, 25 tabelit.

List of abbreviations and terms

ADC	Analog-to-Digital Converter
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
GPIO	General Purpose Input/Output
ISC2PT	Intelligent Smart City and Critical Infrastructure Protection Technologies
I2C	Inter-Integrated Circuit
LDMA	Linked Direct Memory Access
MCU	Microcontroller Unit
MFCC	Mel-Frequency Cepstral Coefficients
RAM	Random-Access Memory
RTOS	Real Time Operating System
SmENeTe	The Smart Environment Networking Technologies
STFT	Short-time Fourier Transform
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WAV	Waveform Audio File Format

Table of Contents

1 Introduction 11
1.1 Background.. 11
1.2 Research problem 12
1.3 Research goals 13
1.4 State of the art... 14
2 Method.. 20
2.1 Microcontrollers 20
2.1.1 Microcontroller features 20
2.1.2 ADC and LDMA 21
2.1.3 EFR32MG12 23
2.1.4 FreeRTOS... 23
2.2 Machine learning 24
2.2.1 General overview.. 24
2.2.2 Neural networks.. 26
2.3 Audio pre-processing.. 29
2.3.1 Signal in time domain.. 29
2.3.2 Discrete Fourier Transform (DFT) 30
2.3.3 Short Time Fourier Transform (STFT) 31
2.3.4 Mel spectrograms 34
2.3.5 Mel-Frequency Cepstral Coefficient (MFCC) 36
2.4 TensorFlow Micro Speech.. 37
2.4.1 Integrating Micro Speech to SmENeTe platform............................... 39
2.4.2 Micro pre-processing... 44
2.4.3 Using MFCC instead of Micro pre-processing 48
3 Results and analysis.. 50
3.1 Conforming MFCC features.. 50
3.2 MFCC and Micro pre-processing tests and analysis 51
3.3 Tests with “yes” and “no” models.. 52

3.3.1 Test 1 53
3.3.2 Test 2 55
3.3.3 Test 3 56
3.3.4 Test 4 57
3.3.5 Test 5 59
3.3.6 Conclusions 60
3.4 Tests with “traffic” and “human” models 60
3.4.1 Test 1 60
3.4.2 Test 2 65
3.4.3 Conclusions 65
3.5 Tests with “traffic”, “industrial” and “human” models 65
3.5.1 Test 1 66
3.5.2 Test 2 70
3.5.3 Test 3 71
3.6 Test with “traffic”, “industrial”, “aircraft” and “human” model 72
3.7 Tests with alternate dataset 73
3.7.1 Test 1 with “traffic”, “industrial” and “human” model 73
3.7.2 Test 2 with “traffic”, “industrial” and “human” model 75
3.7.3 Test 3 with “traffic”, “industrial”, “aircraft” and “human” 76
3.8 Conclusions 76
3.9 The effect of running neural networks with radio thread 77
4 Summary 79
References 80

List of figures

Figure 1. How the sound is converted into digital representation on a microcontroller. 21
Figure 2. SmENeTe device with a EFR32 microcontroller... 23
Figure 3. Small neural network. 27
Figure 4. Waveform of a person saying "yes". 30
Figure 5. Waveform of a person saying "no". 30
Figure 6. DFT calculated from a waveform of a person saying “yes”. 31
Figure 7. DFT calculated from a waveform of a person saying "no". 31
Figure 8. Spectrogram of a person saying "yes". 33
Figure 9. Spectrogram of a person saying "no". 33
Figure 10. Mel-scale [48]. 34
Figure 11. Mel-spectrogram of a person saying "yes". 35
Figure 12. Mel-spectrogram of a person saying "no". 35
Figure 13. MFCC of a person saying "yes". 37
Figure 14. MFCC of a person saying "no". 37
Figure 15. Thunderboard Sense 2. 38
Figure 16. Debug log from running Micro Speech application on Thunderboard Sense 2. 39
Figure 17. Linked list of channel descriptors. 40
Figure 18. Ring buffer of channel descriptors. 42
Figure 19. Overview of buffers and windowing 44
Figure 20. Hann function [57]. 46
Figure 21. Spectral leakage [56]. 46
Figure 22. Effect of Hann function [56]. 46
Figure 23. Micro spectrogram of a person saying "yes". 47
Figure 24. Micro spectrogram of a person saying "no". 47
Figure 25. TensorFlow MFCC of a person saying "yes". 49
Figure 26. TensorFlow MFCC of a person saying "no". 49
Figure 27. "tiny_conv" model architecture by Netron [60]. 52
Figure 28. Debug log of test 1 on SmENeTe device. 54

Figure 29. Debug log of test 4 on SmENeTe device. 58
Figure 30. Debug log of feature matrices. 64
Figure 31. Test 1 results when the device was in silent environment. 67
Figure 32. Test 1 results when human noise was played back. 68
Figure 33. Test 1 results when traffic noise was played back. 69
Figure 34. Test 1 results when industrial noise was played back. 70
Figure 35. "tiny embedding conv" model architecture. 71

List of tables

Table 1. Initial common parameters. 52
Table 2. Test 1 parameters.... 53
Table 3. Test 1 results.... 53
Table 4. Test parameters.... 55
Table 5. Test 2 results.... 55
Table 6. Test 3 parameters.... 56
Table 7. Test 3 results.... 56
Table 8. Test 4 parameters.... 57
Table 9. Test 4 results.... 57
Table 10. Test 5 parameters.... 59
Table 11. Test 5 results.... 59
Table 12. Initial common parameters. 60
Table 13. Test 1 parameters.... 61
Table 14. Test 1 results.... 61
Table 15. Test 2 parameters.... 65
Table 16. Test results.... 65
Table 17. Initial common parameters. 66
Table 18. Test 1 parameters.... 66
Table 19. Test 1 results.... 66
Table 20. Test 3 parameters.... 71
Table 21. Test 3 results.... 72
Table 22. "traffic", "industrial", "aircraft" and "human" model results. 72
Table 23. Test 1 results.... 74
Table 24. Test 2 results.... 75
Table 25. Test 3 results.... 76

1 Introduction

This thesis consists of 4 main parts. The first part is introduction where general information on the topic of this thesis is provided. The second part describes the concepts of this thesis and covers the implementation part. In the third part results and analysis are provided. The fourth part sums up the thesis.

1.1 Background

Since more and more people are moving out of rural areas to urban areas then noise pollution is an increasing problem by the day. The most common source of noise that may cause health issues include traffic, airplanes, construction, people etc. Some studies [1], [2] have shown that living in a constantly noisy environment contributes to serious health issues, which is why scientists and engineers from many countries have researched where and how to measure noise levels and what action to take to relieve them.

In order to mitigate health issues and to inform its citizens about noise levels and the types of noise in different city parts, cities are becoming interested in continuous monitoring of noise in the outdoor environment in different city parts.

Some cities have many small electronic devices monitoring the environment and are making some processes more efficient or convenient for the people [3]. For example, one of the systems in smart cities could be smart traffic lights that can intelligently change the length of red and green phases depending on the need. Smart cities can also make use of acoustical sensors for noise measurement, either specially installed near noise sources or mobile to track noise in different places [4]. For keeping these noise monitoring smart city devices operational for as long as possible, low power embedded microcontrollers can be used. In this way, there can be hundreds or even thousands of microcontrollers installed in urban environments that gather various information about the environment. For example, microcontrollers could be equipped with various sensors such as noise, temperature and humidity, and send it to the cloud server, where the data will be processed.

Currently in Tallinn there are hundreds of low-power sensor devices deployed to different key locations to measure sound pressure levels [5]. These devices are equipped with an analogue microphone which provides raw data that is converted into sound pressure level in decibels using a local computational unit. This data is sent to the cloud server for further analysis. However, the sound pressure level by itself doesn't always give enough information. Therefore, this thesis provides a solution to have different types of noise classified using convolutional neural networks without any AI accelerators to save the cost of the devices. This allows cities or other clients to take actions depending on what type of noise is affecting their environment the most.

This thesis is a part of an Intelligent Smart City and Critical Infrastructure Protection Technologies (ISC2PT) project [6] that focuses on developing the next generation of IoT networking technology components for supporting smart environment applications. The ISC2PT project builds on SmENeTe project [7], during which the sensor devices used in this thesis were designed. The main purpose of SmENeTe project was to develop and test the low power communication technology and ad hoc network with low power consumption, etc. In ISC2PT project these low power sensor devises are supplemented with AI capabilities such as noise classification. Both projects were carried out by research Laboratory for Proactive Technologies from Tallinn University of Technology together with the company named Thinnect OÜ. The sensor devices used in this thesis are equipped with a solar panel for power and for charging the battery, microcontroller, and a microphone.

1.2 Research problem

It is known that some solutions already take advantage of running neural networks on embedded low power devices. Some of them utilize AI accelerators to speed up neural network inferences. One solution [20] found very promising results in noise classification, however that research used a more powerful STM microcontroller, did not implement radio for communicating data to the server, and used Mel spectrograms for pre-processing audio data.

This thesis investigates a solution that uses a Silicon Laboratories EFR32 family microcontroller with twice as slow CPU as the aforementioned STM. Radio thread will

be implemented together with neural networks. Micro and MFCC pre-processing are investigated.

In this thesis answers will be sought to the following research questions:

- How to run convolutional neural networks-based application on low power edge devices without using AI accelerators?
- How to implement the functionality so that the device can output information about different types of noise?
- How can MFCC algorithm be used for pre-processing the sound information?
- How does Micro pre-processing compare to MFCC computational performance and machine learning model accuracy wise?
- How to develop a solution which can output statistics of different classes of noise?
- How to synchronize different FreeRTOS threads so that the data can be processed and sent to the server as efficiently as possible?

1.3 Research goals

The first goal of this thesis is to port TensorFlow Micro Speech application [8] into an EFR32 microcontroller. Micro Speech is an application that can be ported to different types of hardware. This application runs TensorFlow Lite model, that detects different keywords such as if a person is saying “yes” or “no”. Once this application is integrated to EFR32 then most of the necessary functions are already available for processing audio and neural networks.

TensorFlow Micro Speech application is meant for hardware that uses a digital microphone. Because of that the ADC and LDMA drivers must be changed as the ISC2PT project has an analogue microphone in use. Micro Speech application uses a Micro pre-processing (used in production across Google [9]) for audio processing. It has shown very good results in practice but hasn’t been published in the research literature [9]. However, MFCC pre-processing has shown very good results in practice and is backed up by scientific literature. Therefore, as a second goal this thesis will also investigate if the MFCC algorithm can be used for audio pre-processing.

When drivers are implemented, and signal processing is in place then, as a third goal, new pre-trained classification models with new parameters are to be taken into use. Initial

results will be examined, and further steps will be defined based on these results. Also, at this point separate threads for radio and TensorFlow can be implemented using FreeRTOS. To keep the system more deterministic and safer it is required to take RTOS "friendlier" drivers and loggers into use. On top of that statistics of classification results will be implemented. Statistics will be sent to server via the radio. Threads must be synchronized.

1.4 State of the art

There is a relatively new and fast-growing field of machine learning called Tiny machine learning (TinyML). It involves running machine learning algorithms and performing sensor data analytics on low power embedded devices. These are mostly small battery-operated devices such as microcontrollers that can communicate with the cloud via some communication protocol. Over the years these devices have become cheaper, more energy efficient, faster and smaller. These improvements are creating huge opportunities for businesses in industry and academia. To make TinyML possible both software and hardware optimization techniques have been engineered. For example, at the hardware level there are now accelerators that speed up the calculations and at the software level there are pruning and quantization techniques to make machine learning models more efficient for TinyML devices. TinyML is being deployed in many domains including healthcare, security and surveillance, Smart Things (IoT), Industrial Monitoring and Control, Smart and Secure Societies [10].

For example, in Africa, a research group has developed a TinyML based self-diagnostic kit for detecting respiratory diseases from exhaled breath. This TinyML device has a gas sensor to collect Volatile Organic Compounds from exhaled breath as an input for the diagnosis process. Then TinyML model is used to detect if a person has a respiratory disease or not. The users are getting real-time feedback from the device. Furthermore, another research group is developing a TinyML based prototype that can detect water-borne diseases like cholera. This is done by monitoring water's physicochemical patterns. Usually detecting water-borne diseases is an expensive laboratory processes so this device could help many people without costing too much money. Another TinyML example comes from Rwanda, where researchers are developing IoT-based precision farming system that can sense different soil parameters, integrate with forecasted weather

information and use embedded AI to determine which crop would grow best under the existing conditions with minimal use of fertilizers. Finally, TinyML has also been proposed for use in wildlife conservation. Idea is to track wildlife movement and aid in wildlife conservation, especially aiding wildlife at risk. One similar system has GPS trackers attached to elephant collars that can capture elephant movements [11].

One research described in [12], used TinyML tools (Tensorflow Lite) for creating traffic sign recognition system for mobile-based applications, that would benefit autonomous car driving systems. This system nicely demonstrates the need of on-device model inferences. Since autonomous cars are essentially hard real-time systems then deterministic and minimized latency is important. For example, if done right, then it is a considerably faster to run the detection algorithm on-device rather than sending the input data to the cloud and then waiting for the response. This application runs on an android device in a 12 FPS rate for the quantized model. Quantized model showed 4 times faster detection compared to the float model on the mobile device. The image is fed to trained CNN which returns a set of recognized objects surrounded by bounding boxes. This research provides a good starting point towards a better autonomous car driving systems [12]. Another similar solution did an object detection system that that could accurately detect objects in real-time on edge devices. It used MobileNet Single Shot Multibox detection. It was fast and gave accurate results on an edge device, which was verified by detecting objects such as truck, person, dog, bicycle, car and bus [13].

Urban areas are facing challenges in waste management systems due to rapid growth of populations in cities, causing huge amount of waste generation. Therefore, one research developed a smart waste management system using deep learning model that improves the waste segregation processes and enables monitoring bin status in an IoT environment. It used TensorFlow Lite and Raspberry Pi 4 with a camera and a servo motor connected to a plastic board that categorizes waste into respective waste compartment. Ultrasonic sensor was used to monitor the waste fill percentage, and a GPS module obtains the real-time latitude and longitude. The LoRa module sends the status of the bin to the Lora receiver and based on that information system administrator knows whether the bin is empty or should it be emptied as soon as possible [14].

In one project a pre-trained YAMNet model [15] is retrained and used to perform audio classification in real-time to detect gunshots, glass shattering, and speech. The model runs

on an edge device. The hardware on which the application was deployed to was Jetson Nano. This is relatively expensive and power-hungry device; however, it can run inferences quite fast. TensorFlow was used as a machine learning library. Online datasets FSD50K [16] and UrbanSound8K [17] were used for training. Mel-spectrograms were used for pre-processing the data. They concluded based on scientific literature that CNNs would be best for this audio recognition application. The model architecture consisted of YAMNet Layer (feature extraction), embeddings input layer, dense layers and dropout layers. The accuracy with this setup was about 95% with quantized model. RAM usage was about 5.5MB and inference speed 100ms. They compared different quantization methods and TensorFlow Lite Dynamic Quantization proved to be the most efficient albeit with some minor accuracy loss [18].

Another work did an environmental sound classification application on mobile devices. They used online datasets DCASE [19] and ESC-50 [20] and some data collected by themselves for model training. The chosen sound classes were the following: snap, knock, laugh, cough, keys jangling, keyboard typing. They used MFCC for pre-processing the data, sampling rate of 44100Hz, 128 MFCC coefficients, FFT window length of 1024, hop length 512, and 128 Mel-bands. The model architecture consisted of 8 layers. Layer 1 was 2D convolutional layer with 16 filters and a Rectified Linear Unit (ReLU) activation, padding and batch normalization. Second layer was same as the first but followed by max pooling with pool size of 2 and stride length of 2. Layers 3-8 were similar, but with some variances to the number of filters. The MFCC quantized model had an accuracy of 90% with the size 4.5MB. Classification took less than 200ms when ran on Samsung Galaxy S8 and S9 Android phones [21].

Another research paper investigated the effectiveness of using a balanced combination of fog computing and cloud computing for urban sound classification. They tested 2 setups. One configuration was where the edge device collects the data and classifies the data on-device. Results where then sent wirelessly to the server storage. The second setup focused on cloud computing meaning all the computations are done in the cloud. In this setup the device is continuously sending the sound data to the server. Server converts these sound files into feature map and each feature map is used to classify incoming sound. These tests were conducted on Raspberry Pi 3 Model B, which is significantly more powerful than microcontrollers. UrbanSound8K online dataset was used which contains 10 noise classes. They used 5 different feature extraction algorithms including MFCC. They used

2 hidden layers containing 280 and 300 nodes respectively. The power consumption between their 2 test setups were almost the same. The first setup used 21.46mW more power. In terms of latency the second setup was considerably worse. The network slowed down as the number of devices grew [22].

One study in Sweden proposed a noise classification solution using a low-power and inexpensive IoT device. They used MFCC for audio feature extraction and supervised classification algorithms. The hardware used was Raspberry Pi Zero W. UrbanSound8K dataset was used. In the end they achieved accuracies from 85% up to 100% [23].

The continued exposure to noise, in New York, where millions of people live, has proven effects on health. To investigate and aid in the mitigation of urban noise, a network of 55 sensor nodes has been deployed across New York city. These sensors collect sound pressure level and audio data. Each device contains a quad core single board computer - Raspberry Pi 2B. They collect sound pressure level and 10 second audio snippets that are collected for the purpose of machine learning. Every 60 seconds 150kB of sound pressure level data is sent out. Every 20 seconds 500kB of audio snippets are sent to the server. And Every 3 seconds 1kB of node status is sent. These kinds of loads put a big strain on network and a lot of storage space is required to store that kind of data. The used Raspberry Pi 2B is a lot more expensive and power hungry compared to microcontrollers. They use Wi-Fi for sending the information from sensor nodes to the server. This means that they can only deploy their sensors where there is a Wi-Fi connection [4].

In Dublin City researches provided near real-time noise monitoring at 14 sites. Aim is to provide information on sound levels which people are being exposed [24]. Dublin City Council investigates noise pollution complaints and takes enforcement action if necessary. There are some permissible hours allocated during which noisy work is allowed. If noise complaints are made outside that time and if the noisy work has not been approved by the Dublin City Council, then the Noise Control Unit will deal with the complaints. Complaints can be about noise from commercial premises, music from night houses, noise from outdoor music events, noise from security alarms etc [25]. In [26] an annual report of noise in 2017 is described.

There is a noise monitoring system in Barcelona that has 112 devices, 86 sound sensors and 26 sound level meters [27]. Sound sensors are used in fixed locations for a long-term

analysis. Sound level meters can measure more information and make audio recordings with the purpose of identifying the source of the sound. The main types of noise that they have measured includes traffic, recreational noise, private activities, waste collection and construction work [27]. This paper mostly discusses issues and challenges to improve the noise monitoring network and doesn't go into specifics how the network is implemented.

The CENSE project in France aimed at proposing new methodology to produce more realistic noise maps. It relied on a dense network of low-cost sensors deployed in the city of Lorient. This project produced advances on noise modelling, lost-cost sensor network technologies, data assimilation techniques applied to sound levels prediction, urban sound recognition and perception. They implemented a hybrid communication scheme where each node of the network is a sensor with acoustic monitoring capability. Sensors are energetically autonomous and send wirelessly data to the other types of nodes, the gateway nodes. A gateway node is wired both in terms of power and network connectivity. Each sensor produces acoustic data, pressure level statistics that are sent every 10 seconds. Some other information is sent out every 60 seconds and 5 minutes. Sensor nodes send data to gateways and gateways will send data through the Internet to an archiving architecture using the OpenSensorHub protocol. The calculation of noise source presence ratios is implemented on the low-cost sensors. For example, classification of traffic, 2-wheel motor vehicles, human voices and birds. Source classification is done using a deep convolutional architecture [28].

In Norway another state-of-the-art environmental sound classification prototype was made. This system used STM32L476 microcontroller with CNNs. The CPU of this microcontroller runs at 80MHz. It has 1024kB of FLASH and 128kB of RAM. They used Urbansound8k online dataset to train a model for classifying 10 types of noise found in cities. For pre-processing they chose to use Mel-spectrograms. In total they used 10 different model architectures with some minor variances. In general, the model design contained a Conv2d layer, batch normalizations, ReLU, max pooling, flattening and dense layers. The accuracies varied between 60% to 72%. CPU usage varied between 5% and 100%. The model execution took 43ms and calculating Mel-spectrogram approximately 60ms. On-device testing by playing back sound via headphones into the microphone scored 72% on the validation-set [29].

The vision for Tallinn is to have hundreds or even thousands of low-cost low power microcontrollers deployed in various locations that can classify the type of noise aside from measuring sound pressure levels which is done now. Sensor nodes consist of low-power microcontrollers that could act as a TinyML device. Since there have been complaints to Health Board of Estonia regarding the noise then information provided by these devices could be used to investigate and mitigate unnecessary noise. This thesis aims to provide a foundation for that vision.

2 Method

In this paragraph general concepts are described that are required to implement a noise classification application on a low power microcontroller.

2.1 Microcontrollers

Microcontroller is a small piece of hardware that can run various types of embedded applications. Microcontroller can be a part of a bigger device, or it can run a smaller system on its own. The microcontroller market is constantly growing as they are used in many different fields of industry such as automotive, telecommunication, health care, smart cities, consumer electronics etc [30].

The need for microcontrollers in smart cities is motivated because MCUs are ideal for running simple applications cheaply. The cost of the most used microcontrollers is only several euros as of 2022. Furthermore, since microcontrollers don't use a lot of power and some are even optimized to be low power, then the cost of maintenance is also low. Devices with microcontrollers can be deployed, for example, to lighting posts besides urban streets to collect different types of data. Such sensors can then operate autonomously several months before the battery needs to be changed if it needs changing at all.

Some devices in smart cities utilize a paradigm called edge computing. Essentially, it is a practice of processing data on source devices without involving the server. The benefit of doing this is that it reduces both latency and network load and also reduces overhead at the cloud. For example, the goal of a device could be to send noise levels in decibels to the cloud. Calculating the decibels on a microcontroller and sending this data to the cloud instead of sending raw audio data significantly reduces the network load. These are called edge devices.

2.1.1 Microcontroller features

Microcontroller unit (MCU) is an integrated circuit that contains CPU, RAM, FLASH, and input/output peripherals on a single chip. Arm Cortex M4 is one of the most widely used processors in embedded devices. Most commonly the clock speed for

microcontrollers is between 40MHz and 240MHz. The amount of RAM is usually between 2kB up to 512kB. And FLASH varies between 8kB to 2MB.

Input and output peripherals are what makes microcontrollers useful. Peripherals are basically an interface with the outside world [31]. Examples of peripherals are ADC, GPIOs, UART, I^2C, timers, USB, LDMA etc. In this thesis ADC and LDMA are used.

2.1.2 ADC and LDMA

Analog to Digital Converter (ADC) is a peripheral on a microcontroller that is used to convert an analogue signal into a digital representation. Figure 1 shows the role of the ADC on a microcontroller.

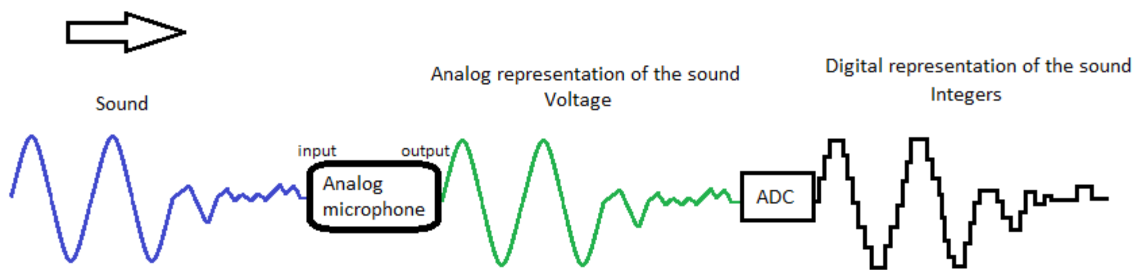

Figure 1. How the sound is converted into digital representation on a microcontroller.

Vibrating air molecules move in waves that an analogue microphone can detect. Microphone will convert sound waves into a voltage between 0.0V and 3.3V in most cases. ADC will convert the continuous voltage into a digital signal. Digital signal consists of integers between 0 and 4095 in most cases. The amount of how many different numbers can represent an analogue signal depends on the ADC precision. For example, if the precision is 12 bits, then 2^{12} equals 4096 different numbers. Every number represents a different voltage. How much voltage does one number represent depends on the ADC resolution. For example, if the maximum voltage of the ADC is 3.3V and the precision is 12 bits then the resolution is 3.3V / (2^{12} - 1) = 0.81mV. This means that if number 0 as a digital signal equals to 0V then number 1 equals to 0.81mV, number 2 equals 1.62mV and so on until number 4095 represents 3.3V. This essentially means that the smallest change in voltage that ADC can detect is 0.81mV.

The sampling rate of the ADC specifies how often it takes an analogue sample and converts it into a digital number. For example, if the sampling rate is 8kHz then ADC is measuring the voltage 8000 times per second and converts the samples into a digital form. If the voltage fluctuates very frequently and the sampling rate is very slow then a

phenomenon called aliasing can happen. This means that the digital signal is not representing the actual analogue signal accurately and some higher frequency events can be missing from the digital signal. To avoid aliasing the sampling rate should always be at least twice as much as the highest frequency in the measured signal. This is stated by the Nyquist criteria [32].

A Linked Direct Memory Access (LDMA) controller is a peripheral on a microcontroller that can move data from one location to another without CPU intervention. LDMA can help to reduce CPU workload and can perform data transfers more energy efficiently than the CPU. For example, while LDMA is moving a large amount of data from one memory location to another then at the same time CPU can process more important instructions [33].

LDMA can have different combinations of source and destination transfers. LDMA can transfer from memory to a memory, from a memory to a peripheral, from a peripheral to a memory and from a peripheral to a peripheral. For example, the source could be an ADC FIFO, and the destination could be a memory location in the RAM (application buffer) [33].

LDMA has channel descriptors. Based on how descriptors are configured they determine what the LDMA controller will do when it receives a DMA transfer request. Channel descriptors reside in RAM, usually forming a linked list or a ring buffer. For example, a descriptor contains the memory address where the data is read from, the memory address where the data is written to and also the link address of the next descriptor to form a linked list or a ring buffer. It also contains the number of bytes to be transferred etc [33].

LDMA reads a block of data at a time from the source, then it stores it in the LDMA's local FIFO and finally writes the block out to the destination from the FIFO. DMA supports byte, half-word and word sized transfers. One descriptor can be configured to move up to 2048 units of data. For example, a descriptor can be configured to move 4 16-bit values from ADC FIFO at a time. And when LDMA has moved 2048 16-bit values to RAM then LDMA will trigger an interrupt to indicate that the data is ready [33].

ADC will issue a request (REQ) when ADC FIFO is full. Then the LDMA will move the data from that FIFO to the destination. LDMA could also be configured in such a way

that LDMA will start transferring as soon as there is at least 1 element in ADC FIFO, however this method is not as efficient.

2.1.3 EFR32MG12

EFR32MG12 is a Silicon Laboratories EFR32 family microcontroller. This is used in this thesis since it's a part of a SmENeTe device as seen in Figure 2.

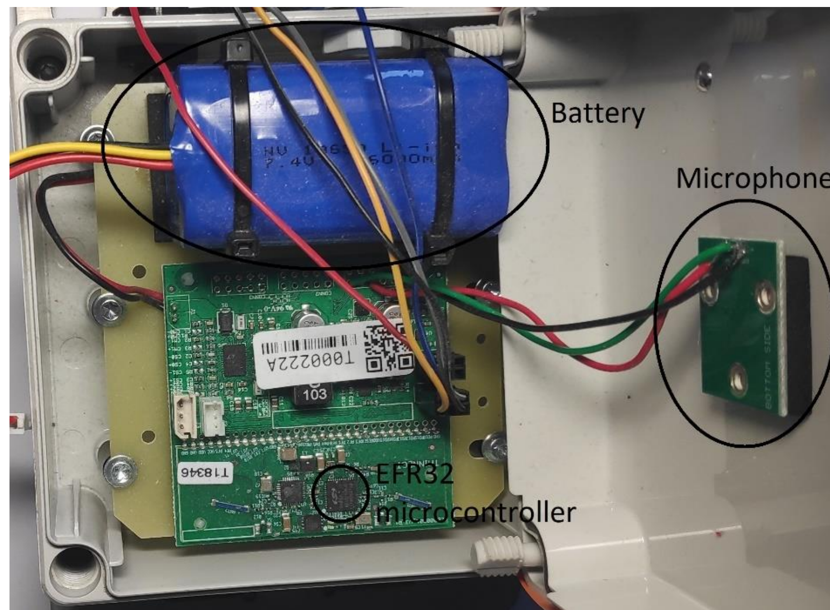

Figure 2. SmENeTe device with a EFR32 microcontroller.

EFR32 has a 32-bit 40MHz ARM Cortex-M4 core. 256kB of RAM and 1024kB of FLASH. Furthermore, it has ADC and LDMA peripherals and support for IEEE 802.15.4 standard. This standard specifies the requirements for radio transiever's physical and MAC layers [34]. That standard is the basis for the data communication via the radio that SmENeTe devices use. SmENeTe devices use proprietary protocol stack for communication.

2.1.4 FreeRTOS

FreeRTOS [35] is a real-time operating system for microcontrollers that allows to run different threads in parallel and synchronize them. If a processor only has a 1 core, then the parallelism is just a way of saying that each thread gets its own time slice when it is allowed to run. For example, one thread can run for 10 milliseconds and then has to give the control to other threads that can also run for 10ms. This creates the illusion that threads are running in parallel. EFR32 has only 1 core so the threads must take turns to run.

Sometimes for efficiency reasons it is better to suspend a thread until another thread has finished processing or producing some resource. For example, if there is a radio thread for sending out the data and noise thread for calculating the decibels. Radio thread doesn't need to run all the time, but only when the decibels have been calculated. Furthermore, sometimes the requirements can dictate that we might want to calculate the average noise level over 10 minutes for example. When a noise thread has averaged 10 minutes' worth of noise then it can signal the radio thread to send out the data. For this purpose, there are Thread Flags [36] in FreeRTOS.

Using Thread Flags one thread can wait for the flag, and the other thread can set the flag. When the thread calls the wait for flag function then it is suspended until the other thread uses the set the flag function. This mechanism can be used to solve one of the requirements of the noise recognition application implemented in this thesis. The requirement says that 10 noise classifications must be averaged and then sent out. It means that when the application outputs what type of noise is currently dominating in the environment then it will be stored locally in the array. After the 10 results are gathered the statistics are calculated of how many percentages each noise was present in the past 10 results. For example, if there were 5 results that said traffic noise was present, 3 results said human, 1 result industrial and 0 aircraft then the output would be that 50% was traffic, 30% human, 10% industrial and 0% aircraft. This data is then written to the radio buffer and noise thread will signal the radio thread to send out the data.

2.2 Machine learning

2.2.1 General overview

Machine learning is a popular technique of solving software engineering problems. It makes use of past observations or its experience in order to produce as correct as possible output. Or in other words as Arthur Samuel defined it: "Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed [37]."

For example, in traditional programming it is very easy to make an algorithm that can detect when the water is boiling. This algorithm would have an input of water temperature, a simple if statement that checks whether the temperature is over 100 degrees and based on that result a decision is made. However, some phenomena, events,

processes or objects, that may have similar parameters as temperature, are hard to describe and even harder to write code for. Consider images of cats and dogs. Humans can easily distinguish between cats and dogs, but how? What makes a cat? What are the distinct elements that cats have and dogs don't and vice versa? In programming it gets even more complicated since images are essentially matrices of numbers. What combination of numbers makes up a cat? How would someone program it if the image has 2 million pixels?

Machine learning comes in handy in problems where it's difficult to interpret what input maps to a specific output. Most commonly a limited amount of dataset is used to train a machine learning model. That means that the machine learning model uses the dataset to learn how to generalize the whole system so that it can even produce correct outputs with the data that it has not been trained with. A machine learning model is essentially a file that contains complex math calculations to convert the input to an output. The type of input and output can be configured during the training process.

Machine learning training is a process where the parameters of the model are fine-tuned so that the model could output more accurate results. There are several different types of machine learning that differ in the way they are trained. Most common types are supervised learning and unsupervised learning.

Supervised learning means that during the training every input correlates to some output. It means that every input has some label that says what should the result be for that input. For example, images of cats could be labelled as number 0, and dogs as 1. In that way when the model has finished training it will output 0 if an image of a cat is fed into the model and will output 1 for dogs. Unsupervised training means that the model must learn from the inputs that have not been labelled. In this thesis only supervised learning is used.

One subclass of supervised learning is classification approach. It means that the output of the model will be an array of numbers, where each element represents a probability of an input belonging to that class. For example, if an image of a cat is fed into a model, then the output could be, for example, 0.9 and 0.1, where 0.9 represents that the model is 90% confident that the input was a cat. In this thesis only classification is used. Different types of noise are to be classified.

Some issues that can happen with the model are overfitting and underfitting. Overfitting means that the model has essentially memorized how specific inputs map to specific outputs but can’t perform well with the data that it hasn’t been trained with. It can happen when the training process is unnecessarily too long, and the dataset is small. Underfitting usually occurs when the model has not been trained with enough data and time. It also can’t generalize and perform well.

The tools used for machine learning in this thesis are TensorFlow and Keras.

2.2.2 Neural networks

Deep learning is a subset of machine learning that uses neural networks to create a model. Neural networks consist of neurons that are connected to each other. Neurons are nodes that are inspired from the neurons in the human brain. The combination of hundreds or thousands of nodes are what allows the network to learn from past observations.

In general, the neural network consists of an input layer, hidden layer and output layer. The input layer consists of nodes where the inputs are written. For example, if an audio file has 8000 samples, then there would be 1 node for every sample. Samples are written into nodes. The output layer consists of nodes that represent output. For example, if the purpose of a model is to classify “yes” and “no” keywords then there would be 2 nodes. One node would represent “yes”, the other one “no”. Finally, the hidden layer usually consists of hundreds or thousands of nodes. It’s where all the math calculations are done to get as correct as possible output. Figure 3 shows the general structure of layers. It also shows the idea of fully connected neural networks where all the neurons in one layer are connected to all the neurons in the next layer. Sometimes this concept is also called dense layer.

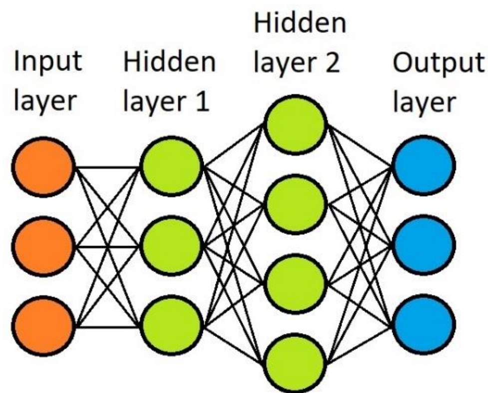

Figure 3. Small neural network.

Every neuron has a weight and bias associated with it. These are parameters that are fine-tuned during the training process until the best weight and bias values are found for every neuron. Every neuron has an output which can be calculated with the following formula: output = sum(weights * inputs) + bias. Where the input is the number from the previous layer. Furthermore, the output is filtered through the activation function. The most used activation function is Rectified Linear Unit activation function, which sets the output to 0 if it was less than 0, otherwise it keeps the same value.

For training a neural network a dataset is needed. When dealing with an audio recognition application where it is required to detect if a person is saying “yes” or no “no” then short audio clips are needed. Usually, the audio clips should be about 1 second long. Clips should be short so that it would be easier to process them. These clips shouldn’t have any background noise and the “yes” or “no” word should be clear. In pre-processing some background noise could be added to simulate the real-life conditions and improve accuracy. In the dataset there should be hundreds of different people saying “yes” and “no”. Same rules apply when dealing with different types of noise such as traffic, industrial, human and aircraft although when dealing with noise recognition then adding additional background noise can be questionable.

A developer must configure the number of epochs and batches before the training process. The number of epochs means how many times the entire dataset is fed through the neural networks during training. However, often the dataset is so large that it's not optimal to pass the entire dataset into the neural network at once. For that reason, the dataset is divided into batches. Each batch consists of some smaller part of the dataset. The number of iterations shows how many batches are needed to complete one epoch [38].

After every iteration a loss function calculates how far off from the actual value the estimation of the neural network was. For example, one of the popular loss functions is mean squared error, which calculates the square of difference between actual and estimated value [39]. Optimizers are algorithms that try to minimize the error from the error function. The optimizer will modify the weights and biases so that the error would decrease. For example, one of the popular optimization algorithms is called Adam [40].

Convolutional neural networks (CNNs) are special types of neural networks that are very popular in computer vision tasks. These networks perform very well with images. However, some research [41], [42], [43] has shown that CNNs show good results for audio classification also. CNNs are usually composed of convolutional layers, pooling layers and fully connected layers. Convolutional layers apply filters to inputs to create a feature map. Pooling layer can be either Max pooling or Average pooling. The goal of the pooling layer is dimensionality reduction, reducing the number of parameters in the input. Although a lot of information is lost in the pooling layer, the gained benefits of reduced complexity, improved efficiency and reduced risk of overfitting are worth it [44]. In many cases a dropout layer is also used to make the model less likely to overfit. It randomly sets input units to 0 [45]. In conclusion the main idea of CNNs is to reduce the images into a form which would be easier to process [46].

When the model is trained it can be converted into TensorFlow Lite (Tflite) format. Tflite helps developers to run models on mobile, embedded and edge devices [47]. The benefits of Tflite are mainly faster inference speed, smaller model size and smaller RAM usage [48]. Furthermore, during the conversion some additional optimizations can be applied. For example, full integer quantization described in [49] is often applied for microcontrollers. It improves latency, reduces peak memory usage and has a compatibility with integer only hardware devices [49]. Quantization can slightly reduce the model accuracy but given that it has many benefits especially for microcontrollers then the trade-off is worth it. Since microcontrollers can’t recognize Tflite format by default, then the easiest way to take the model into use on microcontrollers is to convert the model into a C array [50]. Using the following command: `xxd -i model.tflite > model_array.cc` the Tflite model is converted into a C++ source file. This can be included in the build system and then run with some additional TensorFlow libraries [51].

2.3 Audio pre-processing

It is not a good practice to train a machine learning model that takes raw audio data as an input. If the sampling rate is 44100Hz then the input layer would need to consist of 44100 nodes given that the model takes 1 second audio clip as an input. That number of nodes increases the size of the model, increases RAM and CPU usage. The better way is to convert the 1 second audio clip into a small matrix or an array that represents that audio in a different way, but still keeps the main features of that clip. Converting raw audio into a shorter representation is called audio pre-processing. This conversion involves using complex mathematical functions like Fast Fourier Transform, logarithms and Discrete Cosine Transform. In this paragraph some audio pre-processing algorithms are described.

2.3.1 Signal in time domain

The signal in the time domain is the representation of a signal where on one axis there is time and, on another axis, there is amplitude. In Figure 4 there is a 1 second waveform of a person saying “yes”. In this case for representing 1-second-long audio clip it takes 8000 samples where each sample is 2 bytes. So, in total 16kB is required to store that information. Figure 5 show waveforms of a person saying “no”.

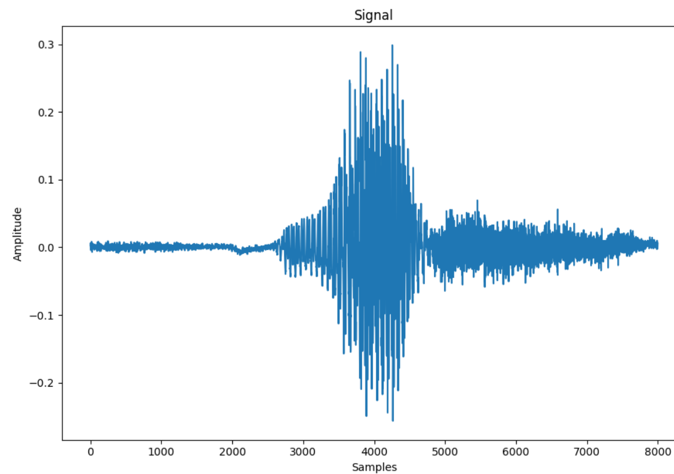

Figure 4. Waveform of a person saying "yes".

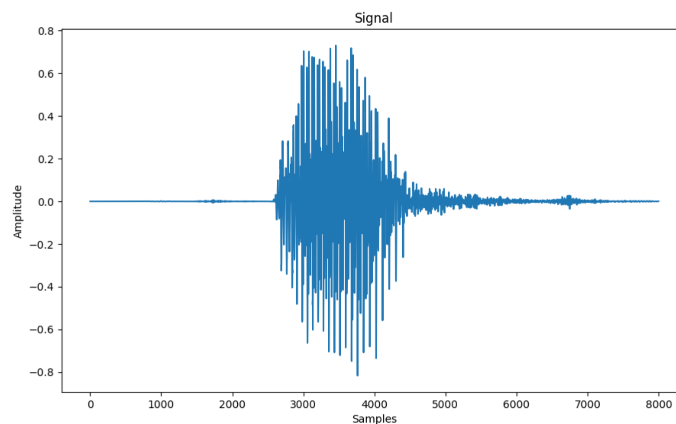

Figure 5. Waveform of a person saying "no".

2.3.2 Discrete Fourier Transform (DFT)

DFT is used to convert a signal from time domain to a frequency domain. This is the discrete version of Fast Fourier Transform (FFT) meaning that DFT is calculated from a digital rather than an analogue signal. DFT produces an array of complex numbers which represent magnitude and phase; however, phase can be discarded since only magnitude is considered to be useful information in audio processing applications. If the output of DFT is plotted, then on one axis there are frequencies between 0 and sampling rate divided by 2. On another axis there is magnitude. Magnitude shows the amount of how much certain frequencies are present in a given audio signal. For example, if magnitude is 100 for 1kHz and 50 for 1.5kHz then it means that in the given audio signal there are twice as much 1kHz frequency sounds as are 1.5kHz sounds.

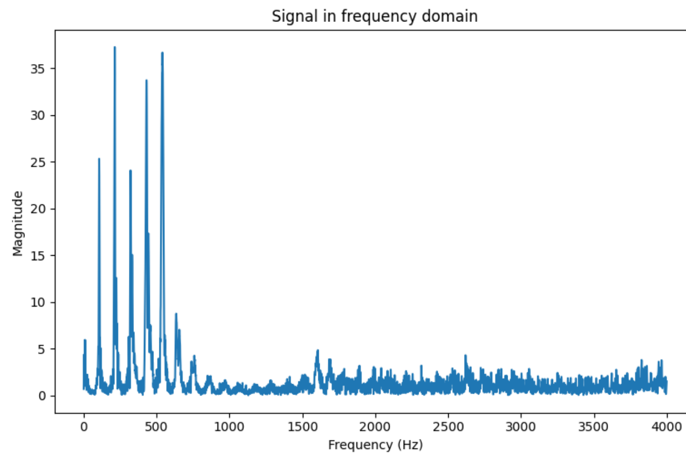

Figure 6. DFT calculated from a waveform of a person saying “yes”.

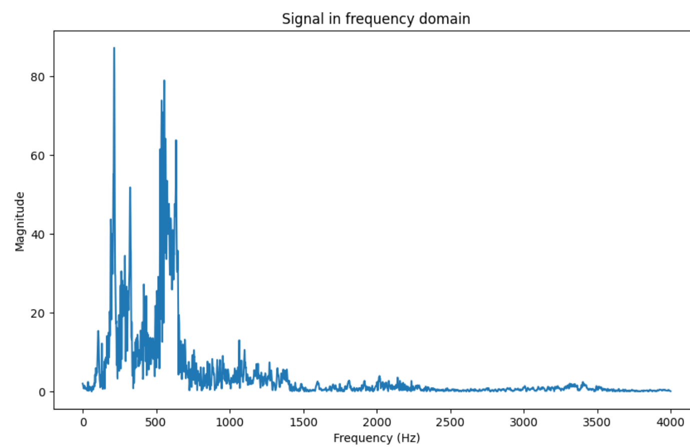

Figure 7. DFT calculated from a waveform of a person saying "no".

Figure 6 and Figure 7 show the same signal as Figure 4 and Figure 5, but in frequency domain. Note that if the sampling rate is 8000Hz then the highest frequency that can be detected is 4000Hz. For that reason, the DFT figures are twice as small as the waveforms, but it still provides some meaningful information about the sound. However, the downside of DFT is that we lose the time component. It is not known when the frequency components are present. For example, from Figure 7 it can be seen that there is a significant amount of 500 Hz sounds, however these sounds could’ve happened in the beginning of the signal, in the middle or in the end. Time domain Figure 5 show that these frequencies were present in the middle of the signal.

2.3.3 Short Time Fourier Transform (STFT)

The purpose of STFT is to calculate the frequency components of a signal over time. The output is a matrix of complex numbers. If squared magnitude is taken from the output, then the result is a spectrogram.

Previously, when DFT was calculated all the samples of a raw signal were given as an input for DFT. In other words, DFT was calculated from the whole signal at once. However, with STFT the signal is split into smaller windows. For example, 1 second signal could be divided into 20 50ms windows. STFT can also be used with overlapping windows. If overlap is configured to be 25ms then this means that adjacent windows have 25ms worth of samples overlapping or in other words, there is 25ms shift for calculating the 50ms window. In this case there would be 40 50ms windows from which the DFT would be calculated. So, if DFT is calculated from every small window separately rather than from the whole audio clip at once then it is called STFT.

When DFT was calculated then the result had frequencies on one axis, where the highest frequency was half the samples that was given as an input for DFT. In the previous case 8000 samples were given as an input and 4000 was the maximum frequency that the DFT could show. The frequency range or in other words frequency bins were from 0 to 4000.

Since STFT uses several smaller windows to calculate the spectrogram then there are fewer frequency bins than there would be if DFT was calculated from the whole signal at once. In case of STFT the amount of frequency bins can be calculated with the following formula:

$$frequency\ bins = \frac{window\ size}{2} + 1.$$

If the sampling rate is 8kHz and the window size is 64ms then there are 512 samples in one window. By applying the formula, there will be 257 frequency bins in total for all the windows and for the end result. These 257 frequency bins represent the frequencies between 0 and sampling rate divided by 2.

Spectrograms have time on one axis, frequencies in another axis and the values in the matrix represent loudness or sometimes the energy of the sound.

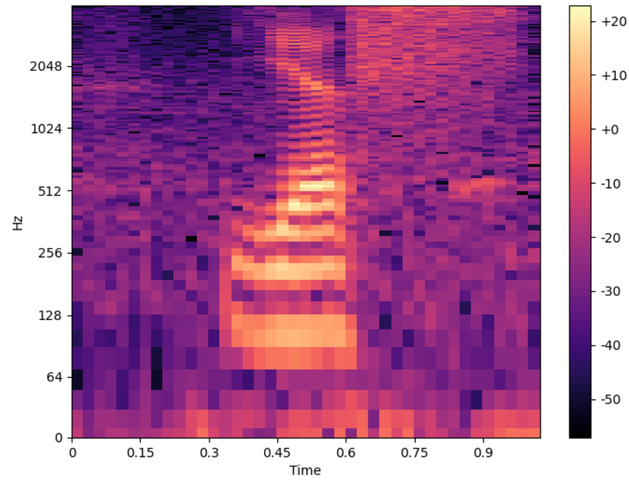

Figure 8. Spectrogram of a person saying "yes".

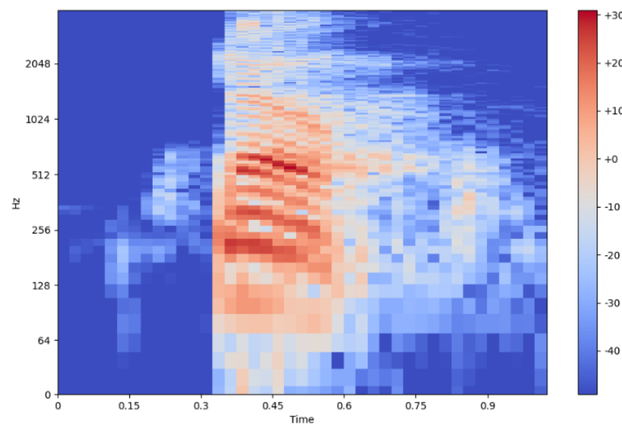

Figure 9. Spectrogram of a person saying "no".

Figure 8 and Figure 9 show that frequencies 256 and 512 are dominant in the middle of a signal. The colour scale shows decibels. The size of this spectrogram is about 16kB; however, it can be optimized a lot by reducing the amount of frequency bins from 257 to 40 for example. By doing so the size of the spectrogram would only be about 3200 bytes.

Spectrograms are used a lot in machine learning applications. Calculating spectrograms is a preferred way of pre-processing the audio. Oftentimes some other pre-processing methods (for example Hann's window) are applied together with STFT to further improve the accuracy of model, inference speed and memory usage. Since spectrograms can be treated as images then combining them with convolutional neural networks (CNN) is a popular practice. The combination of spectrograms and CNNs has shown good results in audio classification applications [42].

2.3.4 Mel spectrograms

It is a known fact that humans perceive sound logarithmically rather than linearly. This means that for example the change of frequency from 100Hz to 200Hz does not sound the same to humans as a change from 500Hz to 600Hz even if the difference is 100Hz for both pairs. The change in the sound is perceived as a ratio of these frequencies. For the first case the ratio is 2.0 and for the second case it is 1.2. People would perceive that the first case had a greater change of frequency [52]. This is something that the basic spectrogram doesn’t take into account.

Mel spectrograms take into account the fact that the sound is perceived logarithmically. Mel is a perceptual scale, where the Mels are calculated from the frequency with the following formula:

$$m = 2595 * \log_{10} 1 + \frac{f}{700} \text{ [53].}$$

The result is a Mel scale unit. For example, the amount of change in frequencies from 100 Mel to 200 Mels sounds the same as a change from 500 Mel to 600 Mels. Figure 10 shows how Mel frequencies translate to regular frequency.

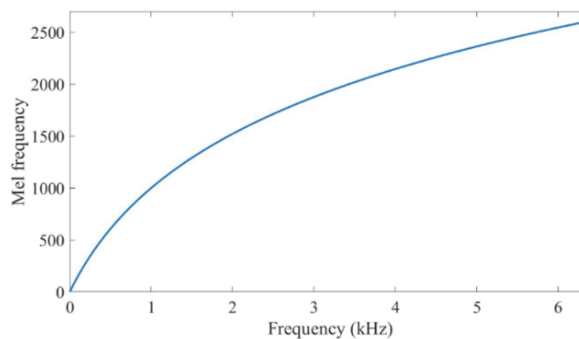

Figure 10. Mel-scale [54].

Mel spectrogram is calculated similarly to the basic spectrogram; however, the frequencies have to be converted to Mel scale. This means that frequency bins will be replaced by the Mel bands. So, frequencies are not represented linearly anymore, but logarithmically. Number of Mel bands must be configured. This could be for example 40, 60, 90, 128 or something similar. It depends on the application and the best number for a specific application can be found by trial and error.

Many sound classification applications use Mel spectrograms rather than ordinary spectrograms for improved accuracy and smaller size [55], [29].

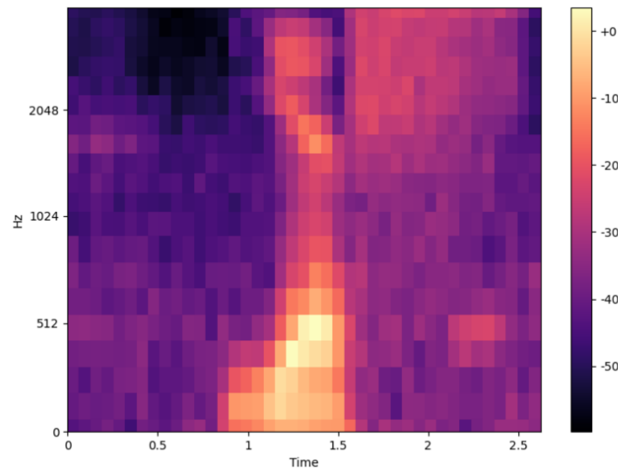

Figure 11. Mel-spectrogram of a person saying "yes".

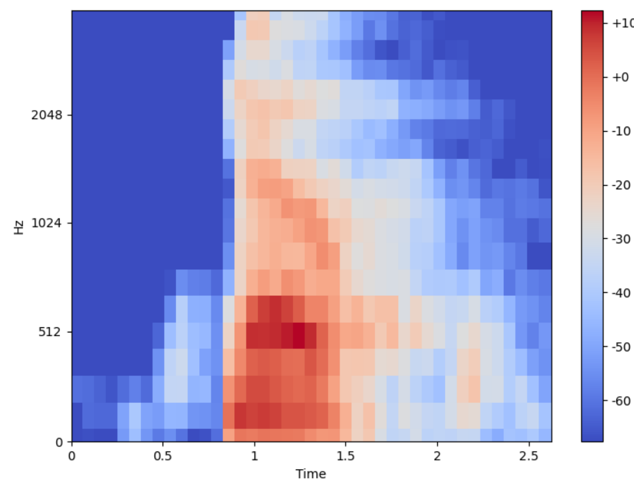

Figure 12. Mel-spectrogram of a person saying "no".

Figure 11 and Figure 12 are Mel spectrograms of the same audio clip as Figure 4 and Figure 5. The number of Mel bands in this case is 20. From the Figure 11 and Figure 12 it can be seen that Mel spectrograms have less noise, and they bring out the relevant features of an audio clip more than the ordinary spectrograms. Also, the size of the image is reduced. Ordinary spectrograms were about 16kB in size (granted the amount of frequency bins could be optimized), however with 20 Mel bands the Mel spectrogram is only 820 bytes. But, calculating the Mel spectrograms introduces some additional computations, which means that calculating it on a microcontroller might be slower than STFT, but using Mel spectrograms can increase the inference speed and reduce the RAM usage as the input to the model would be smaller.

2.3.5 Mel-Frequency Cepstral Coefficient (MFCC)

The motivation for using MFCC is that it's one of the most popular spectral based pre-processing methods used in speech recognition. However, some argue that Mel spectrograms are better in sound recognition applications [29].

Calculating MFCC involves several steps. Firstly, logarithm has to be applied to the output of the DFT to get a log amplitude spectrum. Then Mel-scaling has to be applied as is done for Mel spectrograms. And finally discrete cosine transform (DCT) is applied. DCT is usually used to transform signals from the spatial domain to the frequency domain [56], but in this case of MFCC it is used to convert Mel spectrum into cepstrum. Cepstrum is essentially a spectrum of a spectrum that describes some additional properties of a signal, which are not relevant for the purpose of this thesis.

In MFCC images on the x axis there are time slots and on y axis there are different MFCC coefficients. Most commonly 13 coefficients are used as in the following images. Figure 13 and Figure 14 show MFCC spectrograms of a person saying "yes" and "no".

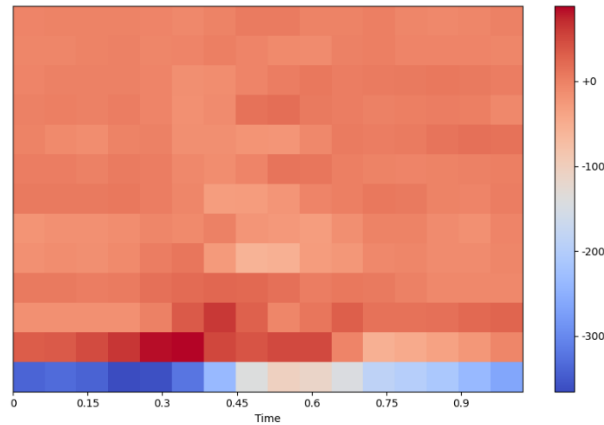

Figure 13. MFCC of a person saying "yes".

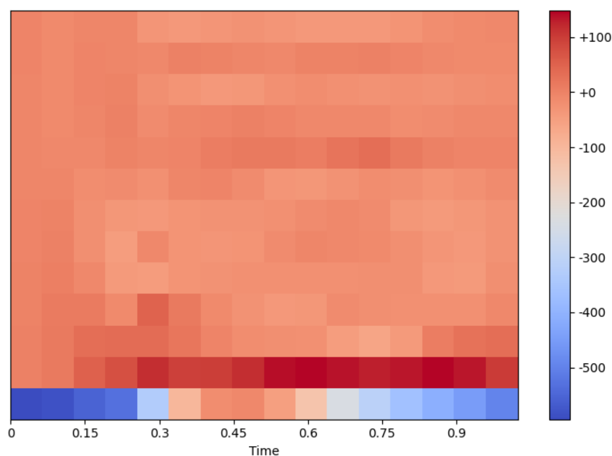

Figure 14. MFCC of a person saying "no".

2.4 TensorFlow Micro Speech

TensorFlow Micro Speech is an example application that demonstrates how to run small machine learning models on different microcontrollers. Official instructions are provided on how to deploy to STM32F747, NXP FRDM K66F and CEVA BX1/SP500. Micro Speech runs a 20kB model that can detect short keywords, for example whether a person is saying „yes“ or „no“ [8].

This application requires a microphone (digital or analogue) and an actuator (for example LED) to indicate the output. Specification says that the Micro Speech code has a small footprint (e.g., 22kB on a Cortex M3) and uses about 10kB of RAM for working memory [8].

Based on the information from the specification and description of the Micro Speech it was deduced that this application can theoretically run on the hardware used in this thesis

(EFR32MG12 [33]). Unfortunately, the official TensorFlow did not provide any instructions or support on how to deploy to Silicon Labs EFR32 microcontroller, but luckily Silicon Labs had forked the TensorFlow GitHub repository and provided a support for EFR32 microcontrollers also.

Silicon Labs extended deployment support and instructions for many other platforms including Arduino, ESP32, Silicon Labs STK3701A, Thunderboard Sense 2 [57], SparkFun Edge and some more. Since Thunderboard Sense 2 (Figure 15Figure 15) uses the same microcontroller as SmENeTe platform then Thunderboard was chosen for doing some initial investigation on how the application behaves on EFR32 microcontroller.

Figure 15. Thunderboard Sense 2.

Thunderboard Sense 2 is a development board with a EFR32MG12 microcontroller that includes different sensors including digital microphone, temperature, humidity, pressure, air quality, gas, 6-axis inertial and hall-effect. It has an ARM Cortex M4 core with an operating frequency of 38.4MHz, 256kB RAM and 1024kB Flash. It also provides a 2.4GHz radio configuration with on-board antenna [57]. For the purposes of this thesis only digital microphone, operating frequency and memory are of interest.

Running Micro Speech on Thunderboard was made simple as Silicon Labs provided a Simplicity Studio project file [58] for the application. Only Simplicity Studio v5 IDE was needed to run Micro Speech. While running the app it was observed from debug log from Putty (a serial port console) that the machine learning model can output a result approximately after every 225ms.

Figure 16. Debug log from running Micro Speech application on Thunderboard Sense 2.

The findings from running Micro Speech on Thunderboard were promising. The debug logs can be seen in Figure 16. One classification after every 225ms was a very good result. The goal for this thesis was to get at least one classification per second with a more complex application.

When the “yes” or “no” word was spoken out then the application was quite quick to respond with the correct classification. The goal for this thesis was to classify noise coming from traffic, pedestrians, industrial activities and aircrafts.

One minor fault with Micro Speech classifications was that during silence the model outputted “unknown” instead of “silence”, which is not an issue, but should be kept in mind when training custom models.

The next step was to integrate the Micro Speech application to the SmENeTe platform. One issue that was noticed was that the Thunderboard Sense 2 had a digital microphone, however the SmENeTe platform had an analogue microphone. Based on that it was inferred that some low-level driver work was on the horizon.

2.4.1 Integrating Micro Speech to SmENeTe platform

Applications on SmENeTe devices have a Makefile for building and flashing the code to a microcontroller. That Makefile consists of various compiler flags, linker flags, configurations, sources and includes. These sources and headers can include low-level peripheral support libraries for EFR32 MCU, radio libraries, FreeRTOS libraries, loggers

and other modules that are necessary for building, running and debugging a MCU. Some of those libraries were part of Thinnect's and Prolab's Git Submodules. Most if not all the sources and includes were C source and header files.

A simple ADC-Stream application [59] was taken as a template for integrating Micro Speech. This template application captures a stream of data from ADC and calculates a signal energy of the measured stream. This application uses ADC and LDMA peripherals and has only one thread for the logic.

Micro Speech application consists of both C and C++ sources. Since C++ sources cannot be compiled with C compiler then it was necessary to compile Micro Speech files with C++ compiler. Rest of the application was compiled using the C compiler. Compiled files were linked together with an object file linker.

The ADC and LDMA drivers for ADC-Stream application were configured so that every second 10000 samples were taken and moved to a buffer for calculating a signal energy. Then ADC was stopped, and application was delayed for 2 seconds. For running Micro Speech this driver setup was not sufficient. Micro Speech requires ADC and LDMA to run continuously without stopping or delaying.

The existing LDMA driver implemented channel descriptors as a linked list. It had 5 descriptors in total as is depicted on Figure 17 and every descriptor transferred 2048 16-bit values from ADC to RAM. When the 5th descriptor had finished transferring then ADC and LDMA were halted until they were restarted again.

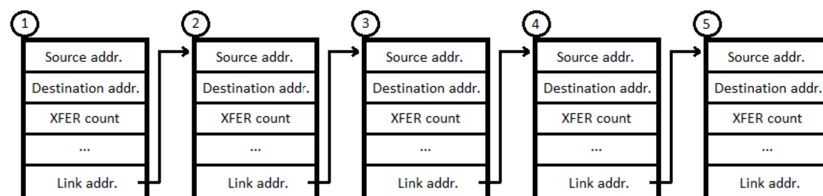

Figure 17. Linked list of channel descriptors.

For adapting channel descriptors to a different application some of the application requirements need to be understood. This is because XFER count, meaning the amount of how many samples a descriptor should transfer depends on the sample rate, and on the window size that is used to calculate a FFT. Furthermore, the size of the buffer, where ADC samples are collected, depends on the sample rate. Also, Micro Speech requires a certain number of samples before it can start audio pre-processing.

The sample rate was 8kHz, which meant that the audio buffer size was 16384 half-word elements. If one FFT window is configured to be 32ms then 256 samples (32ms / 1000ms * 8000 samples) are minimally required so that Micro Speech can start pre-processing the raw audio.

In that case it would be most optimal to set XFER count to 256. So once ADC has converted 256 samples then LDMA will move these to the global audio buffer where all the samples are stored. Then Micro Speech will copy these samples from the audio buffer to another buffer that is used for FFT calculations.

If XFER count is less than FFT window length, then it wouldn't be as optimal. First of all, LDMA would need to initiate more transfers to get the same number of samples. For example, if XFER was 128 then LDMA would need to make 2 transfers to cover 256 samples. Secondly, if there are not enough examples to cover the FFT window length then Micro Speech just returns an error code that there are not enough examples and will loop around again.

If XFER count is more than FFT window length, then there could be data loss if LDMA has managed to transfer more than 1 second worth of audio while the inference is running.

Now if XFER count is set then another important aspect is that the global audio buffer, where the raw audio samples are stored, can store up to 2 seconds of audio. So, if the sample rate is 8kHz then the audio buffer can store 16384 samples. It should be noted that the audio buffer is slightly aligned up. In this case 384 samples are added to 16000. This makes the audio buffer divisible by FFT window length. In that way the audio buffer can fit the exact number of FFT windows. For example, if FFT window length is 256 and audio buffer is 16384 then 16384 divided by 256 is 64. So, each audio buffer can hold 64 FFT windows.

Coming back to the LDMA descriptors. The reason why a linked list of descriptors doesn't work in this case is because it would fill the audio buffer and then ADC would halt. The requirement is that samples must be moved continuously to the audio buffer without stopping. Ring buffer instead of a linked list is a data structure that can be used to satisfy that requirement. Figure 18 shows a ring buffer of channel descriptors.

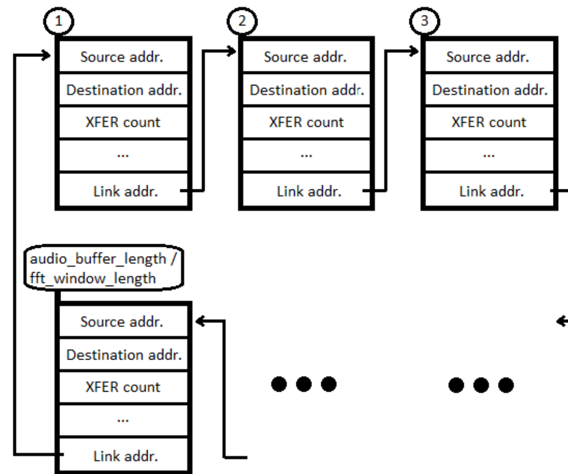

Figure 18. Ring buffer of channel descriptors.

The number of channel descriptors in case of ring buffer is calculated by dividing FFT window length from audio buffer length. So, if the audio buffer is 16384 samples long and FFT buffer is 256 samples long then there would be a need for 64 descriptors. The first descriptor would move 256 samples from LDMA to audio buffer indices [0, 256). The second descriptor would cover indices [256, 512) and so on. Once the final descriptor has finished transferring the data then LDMA will start over from the first descriptor. The data in the audio buffer will then be overwritten. Samples will be overwritten after 2 seconds if the size of the audio buffer is the default 2 seconds chosen by the Micro Speech engineers.

Another aspect that should be considered is that the ADC on EFR32 microcontroller returns a number between 0 and 4095, however the trained model uses WAV files as an input. The data in WAV files are values between -32768 and 32767. This means that the number from the ADC has to be converted to a suitable range. This can be done by subtracting 2048 from the ADC value and multiplying it with 16. The multiplication affects only the representation of the sound so theoretically the multiplication could be omitted or halved.

Essentially that is all that is required to integrate the Micro Speech application to a SmENeTe device. Similar patterns can be followed for integrating to other platforms. However, it should be noted that the developers had used a poor software engineering practice called hard coding. Specifically, one of the important variables should have been calculated using sampling frequency, FFT window length and stride length instead of hard coding it as a magic number of 160.

To reiterate, the samples from LDMA are moved to an audio buffer that can store up to 2 seconds of audio. Now if that buffer contains enough samples to cover at least one FFT window length then they are copied to a FFT input buffer. And then the FFT input buffer is given as an input for different pre-processing calculations.

A one second spectrogram is calculated by taking FFT from several smaller "windows" and concatenating them together. Short Time Fourier Transform (STFT) uses overlapping windows for calculating the spectrogram. Micro Speech takes that into account to slightly optimize the code. At the beginning of the program the very first FFT that is calculated is using the whole FFT input buffer for these calculations. For the next window the FFT input buffer will keep the samples from the previous window that are overlapping with the current window. Also, new samples are copied from the audio buffer to the FFT input buffer. However, if FFT was already calculated on overlapped samples then it can be optimized so that every FFT calculation after the first one will not use the overlapped samples.

Given that sample frequency is 8kHz, FFT window length is 32ms (256 samples) and window stride length is 16ms (128 samples). When the program has been running 32ms then the 256 samples from the audio buffer will be moved to the FFT input buffer. FFT will be calculated, and the results will be stored to the neural network input buffer which when full forms a spectrogram. When the program has been running 48ms then the FFT can be calculated from the second window (16ms to 48ms). The second window starts at 16th millisecond since the window stride was 16ms. It can be seen that the first window and the second window share a common part - samples starting from 16ms to 32ms. Since the first window already calculated the FFT from those samples then there is no need for the second window to do so. For that reason, all the windows after the first window can set the buffer starting location with the following formula:

$$
\begin{aligned}
&FFT\ input\ buffer = \\
&FFT\ input\ buffer + \frac{sample\ frequecny * (FFT\ window\ length - window\ stride\ length)}{1000}
\end{aligned}
$$

This formula calculates the number of samples that are overlapping. In this case the overlapping window is 16ms long which translates to 128 samples. So, if in total there are 256 samples in the FFT input buffer then the first 128 samples can be skipped and only the second half will be used for FFT calculations.

Unfortunately, the developers of Micro Speech did not make this as flexible. They hard coded it so that 160 samples should be skipped no matter what the other parameters are. Originally for Micro Speech the sample rate was 16kHz, window length 30ms and stride length 20ms. With these parameters 160 could be calculated as follows: (16000 * (30 - 20) / 1000). However, they did not make this as apparent. This kind of hard coding makes software less maintainable and extendable. It is difficult to track down why the application is not working as expected when changing the parameters. Finding out what the issue is can cost a lot of time and money so this is something that is strongly recommended to be fixed as soon as possible.

Figure 19 shows a high-level overview of signal processing starting from ADC up until neural network inputs.

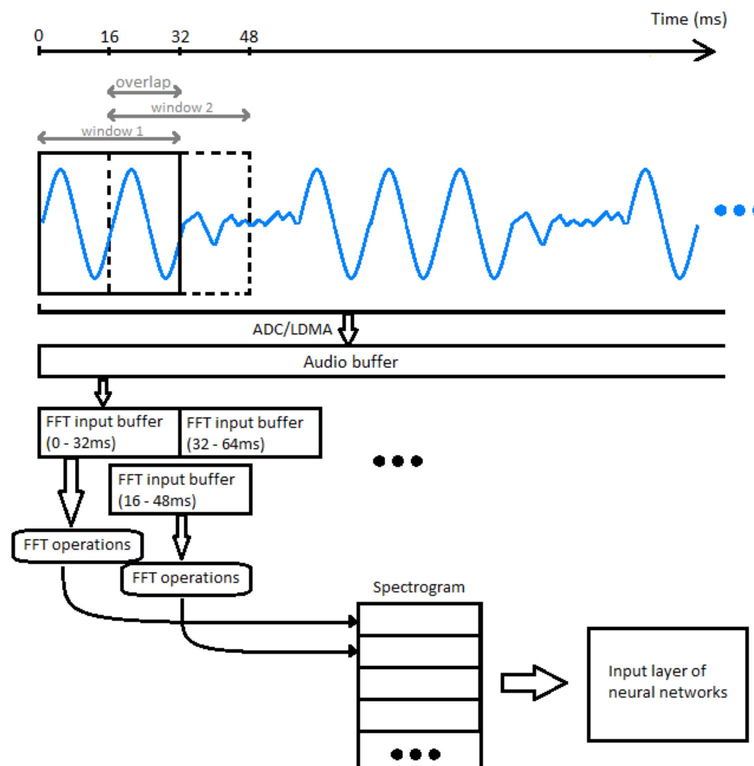

Figure 19. Overview of buffers and windowing

2.4.2 Micro pre-processing

Micro Speech application uses a pre-processing method called Micro pre-processing to convert raw audio to a more suitable format for the input layer of neural networks. Micro pre-processing consists of 6 or 7 steps.

The first step is to filter the input window with the Hann function. Figure 20 shows a Hann function. The Hann or also called Hanning function removes samples at both ends of an input window. Hann's function is used to minimize spectral leakage. Spectral leakage happens when the input window is not represented by an integer number of periods. It has an effect of making high-frequency components appear that are not present in the original signal. For example, if the original signal is between 0 and 2kHz then spectral leakage can cause spectrograms to have values in the 6kHz frequency bin as shown in Figure 21 [60].

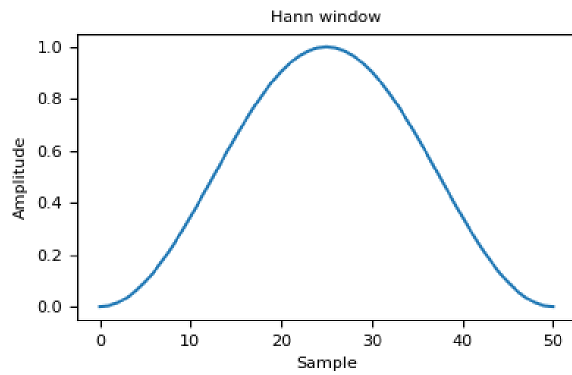

Figure 20. Hann function [61].

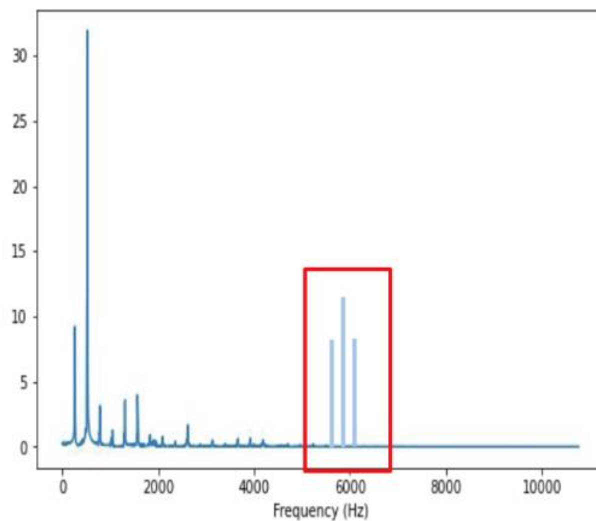

Figure 21. Spectral leakage [60].

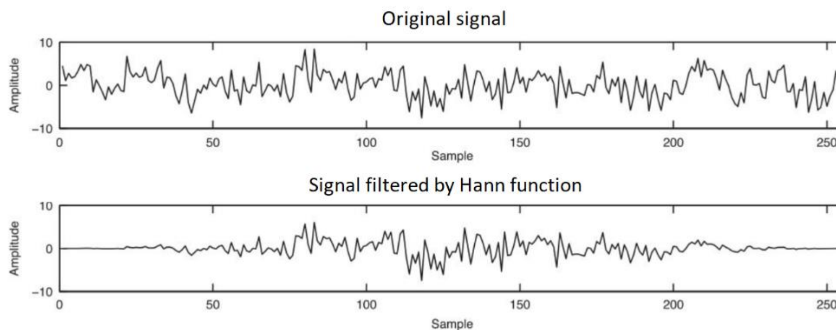

Figure 22. Effect of Hann function [60].

As can be seen from the Figure 22 then there is an issue that some samples will be lost after applying the Hann function. To overcome this overlapping input windows are used as shown in Figure 19.

The second step of Micro pre-processing is calculating the FFT from the signal filtered by the Hann function. The result of this is complex numbers with real and imaginary parts

for every frequency. The third step is calculating the overall energy (computing a modulus or magnitude) by summing the squares of real and imaginary parts.

The fourth step computes the mel-scale filter bank on the given energy array. Then square root is taken from the mel-scale filter bank array.

The last three steps are low pass filter, per-channel amplitude normalization auto-gain and applying of logarithm. These are not as common signal processing methods as the first four steps. The purpose of these steps are described in [9] and [62].

Figure 23 and Figure 24 show Micro spectrogram of a person saying “yes” and “no” respectively.

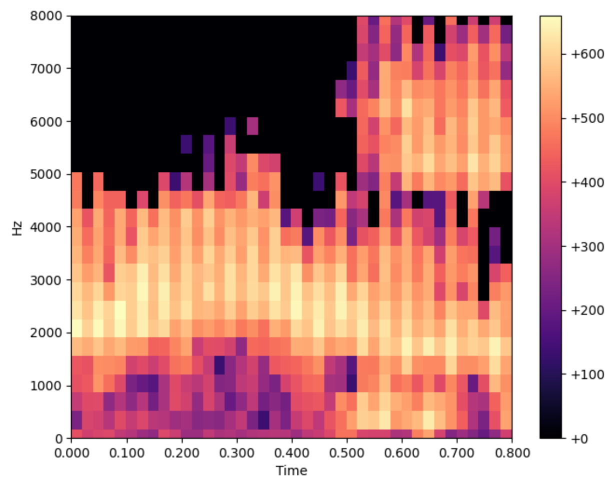

Figure 23. Micro spectrogram of a person saying "yes".

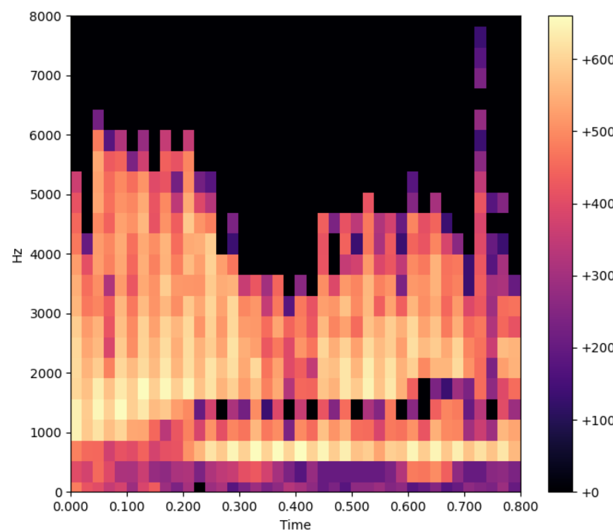

Figure 24. Micro spectrogram of a person saying "no".

2.4.3 Using MFCC instead of Micro pre-processing

As mentioned previously the Micro Speech application uses Micro pre-processing by default. There are no options to use other pre-processing methods on the microcontroller. That means that MFCC has to be manually integrated to the controller. Luckily, TensorFlow provides necessary functions for calculating the MFCC.

The MFCC C++ files have to be integrated into the device's build system. The main file is mfcc.cc and there are mfcc_dct.cc and mfcc_mel_filterbank.cc which are implemented to calculate the specific parts. Furthermore, as MFCC uses a slightly different spectrogram calculation method then the spectrogram calculation needs to be replaced.

While Micro pre-processing needs inputs in the range of [-32768; 32767], the MFCC spectrogram calculation uses normalized values in the range of [-1; 1]. The output is in the range of [-128; 127].

In general, the MFCC calculation consists of 4 steps. The first step is calculating a spectrogram that outputs the sum of squaring real and imaginary parts. The second step is calculating the Mel spectrum from the squared-magnitude FFT input. Thirdly, the logarithm function is applied. And finally, the discrete cosine transform is calculated.

Since the array produced by MFCC has different value ranges than Micro pre-processing then the output normalization on the MCU has to be rewritten. The following formula is used to normalize MFCC output:

$$neural\ network\ input = \frac{mfcc\ output}{input\ scale} + input\ zero\ point$$

The values for input scale and zero point were taken based on the observations in the model training. The formula was derived from [63] and the specific values were observed in the function that runs inferences after the model is trained. Figure 25 and Figure 26 show a TensorFlow MFCC spectrogram of a person saying "yes" and "no" respectively.

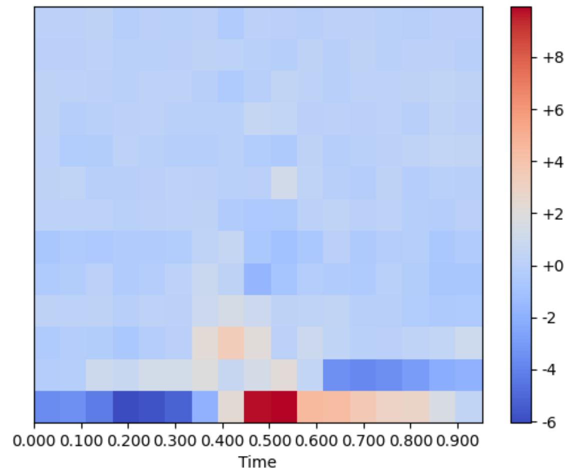

Figure 25. TensorFlow MFCC of a person saying "yes".

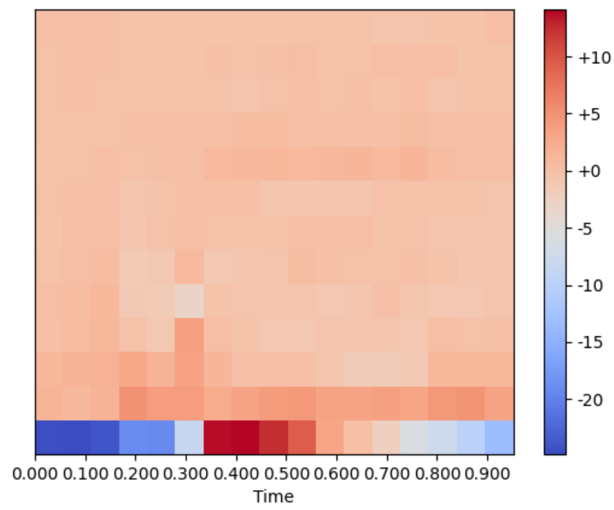

Figure 26. TensorFlow MFCC of a person saying "no".

3 Results and analysis

This chapter describes tests and results of running Micro Speech on the EFR32 microcontroller. Analysis is done for every test.

3.1 Conforming MFCC features

One thing to keep in mind when running neural networks on a microcontroller is that on the MCU the input for the neural network has to be in the same format and the same pre-processing methods have to be used as in model training process. In other words, the audio pre-processing must be synchronized and done in the same way in both places.

TensorFlow provides a script for converting WAV files to input features for neural networks. The script is implemented the same way as is pre-processing in full model training. This script, for example, can convert a 1-second-long WAV file into a representation in MFCC. For testing and porting purposes this script can prove to be useful when comparing the on-device pre-processing implementation against the implementation used in training.

It is not that trivial to test the pre-processing implementation on the MCU. One reason is that MCU is slow and there are no good ways of printing out large matrices for observing the outputs. Fortunately, the same Micro Speech application that is meant to run on the MCU can be run also on PC. The PC version of Micro Speech is mostly meant for testing purposes.

Micro Speech application on a PC does not have a way of getting raw input as MCU with a microphone. For that reason, a WAV file must be converted to a C array of 16-bit values of range [0; 4095]. The ADC on a microcontroller outputs the same range of values. If a sampling rate is 8kHz and the WAV file is 1 second long, then there will be 8000 values in that array. This array can be then connected to Micro Speech pre-processing functionality meaning the values in the array will be given as input to spectrogram calculations.

Once that is done then the output of a pre-processing can easily be written into a file using standard C++ libraries. It can be empirically observed whether the features generated by the TensorFlow script match the features calculated by the implementation done in Micro

Speech PC application. If this is the case, then the same implementation can be used on the MCU also. However, it is important to note that even if the features match on the PC app, the MCU still has its limits, and it might happen that the newly implemented pre-processing might require too much processing power or memory to be used on the MCU.

3.2 MFCC and Micro pre-processing tests and analysis

Some basic tests were run with Micro and MFCC models to see how they behave on the SmENeTe device.

Most of the tests used the same neural network model architecture. Micro Speech application provides 6 premade model architectures in total. They are called "single_fc", "conv", "low_latency_conv", "low_latency_svdf", "tiny_conv" and "tiny_embedding_conv". For initial testing "tiny_conv" was chosen as it is suitable for devices that have limited computational power and memory. It's difficult to produce very accurate results with "tiny_conv", however it can theoretically run on low power microcontrollers without consuming too much resources.

"tiny_conv" consists of an input layer, a 2D convolutional layer, a rectified linear unit, a fully connected layer, a dropout node, a SoftMax function and an output layer. In Figure 27 can be seen an example model that takes 221 elements as an input and outputs an array of 4 elements. Gradient descent optimizer and Sparse SoftMax cross entropy loss function are used. Figure 27 shows the model architecture of "tiny_conv".

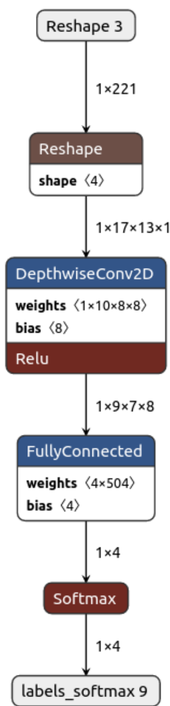

Figure 27. "tiny_conv" model architecture by Netron [64].

The parameters that were used in training are described in the following paragraphs. After the model was trained, it was converted to a C array and taken into use on the microcontroller.

3.3 Tests with "yes" and "no" models

The parameters for the initial tests were mostly the same. Parameters that were changed are described separately under every test description. The dataset was from [65]. Table 1 describes the parameters that were common between all the tests.

Table 1. Initial common parameters.

Sampling rate	8000Hz
Window size	30ms
Window stride	20ms
Input audio	"yes", "no" keywords
Training steps	10000; 3000
Learning rate	0.001; 0.0001
Upper frequency limit	3900
Lower frequency limit	125

Sampling rate was chosen at 8000Hz since one of the goals of this thesis was to classify noise. Noise frequencies are usually quite low, usually up to 4kHz [66], so by considering the Nyquist theorem 8kHz should be the most optimal. The keyword dataset itself consisted of 16kHz audio files which were down sampled to 8kHz. Other parameters were slightly modified or left the same as was by default in Micro Speech.

3.3.1 Test 1

Table 2 describes the parameters that were specific to this test. Table 3 describes the results of this test.

Table 2. Test 1 parameters.

Pre-processing	Micro
Feature bin count	40

Table 3. Test 1 results.

Quantized model (Tflite) accuracy	87%
Quantized model size	18712 bytes
Model (converted to C++ file) as C array size	115kB

Considering that the amount of training steps was low the model accuracy is quite high. The model size is suitable for SmENeTe device microcontrollers, and each classification is made in about 500ms, which is twice as low as the aimed goal of 1000ms.

The accuracy could be improved by fine tuning some of the parameters. For example, training steps could be higher. Or with the sampling rate of 16000Hz the results should be better since humans can detect sounds in a range from 20Hz to 20kHz [67].

Model architecture could be changed. More layers could be added for increasing accuracy, however with the cost of model size and inference speed.

```
07:33:57.487 : D|tensorflow:  29|Microphone Initialized
07:33:58.155 : D|tensorflow:  29|Recognize_commands error
07:33:58.163 : D|tensorflow:  29|silence 0
07:33:58.720 : D|tensorflow:  29|silence 172
07:33:59.257 : D|tensorflow:  29|unknown 145
07:33:59.795 : D|tensorflow:  29|unknown 167
07:34:00.326 : D|tensorflow:  29|unknown 152
07:34:00.843 : D|tensorflow:  29|unknown 153
07:34:01.369 : D|tensorflow:  29|unknown 176
07:34:01.899 : D|tensorflow:  29|unknown 169
07:34:02.425 : D|tensorflow:  29|unknown 145
07:34:02.956 : D|tensorflow:  29|unknown 144
07:34:03.479 : D|tensorflow:  29|unknown 147
07:34:04.010 : D|tensorflow:  29|unknown 147
07:34:04.530 : D|tensorflow:  29|unknown 133
07:34:05.067 : D|tensorflow:  29|unknown 152
07:34:05.588 : D|tensorflow:  29|unknown 188
07:34:06.122 : D|tensorflow:  29|yes 129
07:34:06.646 : D|tensorflow:  29|yes 239
07:34:07.175 : D|tensorflow:  29|yes 152
07:34:07.699 : D|tensorflow:  29|unknown 99
07:34:08.229 : D|tensorflow:  29|unknown 112
07:34:08.752 : D|tensorflow:  29|unknown 107
07:34:09.270 : D|tensorflow:  29|unknown 87
07:34:09.801 : D|tensorflow:  29|unknown 101
07:34:10.328 : D|tensorflow:  29|unknown 128
07:34:10.857 : D|tensorflow:  29|unknown 127
07:34:11.382 : D|tensorflow:  29|unknown 154
07:34:11.914 : D|tensorflow:  29|unknown 169
07:34:12.441 : D|tensorflow:  29|no 151
07:34:12.971 : D|tensorflow:  29|no 243
07:34:13.489 : D|tensorflow:  29|no 132
07:34:14.013 : D|tensorflow:  29|unknown 134
07:34:14.542 : D|tensorflow:  29|unknown 116
07:34:15.066 : D|tensorflow:  29|unknown 107
07:34:15.598 : D|tensorflow:  29|unknown 109
07:34:16.122 : D|tensorflow:  29|no 151
07:34:16.653 : D|tensorflow:  29|no 158
07:34:17.171 : D|tensorflow:  29|unknown 102
07:34:17.707 : D|tensorflow:  29|unknown 149
```


Figure 28. Debug log of test 1 on SmENeTe device.

From the Figure 28 it can be seen that the silence is mostly classified as unknown. However, this is not a problem for the purpose of this thesis as classifying between silence and unknown is not that important. The reason why it classifies silence as unknown could be because in the model training the silence samples are constantly 0. But in reality, there is always some sort of background noise. Furthermore, the dataset for unknown was some random background noise. Also, there might be small measurement inaccuracies coming from the microphone.

When “yes” was said the correct debug message was printed. It should be noted that the model was very confident in classifying that. In the range of 0 to 255 the model's confidence level was 239. The same case was when “no” was said. The model’s confidence level was 243. The range is from 0 to 255, since the probability vector calculated by SoftMax is normalized to range from 0 to 255.

The model made some false classifications by instead of classifying “silence” it classified “no”. However, by looking at the confidence level it can be seen that it wasn’t that confident that it was “no”. The confidence level was only 158. These kinds of false classifications can be avoided by setting a confidence level threshold from which the classifications can be considered as correct. For example, the threshold could be set to 200 meaning that when the confidence level is less than 200 then the classification is not considered and is set to as “unknown”. If the confidence level is 200 then the classification is taken into use by the rest of the application logic. This is something that

could be fine-tuned according to the behaviour and the needs of specific devices and applications.

3.3.2 Test 2

Table 4 describes the parameters that were specific to this test. Table 5 describes the results of this test.

Table 4. Test parameters.

Pre-processing	MFCC
Feature bin count	40
Filter bank channel count	40

Table 5. Test 2 results.

Quantized model accuracy	85.5%
Quantized model size	18712 bytes
Model as C array size	115kB

Unfortunately, with these parameters the MFCC was too slow. One classification could be made every 2.5 seconds which is not acceptable.

To recall the input array for the model or in other words the feature matrix consists of 49 rows and 40 columns with these parameters. Every row represents a 30ms window of samples and since there is 20ms of overlap between the windows then 49 rows make up 1 second of audio. Columns represent the magnitude of frequencies. Lower column indexes represent lower frequencies, higher indexes higher frequencies.

The MFCC for one row (one window of samples) is calculated in approximately 40ms. Compared to Micro pre-processing it takes 35ms longer for MFCC. Therefore, it ran too slow with these parameters. Model inference speed was about 350ms for both cases.

To optimize MFCC calculations the amount columns could be reduced. The following test reduces the number of columns in the feature matrix. Usually, 12 or 13 DCT coefficients are used for calculating MFCC so that can be reduced from 40 to 13.

Furthermore, the amount of filter banks can be reduced since the frequency range this application deals with is not very large.

3.3.3 Test 3

Table 6 describes the parameters specific to this test. Table 7 describes the results of this test.

Table 6. Test 3 parameters.

Pre-processing	MFCC
Feature bin count	13
Filter bank channel count	13

Table 7. Test 3 results.

Quantized model accuracy	86.7%
Quantized model size	8312 bytes
Model as C array size	51kB

With 13 columns instead of 40 in the feature matrix each row is calculated in about 26ms, which is 14ms faster than in test 2. Surprisingly the quantized model accuracy was higher by 1.2%. This can be because the frequency resolution doesn't have to be very precise when dealing with low frequency noise. Also, the accuracy is calculated by running a certain amount of test data through the model. There were only a couple of hundred samples in test data, so this accuracy somewhat depends on how many and what test samples are used.

As the size of this model was more than 2 times smaller the inference speed was also a lot faster. It took 2ms to run an inference. In this case the feature matrix consisted of 637 elements instead of 1960 as in test 2. So, the benefit of this model is that it uses less FLASH and RAM.

It took about 1.4 seconds for one classification, which is still too slow. The MFCC for one row (one window of samples) is calculated in approximately 25ms, which is 15ms

faster than in the previous test, but still too slow. Another thing that could be optimized is the number of rows.

The following test optimizes the number of rows by modifying the duration of window size and window stride length.

3.3.4 Test 4

Table 8 describes the parameters specific to this test. Table 9 describes the results of this test.

Table 8. Test 4 parameters.

Pre-processing	MFCC
Feature bin count	13
Filter bank channel count	13
Window size	64ms
Window stride	56ms

Table 9. Test 4 results.

Quantized model accuracy	84%
Quantized model size	4728 bytes
Model as C array size	29kB

For this test window size was increased from 30ms to 64ms and window stride length from 20ms to 56ms. Previously there was a need for 49 rows in the feature matrix to cover 1 second of audio. With the new window parameters, however, only 17 rows are needed.

Increasing the length of window size and window stride had a minor effect on model accuracy. The accuracy is only 2.7% worse. However, the model size is about 20kB lower, which is significant on microcontrollers.

In this case there were fewer rows from which to calculate MFCC, however each row consists of more ADC samples. In the previous case one row was calculated from 160 ADC samples (20ms window). The fact is that when the program starts the first row is calculated from 240 ADC samples (30ms window). But after that due to the overlapping

windows every row of the feature matrix is calculated from 160 ADC samples. However, in this case one row is calculated from 448 ADC samples (56ms window). More samples means that it takes more time to calculate MFCC.

Each row is calculated in approximately 50ms, which is twice as long as in test 3. However, in test 3 there were 49 rows that needed to be calculated. So, in total it takes approximately 1225ms (49 rows * 25ms) to calculate the full matrix. In this case there are 17 rows so in total it takes approximately 850ms (17 rows * 50ms) to calculate the full feature matrix.

The model inference time was 2ms again. One classification was made in approximately 600ms, which is less than required 1 second so this is a good result. The fact that a full feature matrix is calculated in 850ms, and one classification is made in 600ms makes sense. This is because the full matrix is not calculated for every classification. Some rows are kept in a spectrogram and the rest are replaced by newer ones.

The problem with this model is that it misclassified “unknown” and “silence” as “yes”. It seems to classify “no” correctly, but the confidence level is quite low. When “yes” is said the confidence level is quite high compared to when it is misclassified. Figure 29 shows a debug log of the results.

```
09:37:03.119 : D|tensorflow:  29|yes 131
09:37:03.956 : D|tensorflow:  29|yes 141
09:37:04.784 : D|tensorflow:  29|yes 146
09:37:05.603 : D|tensorflow:  29|yes 123
09:37:06.424 : D|tensorflow:  29|yes 131
09:37:07.135 : D|tensorflow:  29|yes 204
09:37:07.850 : D|tensorflow:  29|yes 190
09:37:08.632 : D|tensorflow:  29|yes 132
09:37:09.364 : D|tensorflow:  29|yes 128
09:37:10.098 : D|tensorflow:  29|yes 130
09:37:10.883 : D|tensorflow:  29|yes 128
09:37:11.581 : D|tensorflow:  29|no 126
09:37:12.305 : D|tensorflow:  29|no 112
09:37:12.984 : D|tensorflow:  29|yes 165
09:37:13.723 : D|tensorflow:  29|yes 126
09:37:14.419 : D|tensorflow:  29|yes 126
09:37:15.080 : D|tensorflow:  29|yes 131
09:37:15.803 : D|tensorflow:  29|yes 108
09:37:16.481 : D|tensorflow:  29|yes 113
```


Figure 29. Debug log of test 4 on SmENeTe device.

Since the aim of this test was to see how MFCC model runs on the microcontroller and not to fine tune the classifications then this misclassification issue can be left unsolved at the moment, but should be kept in mind if similar issue occurs with the noise (traffic, industrial, human, aircraft) dataset.

To verify that the MFCC calculations are working on the microcontroller some WAV files were converted into C array and this array was given as an input for the MFCC functions. Then the output from MFCC was given as input to neural networks and the classifications were empirically observed. In these cases, classifications were correct for the most of the time. This hints that the MFCC was implemented correctly.

Now the question is how these results compare to if Micro pre-processing is used with the same parameters.

3.3.5 Test 5

Table 10 describes the parameters specific to this test. Table 11 describes the results of this test.

Table 10. Test 5 parameters.

Pre-processing	Micro
Feature bin count	13
Window size	64ms
Window stride	56ms

Table 11. Test 5 results.

Quantized model accuracy	84%
Quantized model size	4728 bytes
Model as C array size	29kB

The accuracy and model size are almost identical to test 4. One classification was made in approximately 63ms which is about 15 times faster than in test 4. However, the classifications seemed to be very random and slightly unresponsive to the “yes” keyword. Also, when C arrays converted from WAV files were hard coded then this didn’t seem to have much influence on the classifications. Therefore, this model is not suitable. However, it seems that if this issue could be fixed then Micro pre-processing seems to be a lot faster than MFCC and with more or less the same model accuracy and size.

From all the tests it can be concluded that Micro pre-processing seems to be just as good as MFCC even though Micro pre-processing doesn't have much academic literature backing it up.

3.3.6 Conclusions

For running MFCC suitable parameters seem to be when window size is 64ms, window stride is 56ms and when feature bin count and filter bank channel count are both 13. These counts could maybe be reduced even further. A model with a good dataset and other suitable parameters must be trained. A lot of fine-tuning is needed.

3.4 Tests with "traffic" and "human" models

The following tests are done with traffic and human noise dataset. Noise dataset (traffic, pedestrian, industrial and aircraft noise) is a collection of data from Urban8K dataset [17], Google AudioSet [68], [69] and dataset collected by Tallinn University of Technology. This dataset was not put together by the author of this thesis. All the data is converted into 1 second clips with the frequency of 8000Hz. Table 12 describes the parameters that were common between all the tests.

Table 12. Initial common parameters.

Sampling rate	8000Hz
Input audio	traffic and pedestrian noise
Training steps	10000; 3000
Learning rate	0.001; 0.0001
Upper frequency limit	3900Hz
Lower frequency limit	125Hz

3.4.1 Test 1

Table 13 describes the parameters specific to this test. Table 14 describes the results of this test.

Table 13. Test 1 parameters.

Pre-processing	MFCC
Feature bin count	13
Filter bank channel count	13
Window size	64ms
Window stride	56ms

Table 14. Test 1 results.

Float model accuracy	90%
Quantized model accuracy	81.5%
Quantized model size	4.7kB
Model as C array size	29.2kB

Initially this model was run on a Micro Speech PC application. 3 traffic and 3 human noise WAV files were converted into C arrays and taken into use on the application's build system. These 6 files were not randomly chosen. Inferences were run on those files, in Python using the Tflite model, and the ones that the model classified correctly were chosen. So theoretically the C array model should also classify these 3 traffic files as traffic noise and 3 human files as human noise.

Firstly, it was confirmed if the MFCC produces the same features on the application as the Python scripts that were used for training the model. This was done by writing the features to text files. This test was successful - features were almost identical. Minor +-1 differences were noticed which are caused by calculations with floating point numbers (the floating-point numbers in Python scripts had more precision than the C++ double variables).

Secondly, 3 traffic and 3 human noise C arrays were separately hard coded as an input for the application. The C array model in PC Micro Speech application produced the same results as the Tflite model in Python inference testing. One minor deviation was that the third traffic file was classified as human for the first 500ms worth of samples, but then it switched over to traffic. Based on these results the C array model is behaving similarly

as the Tflite model and strengthens the assumption that the MFCC pre-processing was implemented correctly.

It should be noted that the main loop of Micro Speech application runs very quickly on the PC. For example, in this case the feature matrix (input of the model) has 17 rows and 13 columns, which is actually treated as an array of 221 elements. Where 17 represents the time slots from X + 0ms to X + 1000ms given that X is some time point from the beginning of the program. And 13 columns which represent MFCC coefficients. As was explained previously then each row is calculated from some portion of samples that are usually coming from the microphone, but in this case from a hardcoded array. The way it works on a PC app is that 64ms worth of samples (512 samples) are moved from a hard coded array to the spectrogram input array. When MFCC is calculated from these samples then this row is copied to a feature matrix. When the program first starts then it's copied to the first row of a feature matrix. Then the next row that is calculated is copied to the 2nd row of a feature matrix and so on until the feature matrix is full. When this happens then it starts over. Every time another row is copied to a feature matrix then this same matrix is given as an input for the model. In other words when only one row is updated in the feature matrix then the model is run. So, the model makes a classification every 64ms worth of samples. However, this is not quite how it works on a microcontroller since it's a lot slower than a PC. On a microcontroller there might be several rows copied to a feature matrix before the model is run. Therefore, the hypothesis is that classifications of these 6 hardcoded files on a microcontroller might not be as accurate as were on the PC.

3 traffic and 3 human noise C arrays were hard coded as input for the application on a microcontroller also. The model classified the first human file as a human noise once, but then mostly classified it as a traffic noise. The first traffic file was classified mostly as human, but sometimes correctly as traffic as well. The second human file was classified correctly. The second traffic file is incorrectly classified as human. The third human file was classified correctly. The third traffic file was incorrectly classified as human. These are not great results.

On a microcontroller there is a thread for getting the samples from ADC and moving them to the Micro Speech global audio buffer. And there is a thread for the Micro Speech application itself. Every time when 64ms worth of samples (512 samples) have been

copied from ADC to Micro Speech audio buffer then a call-back function is a called that adds 64 to one of the variables in Micro Speech application. This variable tells how many milliseconds worth of samples are ready to be processed. Also, there is a variable that keeps track of a time when the samples were processed previously. So, by subtracting these two variables the program knows how many samples it has to take from the buffer for processing. The application uses these 2 variables to calculate the positions from which indexes to take the samples for further processing. The main point is that at the same time while the MFCC is calculated, and the model is ran the ADC is constantly producing new samples. So, for example if it takes 500ms to calculate the MFCC and run the model, then there are approximately 500ms worth of samples ready to be processed. These 500ms worth of samples translate to about 8 (17 / 2) rows in the feature matrix. What it means is that theoretically half the rows are thrown out from the feature matrix, the other half is shifted, and then new rows will replace the thrown-out rows. And only then the model is run. So, there will be significant changes to the feature matrix before the model can make a classification. On the PC only one row was updated in the feature matrix before the model made a classification. Therefore, the accuracy on the microcontroller is worse than on a PC or in a Python Tflite inference test. So, in conclusion, the model on a microcontroller can’t make as accurate classification as the model on a PC since the microcontroller is much slower which results in larger changes in the feature matrix which in consequence has a negative effect on accuracy.

```
14:45:39.012 : D|tensorflow:  29|Time: 768
14:45:39.547 : D|tensorflow:  29|83, 15, 11, 13, 9, 15, 11, 15, 18, 18, 17, 12, 14,
14:45:39.553 : D|tensorflow:  29|83, 15, 11, 13, 9, 15, 11, 15, 18, 18, 17, 12, 14,
14:45:39.559 : D|tensorflow:  29|83, 15, 11, 13, 9, 15, 11, 15, 18, 18, 17, 12, 14,
14:45:39.564 : D|tensorflow:  29|83, 15, 11, 13, 9, 15, 11, 15, 18, 18, 17, 12, 14,
14:45:39.570 : D|tensorflow:  29|81, 19, 14, 15, 7, 17, 10, 12, 17, 18, 18, 12, 13,
14:45:39.576 : D|tensorflow:  29|78, 18, 16, 14, 10, 18, 10, 12, 15, 15, 19, 12, 15,
14:45:39.582 : D|tensorflow:  29|74, 16, 13, 15, 10, 19, 9, 15, 17, 18, 18, 13, 14,
14:45:39.588 : D|tensorflow:  29|76, 16, 14, 14, 9, 18, 9, 15, 17, 16, 18, 13, 13,
14:45:39.594 : D|tensorflow:  29|78, 18, 15, 14, 10, 17, 11, 17, 16, 19, 20, 13, 14,
14:45:39.600 : D|tensorflow:  29|75, 16, 16, 17, 9, 16, 9, 15, 16, 17, 20, 15, 13,
14:45:39.606 : D|tensorflow:  29|77, 15, 18, 15, 10, 15, 6, 17, 16, 18, 18, 14, 14,
14:45:39.612 : D|tensorflow:  29|80, 15, 20, 14, 10, 17, 9, 18, 14, 19, 17, 15, 14,
14:45:39.617 : D|tensorflow:  29|84, 15, 12, 13, 9, 15, 10, 15, 18, 19, 17, 12, 14,
14:45:39.625 : D|tensorflow:  29|80, 18, 14, 15, 7, 17, 10, 13, 17, 17, 17, 12, 13,
14:45:39.631 : D|tensorflow:  29|79, 18, 16, 14, 10, 18, 11, 12, 14, 14, 20, 12, 15,
14:45:39.637 : D|tensorflow:  29|74, 17, 14, 15, 10, 18, 9, 15, 18, 18, 18, 13, 14,
14:45:39.643 : D|tensorflow:  29|77, 16, 14, 15, 9, 19, 9, 15, 17, 17, 18, 13, 14,
14:45:39.649 : D|tensorflow:  29|-125 -121 -44 33
14:45:39.651 : D|tensorflow:  29|human 173
14:45:39.654 : D|tensorflow:  29|Time: 1408
14:45:40.150 : D|tensorflow:  29|84, 15, 12, 13, 9, 15, 10, 15, 18, 19, 17, 12, 14,
14:45:40.156 : D|tensorflow:  29|80, 18, 14, 15, 7, 17, 10, 13, 17, 17, 17, 12, 13,
14:45:40.162 : D|tensorflow:  29|79, 18, 16, 14, 10, 18, 11, 12, 14, 14, 20, 12, 15,
14:45:40.168 : D|tensorflow:  29|74, 17, 14, 15, 10, 18, 9, 15, 18, 18, 18, 13, 14,
14:45:40.174 : D|tensorflow:  29|77, 16, 14, 15, 9, 19, 9, 15, 17, 17, 18, 13, 14,
14:45:40.180 : D|tensorflow:  29|78, 18, 14, 14, 10, 17, 11, 17, 16, 19, 20, 13, 14,
14:45:40.186 : D|tensorflow:  29|75, 17, 17, 17, 10, 16, 8, 15, 16, 17, 20, 15, 13,
14:45:40.191 : D|tensorflow:  29|77, 14, 18, 15, 10, 15, 6, 16, 15, 18, 18, 15, 13,
14:45:40.199 : D|tensorflow:  29|81, 15, 19, 14, 10, 17, 9, 19, 14, 19, 17, 15, 14,
14:45:40.205 : D|tensorflow:  29|84, 15, 12, 14, 9, 15, 10, 15, 18, 19, 17, 12, 15,
14:45:40.211 : D|tensorflow:  29|79, 18, 14, 15, 7, 17, 10, 13, 17, 17, 17, 12, 13,
14:45:40.217 : D|tensorflow:  29|80, 18, 17, 15, 10, 18, 11, 12, 13, 14, 20, 12, 15,
14:45:40.222 : D|tensorflow:  29|75, 17, 15, 15, 10, 17, 9, 16, 18, 18, 18, 14, 14,
14:45:40.228 : D|tensorflow:  29|78, 16, 14, 16, 10, 19, 9, 15, 16, 17, 19, 14, 14,
14:45:40.234 : D|tensorflow:  29|78, 17, 14, 14, 9, 16, 11, 16, 16, 19, 20, 13, 13,
14:45:40.240 : D|tensorflow:  29|75, 17, 17, 17, 10, 15, 7, 15, 16, 17, 20, 15, 13,
14:45:40.246 : D|tensorflow:  29|77, 13, 17, 14, 9, 15, 7, 16, 15, 18, 19, 15, 13,
14:45:40.252 : D|tensorflow:  29|-125 -122 -31 22
14:45:40.256 : D|tensorflow:  29|human 155
14:45:40.258 : D|tensorflow:  29|Time: 2048
14:45:40.707 : D|tensorflow:  29|80, 18, 17, 15, 10, 18, 11, 12, 13, 14, 20, 12, 15,
14:45:40.713 : D|tensorflow:  29|75, 17, 15, 15, 10, 17, 9, 16, 18, 18, 18, 14, 14,
14:45:40.719 : D|tensorflow:  29|78, 16, 14, 16, 10, 19, 9, 15, 16, 17, 19, 14, 14,
14:45:40.726 : D|tensorflow:  29|78, 17, 14, 14, 9, 16, 11, 16, 16, 19, 20, 13, 13,
14:45:40.732 : D|tensorflow:  29|75, 17, 17, 17, 10, 15, 7, 15, 16, 17, 20, 15, 13,
14:45:40.738 : D|tensorflow:  29|77, 13, 17, 14, 9, 15, 7, 16, 15, 18, 19, 15, 13,
14:45:40.744 : D|tensorflow:  29|83, 15, 19, 14, 10, 17, 9, 19, 15, 19, 17, 16, 14,
14:45:40.749 : D|tensorflow:  29|85, 15, 12, 14, 9, 15, 10, 14, 19, 19, 18, 12, 15,
14:45:40.755 : D|tensorflow:  29|78, 18, 14, 14, 7, 17, 9, 13, 18, 17, 17, 12, 13,
14:45:40.761 : D|tensorflow:  29|80, 17, 18, 15, 9, 18, 11, 12, 13, 14, 20, 13, 15,
14:45:40.767 : D|tensorflow:  29|74, 17, 16, 16, 10, 16, 8, 16, 18, 17, 18, 14, 14,
14:45:40.775 : D|tensorflow:  29|80, 16, 14, 16, 10, 20, 10, 16, 16, 17, 19, 14, 14,
14:45:40.781 : D|tensorflow:  29|78, 16, 14, 14, 9, 16, 10, 15, 16, 19, 20, 13, 13,
14:45:40.787 : D|tensorflow:  29|74, 17, 18, 18, 10, 15, 7, 15, 15, 17, 20, 15, 13,
14:45:40.793 : D|tensorflow:  29|76, 13, 17, 14, 9, 14, 8, 16, 15, 18, 19, 15, 13,
14:45:40.798 : D|tensorflow:  29|86, 16, 18, 14, 10, 16, 9, 19, 15, 19, 16, 16, 15,
14:45:40.804 : D|tensorflow:  29|85, 16, 12, 14, 9, 16, 10, 14, 19, 19, 18, 12, 15,
14:45:40.810 : D|tensorflow:  29|-124 -120 -27 16
14:45:40.814 : D|tensorflow:  29|human 147
```

Feature matrix

Figure 30. Debug log of feature matrices.

Figure 30 shows the feature matrices that are calculated on a microcontroller. It can be seen that some rows are kept from the previous feature matrix, but most rows are replaced. On a PC only one row would be replaced for every feature matrix, which makes the classification more accurate. It can be seen from the picture that there is about 700ms time difference between when the feature matrices are calculated. In other words, there are 700ms worth of ADC samples ready to be processed at the beginning of every loop.

The feature matrices that were calculated on a PC were hard coded to microcontroller to check if the model on a microcontroller would make a correct classification in that case. It made a correct (same as on the PC app) classification and therefore it can be concluded that the accuracy is worse on the MCU due to low CPU frequency.

Some random human and traffic noise audio found on YouTube were played back through an external speaker to the microphone. Human noise was mostly classified correctly, but some misclassifications. Traffic noise was classified as unknown mostly but was sometimes classified correctly also. Silence was classified as traffic.

3.4.2 Test 2

Table 15 describes the parameters specific to this test. Table 16 describe the results of this test.

Table 15. Test 2 parameters.

Pre-processing	Micro
Feature bin count	20
Window size	50ms
Window stride	40ms

Table 16. Test results.

Float model accuracy	86.6%
Quantized model accuracy	86.6%
Quantized model size	6.6kB
Model as C array size	40.5kB

Running this model with these same 6 hard-coded files resulted in not so good results. One classification was made in approximately 120ms. But traffic files were misclassified as human. Human files were accurately classified as human.

3.4.3 Conclusions

The slow processing speed of microcontrollers may have a negative effect on a classification accuracy. This may be somewhat compensated with a well-trained model. The MFCC implementation a microcontroller seems to be working correctly albeit not very fast which was expected. By fine-tuning and choosing the right parameters it should be possible to run a model that would show good results on classifying noise.

3.5 Tests with "traffic", "industrial" and "human" models

In following tests industrial noise was included in the model. Furthermore, some parameters were fine-tuned in the hope of finding satisfactory results.

The main changes from previous tests were that lower frequency limit was decreased to 25Hz, feature bin count increases to 20 and filter bank channel count was increased to 20.

Furthermore, the amount of training steps was increased significantly as seen from Table 17.

Table 17. Initial common parameters.

Sampling rate	8000Hz
Input audio	Traffic, industrial and human noise
Training steps	40000; 5000
Learning rate	0.001; 0.0001
Upper frequency limit	3900Hz
Lower frequency limit	25Hz

3.5.1 Test 1

Table 18 describes the parameters specific to this test. Table 19 describes the results of this test.

Table 18. Test 1 parameters.

Pre-processing	MFCC
Feature bin count	20
Filter bank channel count	20
Window size	64ms
Window stride	56ms

Table 19. Test 1 results.

Float model accuracy	82.5%
Quantized model accuracy	78.5%
Quantized model size	6.2kB
Model as C array size	39kB

For testing this model 9 mocked audio files were used. 3 mocked files for every noise class. Two human files were classified correctly, 1 was confused with industrial noise.

All the traffic files were confused with industrial noise. Two industrial files were classified correctly, 1 was confused with unknown.

Furthermore, training dataset was played back with external speakers to see if the model can at least classify the training data set correctly. In some previous test random noise clips from YouTube were played back, but that data might have been too specific and not what the dataset used in thesis contains. Also, this test by playing back audio from speakers is not ideal since the same test conditions and results cannot be accurately reproduced or repeated. In [29] they also played back the dataset from external speakers so in this thesis the same method is used.

Figure 31 shows the debug log of when the device was recording silence. It had a very strong confidence that unknown noise was recorded, and sometimes a low confidence that human noise was present. Unknown rather than silence was classified since the silence dataset consisted of data that had amplitude of 0. In real world conditions it is hard to create that kind of environment. The dataset that was provided did not contain background noise or static noise that could've been used.

```
11:43:38.843 : D|tensorflow:  29|Microphone Initialized
11:43:39.717 : D|tensorflow:  29|Recognize_commands error:
11:43:39.725 : D|tensorflow:  29|silence 0
11:43:39.727 : D|tensorflow:  29|Time: 832
11:43:40.468 : D|tensorflow:  29|unknown 118
11:43:40.470 : D|tensorflow:  29|Time: 1600
11:43:41.218 : D|tensorflow:  29|unknown 126
11:43:41.220 : D|tensorflow:  29|Time: 2368
11:43:41.967 : D|tensorflow:  29|unknown 166
11:43:41.969 : D|tensorflow:  29|Time: 3072
11:43:42.619 : D|tensorflow:  29|unknown 232
11:43:42.623 : D|tensorflow:  29|Time: 3712
11:43:43.271 : D|tensorflow:  29|unknown 246
11:43:43.273 : D|tensorflow:  29|Time: 4416
11:43:43.921 : D|tensorflow:  29|unknown 169
11:43:43.925 : D|tensorflow:  29|Time: 5056
11:43:44.574 : D|tensorflow:  29|human 135
11:43:44.576 : D|tensorflow:  29|Time: 5696
11:43:45.177 : D|tensorflow:  29|unknown 171
11:43:45.180 : D|tensorflow:  29|Time: 6272
11:43:45.779 : D|tensorflow:  29|unknown 220
11:43:45.781 : D|tensorflow:  29|Time: 6912
11:43:46.380 : D|tensorflow:  29|unknown 176
11:43:46.384 : D|tensorflow:  29|Time: 7488
11:43:46.940 : D|tensorflow:  29|unknown 159
11:43:46.942 : D|tensorflow:  29|Time: 8064
11:43:47.537 : D|tensorflow:  29|unknown 215
11:43:47.540 : D|tensorflow:  29|Time: 8640
11:43:48.089 : D|tensorflow:  29|unknown 152
11:43:48.093 : D|tensorflow:  29|Time: 9216
11:43:48.643 : D|tensorflow:  29|unknown 153
11:43:48.645 : D|tensorflow:  29|Time: 9792
11:43:49.194 : D|tensorflow:  29|unknown 160
11:43:49.197 : D|tensorflow:  29|Time: 10304
11:43:49.750 : D|tensorflow:  29|unknown 160
11:43:49.751 : D|tensorflow:  29|Time: 10880
```

Figure 31. Test 1 results when the device was in silent environment.

Figure 32 shows the debug log of when pedestrian noise was played back from external speakers to device's microphone. It was quite accurate in classifying human noise, sometimes unknown was classified which is probably caused because there was some silence between the sounds that was played back.

```
12:05:02.249 : D|tensorflow:  29|unknown 211
12:05:02.251 : D|tensorflow:  29|Time: 3072
12:05:02.899 : D|tensorflow:  29|unknown 150
12:05:02.901 : D|tensorflow:  29|Time: 3712
12:05:03.549 : D|tensorflow:  29|human 213
12:05:03.551 : D|tensorflow:  29|Time: 4416
12:05:04.198 : D|tensorflow:  29|human 246
12:05:04.201 : D|tensorflow:  29|Time: 5056
12:05:04.848 : D|tensorflow:  29|human 249
12:05:04.850 : D|tensorflow:  29|Time: 5696
12:05:05.447 : D|tensorflow:  29|human 254
12:05:05.449 : D|tensorflow:  29|Time: 6272
12:05:06.050 : D|tensorflow:  29|human 217
12:05:06.052 : D|tensorflow:  29|Time: 6912
12:05:06.647 : D|tensorflow:  29|human 177
12:05:06.652 : D|tensorflow:  29|Time: 7488
12:05:07.202 : D|tensorflow:  29|human 169
12:05:07.204 : D|tensorflow:  29|Time: 8064
12:05:07.803 : D|tensorflow:  29|human 204
12:05:07.806 : D|tensorflow:  29|Time: 8640
12:05:08.355 : D|tensorflow:  29|human 213
12:05:08.357 : D|tensorflow:  29|Time: 9216
12:05:08.907 : D|tensorflow:  29|human 198
12:05:08.909 : D|tensorflow:  29|Time: 9728
12:05:09.411 : D|tensorflow:  29|human 181
12:05:09.413 : D|tensorflow:  29|Time: 10240
12:05:09.913 : D|tensorflow:  29|human 198
12:05:09.917 : D|tensorflow:  29|Time: 10752
12:05:09.919 : W|main: 272|MESSAGE SENT:0
12:05:09.921 : I|main: 256|RADIO THREAD
12:05:10.465 : D|tensorflow:  29|human 251
12:05:10.469 : D|tensorflow:  29|Time: 11328
```

Figure 32. Test 1 results when human noise was played back.

Figure 33 shows the debug log of when traffic noise was played back. From the observation of the author of this thesis it seemed that sirens and alarms were classified more or less accurately however some engine idling and other not distinguishable traffic noise was more likely to be misclassified for human and unknown.

```
12:06:36.259 : D|tensorflow:  29|human 166
12:06:36.261 : D|tensorflow:  29|Time: 25792
12:06:36.762 : D|tensorflow:  29|human 132
12:06:36.766 : D|tensorflow:  29|Time: 26304
12:06:37.268 : D|tensorflow:  29|traffic 143
12:06:37.270 : D|tensorflow:  29|Time: 26752
12:06:37.720 : D|tensorflow:  29|traffic 95
12:06:37.721 : D|tensorflow:  29|Time: 27200
12:06:38.172 : D|tensorflow:  29|traffic 90
12:06:38.176 : D|tensorflow:  29|Time: 27712
12:06:38.673 : D|tensorflow:  29|human 144
12:06:38.677 : D|tensorflow:  29|Time: 28160
12:06:39.126 : D|tensorflow:  29|human 149
12:06:39.128 : D|tensorflow:  29|Time: 28608
12:06:39.579 : D|tensorflow:  29|traffic 152
12:06:39.583 : D|tensorflow:  29|Time: 29120
12:06:40.131 : D|tensorflow:  29|traffic 175
12:06:40.133 : D|tensorflow:  29|Time: 29632
12:06:40.635 : D|tensorflow:  29|human 161
12:06:40.637 : D|tensorflow:  29|Time: 30144
12:06:41.137 : D|tensorflow:  29|traffic 150
12:06:41.140 : D|tensorflow:  29|Time: 30656
12:06:41.143 : W|main: 272|MESSAGE SENT:2
12:06:41.146 : I|main: 256|RADIO THREAD
12:06:41.641 : D|tensorflow:  29|traffic 249
12:06:41.644 : D|tensorflow:  29|Time: 31168
12:06:42.144 : D|tensorflow:  29|traffic 123
12:06:42.147 : D|tensorflow:  29|Time: 31680
12:06:42.646 : D|tensorflow:  29|human 134
12:06:42.648 : D|tensorflow:  29|Time: 32128
12:06:43.100 : D|tensorflow:  29|human 121
12:06:43.103 : D|tensorflow:  29|Time: 32640
12:06:43.604 : D|tensorflow:  29|traffic 123
12:06:43.606 : D|tensorflow:  29|Time: 33088
12:06:44.057 : D|tensorflow:  29|traffic 118
12:06:44.061 : D|tensorflow:  29|Time: 33536
12:06:44.512 : D|tensorflow:  29|traffic 137
12:06:44.515 : D|tensorflow:  29|Time: 34048
12:06:45.061 : D|tensorflow:  29|traffic 149
12:06:45.065 : D|tensorflow:  29|Time: 34560
12:06:45.562 : D|tensorflow:  29|traffic 216
```

Figure 33. Test 1 results when traffic noise was played back.

Figure 34 shows a debug log of when industrial noise was played back. It sometimes classified industrial noise correctly, but often confused it with human or unknown.

```
12:13:08.017 : D|tensorflow:  29|Time: 23680
12:13:08.517 : D|tensorflow:  29|human 124
12:13:08.519 : D|tensorflow:  29|Time: 24128
12:13:08.969 : D|tensorflow:  29|human 150
12:13:08.971 : D|tensorflow:  29|Time: 24576
12:13:09.423 : D|tensorflow:  29|human 199
12:13:09.425 : D|tensorflow:  29|Time: 25088
12:13:09.977 : D|tensorflow:  29|human 176
12:13:09.978 : D|tensorflow:  29|Time: 25600
12:13:10.475 : D|tensorflow:  29|industrial 119
12:13:10.479 : D|tensorflow:  29|Time: 26112
12:13:10.976 : D|tensorflow:  29|human 121
12:13:10.983 : D|tensorflow:  29|Time: 26624
12:13:11.479 : D|tensorflow:  29|human 142
12:13:11.482 : D|tensorflow:  29|Time: 27136
12:13:11.983 : D|tensorflow:  29|human 171
12:13:11.985 : D|tensorflow:  29|Time: 27584
12:13:12.439 : D|tensorflow:  29|industrial 123
12:13:12.441 : D|tensorflow:  29|Time: 28096
12:13:12.940 : D|tensorflow:  29|human 90
12:13:12.942 : D|tensorflow:  29|Time: 28544
12:13:13.394 : D|tensorflow:  29|industrial 159
12:13:13.397 : D|tensorflow:  29|Time: 28992
12:13:13.845 : D|tensorflow:  29|industrial 112
12:13:13.849 : D|tensorflow:  29|Time: 29504
12:13:14.347 : D|tensorflow:  29|industrial 136
12:13:14.350 : D|tensorflow:  29|Time: 29952
12:13:14.799 : D|tensorflow:  29|human 130
12:13:14.802 : D|tensorflow:  29|Time: 30400
12:13:15.251 : D|tensorflow:  29|human 151
12:13:15.256 : D|tensorflow:  29|Time: 30912
```

Figure 34. Test 1 results when industrial noise was played back.

This test showed a lot better results than the previous tests. Most likely since the amount of training steps was significantly increased and some other parameters fine-tuned slightly.

3.5.2 Test 2

A model with same parameters as in this test, but with “low latency conv” model architecture was trained. It had a quantized accuracy of 77.5%. The size of the model was 2.2MB which is too large to fit this microcontroller. Therefore, “low latency conv” and “conv” architectures are not suitable to be running on this microcontroller.

“tiny embedding conv” model architecture was also used with the same parameters. It had the same accuracies as “tiny conv” model in test 1, however the model sizes were slightly larger. Figure 35 shows the architecture of “tiny embedding conv” model. Compared to “tiny conv” (Figure 27) the “tiny embedding conv” model has an additional convolutional layer.

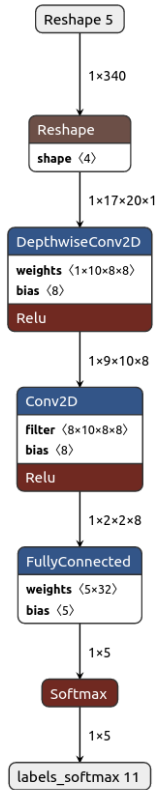

Figure 35. "tiny embedding conv" model architecture.

Convolutional 2D network operations had to be included on the microcontroller in order to run that model. By playing back some sound from external microphone the results were not any better than in test 1.

3.5.3 Test 3

This is a test with Micro pre-processing to get an idea of how it behaves compared to test 1. Since Micro pre-processing is faster than MFCC then shorter window size and window stride were chosen. Table 20 describes the parameters specific to this test. Table 21 describes the results of this test.

Table 20. Test 3 parameters.

Pre-processing	Micro
Feature bin count	20
Window size	32
Window stride	16

Table 21. Test 3 results.

Float model accuracy	79.2%
Quantized model accuracy	77.5%
Quantized model size	15kB
Model as C array size	93.3kB

The accuracy was more or less the same as in test 1, but model size is slightly larger since window size and stride were lowered.

For testing this model 9 mocked audio files were used. 3 mocked files for every noise class. Human files were classified correctly. Traffic and industrial files were mostly confused with human.

This model showed a lot worse results than the MFCC model in test 1 even though the model size was larger and window lengths shorter. Therefore, possibly MFCC would be preferred algorithm for pre-processing this given dataset.

3.6 Test with “traffic”, “industrial”, “aircraft” and “human” model

Finally, the model with all the noise classes was trained. Parameters were the same as in Test 1 on page 66, but aircraft noise was included in the training data. Table 22 describes the results of this test.

Table 22. "traffic", "industrial", "aircraft" and "human" model results.

Float model accuracy	78.4%
Quantized model accuracy	67.9%
Quantized model size	6.9kB
Model as C array size	43.5kB

For testing this model 12 mocked audio files were used. 3 mocked files for every noise class. 2 human files were confused with industrial and 1 human file was classified correctly. Traffic files were confused with aircraft and industrial. 1 industrial file was classified correctly other 2 were confused with unknown and aircraft. It is obvious that these are terrible results, which are expected since the quantized accuracy was only 67%.

3.7 Tests with alternate dataset

Another dataset was provided by a student who collected and prepared their own dataset [70]. They collected vehicle, industrial and aircraft noise. Creating and processing the dataset was not the aim of this thesis, but the author decided to combine old and new datasets and clean it by removing some data that was unnecessary or not satisfactory in terms of quality in author's opinion.

The author tried to keep and use data that the device would most likely record in the environment where the device will be deployed. For example, there is no need for aircraft noise that is recorded next to aircraft turbine since the devices will not be deployed to the airport. The most common scenario would be that the plane flies over the device several hundred meters or a couple of kilometres up in the sky.

It was noted that the human data consisted mostly of children playing and shouting. The dataset should be improved by adding more diverse pedestrian noise for example adults shouting, a group of people talking etc. Some human data resembled more to traffic rather than human therefore such data was removed.

Furthermore, sometimes traffic and industrial noise can be very similar or even unexchangeable. For example, there can be heavy trucks and such vehicles operating at the construction site. That kind of data could belong both to industrial and traffic noise category, however when creating a dataset for machine learning purposes that doesn't make sense. Therefore, the author removed some of the data from industrial noise category that could potentially be confused with traffic. Most of the industrial dataset now consisted of drilling, sawing, some knocking etc. Traffic dataset consisted of sirens, vehicles passing by and engines idling.

For testing the models, 3 files from each category were chosen. The author tried to choose relatively different sounds from each category. Files were converted into C arrays that were used for audio input on a microcontroller.

3.7.1 Test 1 with "traffic", "industrial" and "human" model

This test uses the same parameters as a test in paragraph 3.5.1. Only the dataset is different. Table 23 describes the results of this test.

Table 23. Test 1 results.

Float model accuracy	91.3%
Quantized model accuracy	86.8%
Quantized model size	6.9kB
Model as C array size	42kB

With this new dataset the accuracy improved about 8% as can be seen from Table 23.

The result of running mocked files on a microcontroller were more or less good. Two out of three human files were classified correctly, one was confused with industrial. Two out of three traffic files were confused with industrial noise, one was classified correctly. Two out of three industrial files were classified correctly, one was confused with human.

In comparison when inferences were run using the Tflite model then all the traffic, industrial and human data were classified correctly. Furthermore, when running the same mocked data on PC Micro Speech application then all the data was also classified correctly. Therefore, the fact that MCU has a low processing speed affects the accuracy of the results as explained in paragraph 3.4.1. One thing that could be tried to mitigate this is to use lower filter bank and feature bin count so that pre-processing would be faster, however by reducing these parameters the accuracy of quantized model will probably decrease.

Training dataset was also played back to the microphone via the external speaker just to see what happens. The model classified silence as human, but with a confidence level of about 130. When human noise was played back then the model classified this as human, but this time the confidence level was 250. When traffic data was played back then some traffic was classified correctly, and some was confused with human. When industrial noise was played back again some was confused with human and some was classified correctly.

The next test sets filter bank and feature bin count to 13 to see if that improves the result.

3.7.2 Test 2 with "traffic", "industrial" and "human" model

This test uses the same dataset and mostly the same parameters as Test 1 with "traffic", "industrial" and "human" model but filter bank channel count and feature bin count are now reduced to 13. Table 24 describes the results of this test.

Table 24. Test 2 results.

Float model accuracy	83%
Quantized model accuracy	76.7%
Quantized model size	6.9kB
Model as C array size	43.5kB

As was expected the accuracy is slightly worse than in Test 1 with "traffic", "industrial" and "human" model. In previous test it took approximately 650ms before a classification result was calculated. In this test it was about 400ms.

On a microcontroller the first human mock file was classified correctly, but sometimes it confused it with industrial. The second got confused with industrial noise all the time and the third was classified correctly. The first traffic file was confused with unknown. The second was confused with industrial noise. The third was classified correctly. The first industrial file was classified correctly, but sometimes got confused with human. The second was confused with human. And the third was classified correctly with some minor confusions.

In comparison when inferences were run with Tflite model then all the data was classified correctly. On a PC Micro Speech all the mocked data was also classified correctly. Therefore, this optimization of reducing filter bank channel count and feature bin count was not enough to get better results on a microcontroller.

It would be interesting to see what the results would be if it was running on a more powerful microcontroller. Theoretically it should be more accurate, especially if window size and stride length can be shortened.

3.7.3 Test 3 with “traffic”, “industrial”, “aircraft” and “human”

This test uses the same parameters and dataset as Test 1 with “traffic”, “industrial” and “human” model but this time aircraft noise was also included. Table 25 describes the results of this test.

Table 25. Test 3 results.

Float model accuracy	87%
Quantized model accuracy	79.5%
Quantized model size	5.2kB
Model as C array size	32.3kB

On a microcontroller 2 out of 3 human mock files were classified correctly, one of them got confused with industrial noise. One out of three traffic files were classified correctly, other 2 were confused with aircraft and industrial noise. Two industrial files were confused with human, one was classified correctly. One aircraft mock file was classified correctly, another had minor confusion with industrial noise and third got totally confused with industrial noise.

When inferences were run with Tflite model then human, traffic and aircraft files were classified correctly. Two out of three industrial files were classified correctly, one was confused with human. Since quantized accuracy was 80% then 1-2 misclassifications would be expected. Exactly the same results were observed on a PC Micro Speech application. The reason why microcontroller produced worse results is mentioned in 3.7.1.

3.8 Conclusions

In general, MFCC showed better results than Micro pre-processing. It might be because in author’s opinion Micro is most likely optimized and used for pre-processing human speech. The opinion is based on the fact that Micro Speech was originally meant for detecting keywords like “yes”, “no”, “dog” etc. And the default pre-processing algorithm for Micro Speech is Micro pre-processing. However, Micro would be a good candidate for pre-processing noise data as well since this algorithm seems to be a lot faster than MFCC.

MFCC can be used on a low power microcontrollers for pre-processing. However, the parameters must be chosen carefully. Based on the test then window size of 64ms and window stride 56ms seemed to be most optimal. Feature bin count and filter bank channel count could be 20 or 13, but not 40 since it will be too slow.

It turned out that even if quantized model accuracy is excellent the results on a microcontroller might not be as good. Since pre-processing and running an inference on a microcontroller takes more time than on more powerful devices then it will lose some accuracy. This comes from the fact that the feature matrix is updated in bigger chunks before every classification. For example, on a PC Micro Speech application, only one row was updated in feature matrix before each classification. This gave very accurate results with mocked noise data. It had the same results when inferences where run on Tflite model. As a side note this also confirms that drivers and MFCC were implemented correctly. However, on a microcontroller multiple feature matrix rows are updated before every classification. This is because a lot of new data is sampled while pre-processing and inference is running. This can be somewhat mitigated by cleverly fine-tuning some parameters or by using slightly more powerful microcontroller. Of course, when using a more powerful microcontroller then the trade-off is that the power usage will be higher.

With “traffic”, “industrial” and “human” model the best quantized model accuracy was 86.8%. With “traffic”, “industrial”, “aircraft” and “human” model the best quantized model accuracy was 79.5%. The FLASH and RAM usage were small enough for a microcontroller used in this thesis.

Theoretically if these models were run on a microcontroller with a better CPU, then the results would be better.

3.9 The effect of running neural networks with radio thread

There are 3 threads running in total. The first thread is initializing the ADC and LDMA. Then it endlessly copies the data received from ADC to the Micro Speech audio buffer. The second thread is for the Micro Speech application that pre-processes the audio available in the audio buffer. The result of pre-processing is given as an input for the machine learning model that classifies what type of noise was recorded. Finally, it calculates the statistics of classifications. The third thread is a radio thread that is woken

up by the Micro Speech thread when 10 seconds of statistics have been recorded. These statistics are then sent out via the radio. The radio thread is then suspended until statistics of next 10 seconds have been calculated.

The model from paragraph 3.7.1 was taken to see how much effect radio thread has on Micro Speech thread. Without the radio each classification is made in approximately 650ms. The effect of calculating statistics and sending the data out via the radio every 10 seconds is negligible in terms of CPU usage. Power usage was not measured.

4 Summary

This thesis aimed to develop and test an application that could provide information on what type of noise was measured in the environment. The research problem was how to implement a solution to a low power microcontroller that can run neural networks with MFCC pre-processing without any AI accelerators. The goal was to integrate TensorFlow Micro Speech application to a microcontroller and implement drivers and MFCC pre-processing so that microcontroller could classify whether traffic, industrial, aircraft or human noise was measured.

MFCC proved to be a more suitable than Micro pre-processing. It turned out that MFCC can be run on a low power microcontroller, but with somewhat optimized parameters and some drawbacks. For “traffic”, “industrial” and “human” model the best result for quantized model was accuracy of 86.8%, and with aircraft included it was 79.5%. Having a radio thread that sends data out every 10 seconds had neglible impact on CPU usage.

Even though the quantized model had very high accuracy, the low processing speed of a microcontroller can reduce the accuracy. This was because the input matrix is updated in big chunks between every classification. To remedy this a CPU with higher frequency could be used, but with a higher power consumption as a trade-off. Therefore, it can be concluded that noise recognition with MFCC can be run on a low power microcontroller with radio for transmitting the data, but CPU of 40 MHz seems to be slightly too low or barely on the border of the possibility of getting acceptable results.

References

[1] L. Goines and L. Hagler, “Noise pollution: a modem plague,” *South Med J,* vol. 100, p. 287–94, 2007.

[2] S. A. Stansfeld and M. P. Matheson, “Noise pollution: non-auditory effects on health,” *British Medical Bulletin,* vol. 68, pp. 243-257, December 2003.

[3] J. Kosowatz, “Top 10 Growing Smart Cities,” The American Society of Mechanical Engineers, 3 February 2020. [Online]. Available: https://www.asme.org/topics-resources/content/top-10-growing-smart-cities. [Accessed 4 April 2022].

[4] C. Mydlarz, M. Sharma, Y. Lockerman, B. Steers, C. Silva and J. P. Bello, “The life of a New York City noise sensor network,” *Sensors,* vol. 19, p. 1415, 2019.

[5] “Taltech and Thinnect are building world´s largest smart city sensor network in Tallinn,” Tallinn, 11 December 2019. [Online]. Available: https://www.tallinn.ee/eng/Uudis-Taltech-and-Thinnect-are-building-world-s-largest-smart-city-sensor-network-in-Tallinn. [Accessed 4 April 2022].

[6] “Intelligentsed Targa Linna ja Kriitilise Infrastruktuuri Toimepidevuse Tehnoloogiad II,” Eesti Teadusinfosüsteem, 1 October 2020. [Online]. Available: https://www.etis.ee/Portal/Projects/Display/96569b7f-a42a-422f-bd8f-97cbb3a483b1. [Accessed 4 April 2022].

[7] Thinnect, “SMENETE – järgmise põlvkonna võrguühendus,” Thinnect, 1 January 2022. [Online]. Available: https://thinnect.com/et/smenete-jargmise-polvkonna-vorguuhendus/. [Accessed 4 April 2022].

[8] TensorFlow, “Micro Speech,” TensorFlow, 2021. [Online]. Available: https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech. [Accessed 5 April 2022].

[9] P. Warden and D. Situnayake, TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers, O'Reilly, 2020.

[10] M. Shafique, T. Theocharides, V. J. Reddy and B. Murmann, “TinyML: Current Progress, Research Challenges, and Future Roadmap,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[11] S. O. Ooko, M. M. Ogore, J. Nsenga and M. Zennaro, “TinyML in Africa: Opportunities and Challenges,” in *2021 IEEE Globecom Workshops (GC Wkshps)*, 2021.

[12] A. Benhamida, A. R. Várkonyi-Kóczy and M. Kozlovszky, “Traffic Signs Recognition in a mobile-based application using TensorFlow and Transfer Learning technics,” in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, 2020.

[13] J. Dai, “Real-time and accurate object detection on edge device with TensorFlow Lite,” *Journal of Physics: Conference Series,* vol. 1651, p. 012114, November 2020.

[14] N. C. A. Sallang, M. T. Islam, M. S. Islam and H. Arshad, “A CNN-Based Smart Waste Management System Using TensorFlow Lite and LoRa-GPS Shield in Internet of Things Environment,” *IEEE Access,* vol. 9, p. 153560–153574, 2021.

[15] Tensorflow, “Sound classification with YAMNet,” Google, 2022. [Online]. Available: https://www.tensorflow.org/hub/tutorials/yamnet. [Accessed 29 April 2022].

[16] E. Fonseca, X. Favory, J. Pons, F. Font and X. Serra, *FSD50K,* Zenodo, 2020.

[17] J. Salamon, C. Jacoby and J. P. Bello, “A Dataset and Taxonomy for Urban Sound Research,” in *22nd ACM International Conference on Multimedia (ACM-MM'14)*, Orlando, 2014.

[18] C. Malmberg, *Real-time Audio Classification onan Edge Device : Using YAMNet and TensorFlow Lite,* 2021, p. 39.

[19] A. Mesaros, “DCASE 2016,” 2016. [Online]. Available: https://paperswithcode.com/dataset/dcase-2016. [Accessed 29 April 2022].

[20] K. J. Piczak, “ESC: Dataset for Environmental Sound Classification,” in *Proceedings of the 23rd Annual ACM Conference on Multimedia*, Brisbane, 2015.

[21] D. Elliott, E. Martino, C. E. Otero, A. Smith, A. M. Peter, B. Luchterhand, E. Lam and S. Leung, “Cyber-physical analytics: Environmental sound classification at the edge,” in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020.

[22] M. J. Baucas and P. Spachos, “Using cloud and fog computing for large scale IoT-based urban sound classification,” *Simulation Modelling Practice and Theory,* vol. 101, p. 102013, 2020.

[23] Y. Alsouda, S. Pllana and A. Kurti, “A machine learning driven IoT solution for noise classification in smart cities,” *arXiv preprint arXiv:1809.00238,* 2018.

[24] Dublin City, “Realtime Sound Monitoring Network,” 2017. [Online]. Available: https://www.dublincity.ie/residential/environment/role-air-quality-monitoring-and-noise-control-unit/dublin-city-noise-maps/real-time-noise-monitoring-dublin. [Accessed 22 april 2022].

[25] Dublin City, “Making a Noise Pollution Complaint,” 2022. [Online]. Available: https://www.dublincity.ie/residential/environment/air-quality-monitoring-and-noise-control-unit/making-noise-pollution-complaint. [Accessed 22 April 2022].

[26] Traffic Noise & Air Quality Unit, “DUBLIN CITY COUNCIL AMBIENT SOUND MONITORING NETWORK,” Dublin City Council, 2017. [Online]. Available: https://www.dublincity.ie/sites/default/files/media/file-uploads/2018-07/AMBIENT_SOUND_MONITORING_NETWORK_ANNUAL_REPORT_2017.pdf. [Accessed 22 April 2022].

[27] J. C. Farrés and J. C. Novas, “Issues and challenges to improve the Barcelona Noise Monitoring Network,” in *Proceedings of the 11th European Congress and Exposition on Noise Control Engineering, Heraklion, Crete, Greece*, 2018.

[28] A. Can, J. Picaut, J. Ardouin, P. Crepeaux, E. Bocher, D. Ecotiere, M. Lagrange, C. Lavandier and V. Mallet, “CENSE Project: general overview,” in *Euronoise 2021: European Congress on Noise Control Engineering*, 2021.

[29] J. O. Nordby, “Environmental sound classification on microcontrollers using Convolutional Neural Networks,” 2019.

[30] Grand View Research, “Microcontroller Market Size, Share & Trends Analysis Report By Product (8-bit, 16-bit, 32-bit), By Application (Consumer Electronics & Telecom, Industrial, Automotive), By Region, And Segment Forecasts, 2022 - 2030,” Grand View Research, February 2022. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/microcontroller-market. [Accessed 17 April 2022].

[31] C. Svec, “ESE101: MICROCONTROLLER PERIPHERALS, GPIOS, AND BLINKING LIGHTS: PART 1,” 17 May 2016. [Online]. Available: https://embedded.fm/blog/2016/5/16/ese101-peripherals-part-1. [Accessed 8 April 2022].

[32] W. Kester, “What the Nyquist criterion means to your sampled data system design,” *Analog Devices,* p. 1–12, 2009.

[33] Silicon Labs, “EFR32xG12 Wireless Gecko,” Silicon Labs, 2022. [Online]. Available: https://www.silabs.com/documents/public/reference-manuals/efr32xg12-rm.pdf. [Accessed 5 April 2022].

[34] Wikipedia contributors, *IEEE 802.15.4 — Wikipedia, The Free Encyclopedia,* 2022.

[35] FreeRTOS, “FreeRTOS Real-time operating system for microcontrollers,” Real Time Engineers Ltd, 2022. [Online]. Available: https://www.freertos.org/. [Accessed 5 April 2022].

[36] Keil, “Thread Flags,” Keil, 2022. [Online]. Available: https://www.keil.com/pack/doc/cmsis/RTOS2/html/group__CMSIS__RTOS__ThreadFlagsMgmt.html. [Accessed 6 April 2022].

[37] A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM J. Res. Dev.,* vol. 3, pp. 210-229, 1959.

[38] S. Sharma, “Epoch vs Batch Size vs Iterations,” Toward Data Science, 23 September 2017. [Online]. Available: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9. [Accessed 7 April 2022].

[39] A. Agrawal, “Loss Functions and Optimization Algorithms. Demystified.,” Medium, 29 September 2017. [Online]. Available: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c. [Accessed 7 April 2022].

[40] J. Brownlee, “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning,” Machine Learning Mastery, 3 July 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed 8 April 2022].

[41] P. Nandi, “CNNs for Audio Classification,” Toward Data Science, 24 March 2021. [Online]. Available: https://towardsdatascience.com/cnns-for-audio-classification-6244954665ab. [Accessed 7 April 2022].

[42] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss and K. Wilson, “CNN architectures for large-scale audio classification,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[43] K. Palanisamy, D. Singhania and A. Yao, “Rethinking CNN models for audio classification,” *arXiv preprint arXiv:2007.11154,* 2020.

[44] IBM Cloud Education, “Convolutional Neural Networks,” IBM, 20 October 2020. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks. [Accessed 9 April 2022].

[45] TensorFlow, “tf.keras.layers.Dropout,” TensorFlow, 3 February 2022. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout. [Accessed 10 April 2022].

[46] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way,” Toward Data Science, 15 December 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed 10 April 2022].

[47] TensorFlow, “TensorFlow Lite,” TensorFlow, 31 March 2022. [Online]. Available: https://www.tensorflow.org/lite/guide. [Accessed 10 April 2022].

[48] Google, “FlatBuffers,” Google, 2022. [Online]. Available: https://google.github.io/flatbuffers/. [Accessed 11 April 2022].

[49] TensorFlow, “Post-training quantization,” TensorFlow, 2022. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization. [Accessed 11 April 2022].

[50] TensorFlow, “Build and convert models,” TensorFlow, 2022. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers/build_convert. [Accessed 12 April 2022].

[51] TensorFlow, “Get started with microcontrollers,” TensorFlow, 2022. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers/get_started_low_level. [Accessed 12 April 2022].

[52] E. Doering, “Musical Signal Processing with LabVIEW -- Introduction to Audio and Musical Signals,” OpenStax CNX, 6 March 2018. [Online]. Available: http://cnx.org/contents/19fa0272-7172-479a-8617-a7e348624350@1.6. [Accessed 15 April 2022].

[53] Wikipedia contributors, *Mel scale — Wikipedia, The Free Encyclopedia,* 2021.

[54] V. Velardo, “Mel-spectrograms Explained Easily,” 13 September 2020. [Online]. Available: https://github.com/musikalkemist/AudioSignalProcessingForML/blob/master/17%20-%20Mel%20Spectrogram%20Explained%20Easily/Mel%20Spectrograms%20Explained%20Easily.pdf. [Accessed 14 April 2022].

[55] K. Choi, G. Fazekas, M. Sandler and K. Cho, “A Comparison of Audio Signal Preprocessing Methods for Deep Neural Networks on Music Tagging,” in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018.

[56] D. Marshall, “The Discrete Cosine Transform (DCT),” 4 October 2001. [Online]. Available: https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html. [Accessed 13 April 2022].

[57] Silicon Labs, “UG309: Thunderboard Sense 2 User's,” Silicon Labs, 2022. [Online]. Available: https://www.silabs.com/documents/public/user-guides/ug309-sltb004a-user-guide.pdf. [Accessed 16 April 2022].

[58] Silicon Labs, “TensorFlow Micro Speech,” Silicon Labs, October 2020. [Online]. Available:

https://github.com/SiliconLabs/platform_applications/tree/master/platform_tensorflow_micro_speech. [Accessed 16 April 2022].

[59] J. Ehala, “ADC-Stream,” Thinnect, 2021. [Online]. Available: https://github.com/proactivity-lab-study/adc-ldma-read-silabs. [Accessed 5 April 2022].

[60] V. Valerdo, “How do we extract audio features?,” 16 July 2020. [Online]. Available: https://github.com/musikalkemist/AudioSignalProcessingForML/tree/master/6-%20How%20to%20extract%20audio%20features. [Accessed 17 April 2022].

[61] Numpy, “numpy.hanning,” NumPy Developers, 2022. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.hanning.html. [Accessed 16 April 2022].

[62] Y. Wang, P. Getreuer, T. Hughes, R. F. Lyon and R. A. Saurous, “Trainable frontend for robust and far-field keyword spotting,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[63] TensorFlow, “Tensor.QuantizationParams,” TensorFlow, 2022. [Online]. Available: https://www.tensorflow.org/lite/api_docs/java/org/tensorflow/lite/Tensor.QuantizationParams. [Accessed 20 April 2022].

[64] L. Roeder, “Netron,” 2022. [Online]. Available: https://netron.app/. [Accessed 4 March 2022].

[65] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” *arXiv preprint arXiv:1804.03209,* 2018.

[66] D. A. Potvin, K. M. Parris and R. A. Mulder, "Geographically pervasive effects of urban noise on frequency and syllable rate of songs and calls in silvereyes (Zosterops lateralis).," *Proceedings. Biological sciences,* vol. 278, no. 1717, pp. 2464-9, August 2011.

[67] D. Purves and S. M. Williams, Neuroscience. 2nd edition, Sinauer Associates 2001, 2001.

[68] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal and M. Ritter, “Audio Set: An ontology and human-labeled dataset for audio events,” in *Proc. IEEE ICASSP 2017*, New, 2017.

[69] Google, “A large-scale dataset of manually annotated audio events,” Google, 2022. [Online]. Available: https://research.google.com/audioset/. [Accessed 22 April 2022].

[70] J. Kärner, “Custom audio dataset,” 10 May 2022. [Online]. Available: https://github.com/jelikaer/custom_audio_dataset. [Accessed 10 May 2022].