Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Duy Anh Pham

Implementation of a Speech-command-interface on Microcontroller with TinyML

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive and Aeronautical Engineering

Duy Anh Pham

Implementation of a Speech-command-interface on Microcontroller with TinyML

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Mechatronik
am Department Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Andreas Meisel
Zweitgutachter: Prof. Dr. Jörg Dahlkemper

Eingereicht am: June 21, 2021

Kurzreferat

Name desstudierenden

Duy Anh Pham

Thema der Bachelorthesis

Implementierung von Sprachkommandointerface auf der Basis eines Mikrocontrollers mit TinyML

Stichworte

Maschinelles Learnen, Deep Learning, Eingebettetes System, Spracherkennung, Eingebettetes KI

Kurzreferat

TinyML ist die neue Technologie, die die Implementierung und Bereitstellung von Maschinelles Learnen auf eingebetteten Systemen, insbesondere Mikrocontrollersystemen, ermöglicht. Das Kernstück einer TinyML-Anwendung ist die Inferenz-API, die auf dem TensorFlow Lite/Mikrokernel basiert. Diese Arbeit ist eine experimentelle Implementierung einer Sprachbefehlsschnittstelle auf einem Mikrocontroller. Das implementierte ML-Modell verwendet das MFCC als Sprachmerkmal, weil es häufig verwendet wird und sich in vielen Anwendungen als effektiv erwiesen hat. Anstelle eines Standard-CNN-Modells mit 2D-Faltungsfiltern wird der 1D-Faltungsoperator zum Extrahieren von Informationen aus Eingaben verwendet, da diese Methode dazu beiträgt, die Modellgröße noch weiter zu reduzieren, ohne viel Leistung zu verlieren. Am Ende wird ein winziges 1D-Conv-Modell geschaffen, das einen minimalen RAM-Verbrauch von $13,8kB$ hat. Das SCI ist als individuelles Sprachverarbeitungsmodul konzipiert, sodass es über eine serielle Kommunikation oder UART mit dem AT-Befehl als Anwendungsnachrichtenprotokoll mit dem externen Hostsystem verbunden ist.

Abstract

Name of the Student

Duy Anh Pham

Title of the paper

Implementation of a Speech-command-interface on Microcontroller with TinyML

Keywords

Machine learning, Deep Learning, Embedded System, Voice Recognition, Embedded ML

Abstract

TinyML is the new technology that enables the implementation and deployment of ML on embedded systems, particularly microcontroller systems. The core part of a TinyML application is the inference API built upon the TensorFlow Lite/micro-kernel. This document is an experimental implementation of a speech-command interface on a microcontroller. The implemented ML model uses the MFCC as speech features as it is commonly used and proven to be effective in many applications. Instead of a standard CNN model using 2D convolutional filters, the 1D convolution operator is applied for extracting information from inputs since this method helps to reduce the model size even more without losing much performance. In the end, we have achieved a tiny 1D-Conv model consuming minimal RAM usage of $13,8kB$. The SCI is designed as an individual speech processing module, interfacing with the external host system through a serial communication or UART with the AT command as the application message protocol.

Contents

List of Tables

List of Figures

Listings

1 Introduction

1.1 Motivation

Introduction to Edge Computing

We are living in the time of the Fourth Industrial Revolution, where everything is connected through the internet and becoming more intelligent. In the internet era, the amount of information transported at a time is incredibly huge. One of the reasons for this is how the internet infrastructure is designed. As the need for selective information is increasing time after time, we have to transform it and make it more meaningful before using it. This demand leads to the dominance of the cloud computing paradigm. At the time of this thesis, most engineers are tempted to build their system after this design concept. The information is operated on remote cloud services for data processing before being consumed later. With the rise of the internet of things, both humans and all kinds of devices will become a crucial part of the whole internet ecosystem. Many statistics reports show that the number of IoT devices is now surpassing non-connected ones. As a result, the need for even bigger data centers and faster communication technology will increase. That means there is a higher risk that these systems will break down as they are harder and costlier to maintain. One solution for this problem is the edge computing paradigm. This new design pattern helps bring the processing of data closer to the data source [Shi u. a., 2016]. The devices closer to the data source are considered edge devices, such as smartphones, wearables, or tiny sensor nodes. Their mission in the old-style IoT system is to gather the raw data, send it to the cloud server for processing and consume the processed information sent back to them. But, in an edge computing ecosystem, their role is changed. They act as data consumers and as data producers or data processors. That means they must produce more readable and meaningful information. The users may use and interact with this information later through an user interface at the application level. The table below shows some examples of raw data and application data for comparison. For instance, in a speech recognition system, raw data is digitalized electrical signals illustrating acoustic sound

waves. The user, in this case, is interested in the meaning of the sound indicating an event or a spoken word. To do this, we will need robust information processing algorithms and techniques like machine learning. Their performance is outstanding and comparable to or even surpasses humans' scores. But these methods are invented for computers with sufficient resources. They may not meet the requirement of real-life use cases when running on edge devices.

Table 1.1: Raw and application data

raw data (may not useful for user)	**application data (meaningful and readable)**
sensor data (acceleration, magnet field)	movement (running, walking, standing ...)
images	counting detected object class
converted electrical sound signal	voice or speech detection

1.1.1 Pros and Cons of edge computing.

Compared to the older paradigm, edge computing has some great benefits that need to be mentioned. First of all, the system will be more secure and reliable. We don't need to send any sensitive information anywhere, as these are produced locally. Therefore, it improves privacy, which is a critical prerequisite of any application involving the internet and connection. Less transported data also means a reduction of data loss and bandwidth saving. Not waiting for responses from servers helps cut down latency. This makes it more usable for real-time applications. Finally, the scalability grows as the cost of server and communication is lowered, and the edge devices' deployment is much easier than the rest of the system.

Many efforts have been made to make use of those benefits, but we are still facing many difficulties implementing machine learning algorithms at the edge until recently. Although many hardware manufacturers are still developing specialized devices to run AI models without using a sufficiently furnished PC, the prices of this kind of equipment are still too high to use in large-scale applications. Some problems could indeed be solved with a simple design, but deploying these models on a too capable computing system also means wasting too many resources. As a result, they will soon be dominated by smaller devices like embedded systems on the microcontroller. This leads to the beginning of a new research area referred to as "Embedded Maschine Learning" or "TinyML."

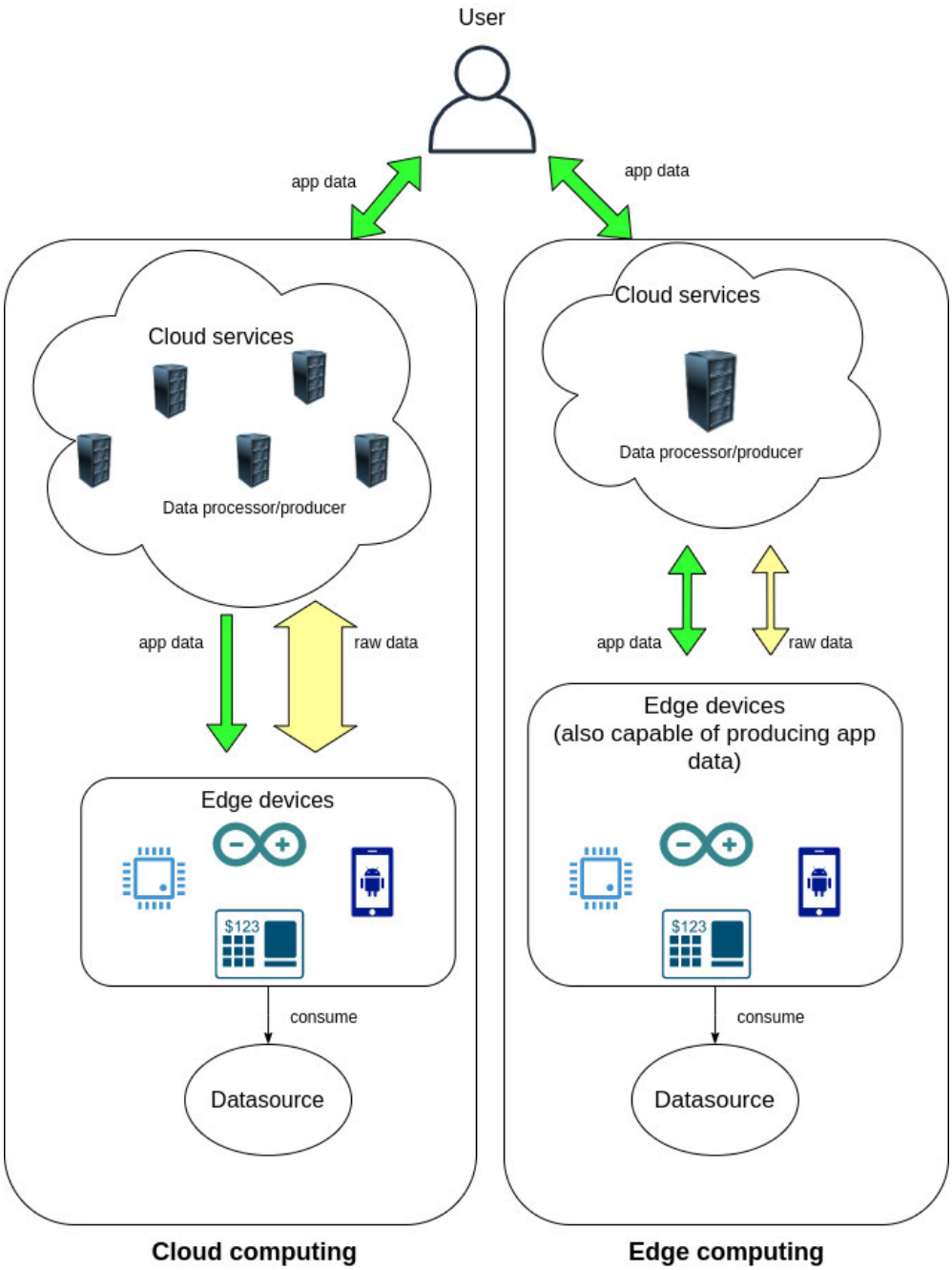

Figure 1.1: Cloud- and Edge-computing

Speech recognition on Microcontroller systems

Speech or voice recognition is one of the most famous problems possibly solved with machine learning, among many other use cases. Some well-known examples are digital assistants (e.g., Siri, Alexa) and speech-to-text systems. These models are already deployed on devices like smartphones, PCs, and wearables. With the birth of embedded machine learning, we can now run these models on tiny devices based on a microcontroller. A Microcontroller has minimal resources compared to other edge devices. Nevertheless, it perfectly fits an IoT system and tends to play a more critical role in the future of edge computing and machine learning as a targeted platform for deployment. Microcontrollers have some noticeable advantages making them more and more popular nowadays. They have low prices and power consumption, making them more portable and easier to scale. On average, it cost just under a dollar for a microcontroller with up to $100kHz$ clock frequency. In operation, they could consume a few hundreds of mW at peak. When on sleep mode, they could still run at tens of µW. Therefore, they are capable of running on batteries for months or years. Apparently, an embedded system consists of many more components than just only a microcontroller. Here are some overviews of the embedded system's typical usage [Warden und Situnayake,2019]:

Table 1.2: Power usage over view of some embedded system component

component	**approximately power usage**
display	400 mW
active cell radio	800 mW
Bluetooth	100 mW
Bluetooth Low Energy	40 mW
MEMS microphone sensor	1 mW
320x320 image sensor (Himax HM01B0)	1 mW at 30 FPS
accelerometer	1 mW

One of the limitations is that microcontrollers have some tough resource constraints. They often have just a few hundred kilobytes of RAM and a little more for the Flash memory. Moreover, the development of embedded systems is sometimes difficult to access because of the lack of standards. A big step forward in modern days is introducing many user-friendly integrated development frameworks for embedded systems like Arduino and MbedOS, along with more standardized hardwares. Consequently, the newcomer to the embedded world like mathematicians and AI expertises can now

keep up with the other engineers because the development is more straightforward and accessible. The new ICT and ML progresses expand the embedded system application further. Microcontrollers can now be used to control, gather data and do more complex tasks like motion detection, environment monitoring, abnormalities defection detection, or image and voice classification. They can provide end-to-end solutions to these problems using embedded ML methods. Among those problems which we can solve with ML, speech recognition is one of the most popular. Many AI models are designed and already implemented. These types of applications help to improve human-machine interaction in many cases. Here are some examples of how a speech recognition system may work:

- intelligent control in a smart home or automobile (e.g., voice-controlled lock/switch or light system)
- controlling multiple units with voice (e.g., a swarm of robots or drones)
- adding a voiced shortcut for some commands in a system with a complex monitoring user interface

In conclusion, applying voice recognition on microcontrollers is an excellent demonstration of extending ML's application and enhancing embedded system performance, making it more interactive and more intelligent.

1.2 Objectives

The thesis aims to use the new approach of "TinyML" to implement a speech-command-interface, which improves interactivity and cooperation between humans and machines. The targeted developing platforms for this task are microcontroller systems. We will first go through some fundamentals theory on signal processing methods to analyze and reconstruct human voice into data used in a deep learning model. We also examine how a trained model can be transformed and deployed on a microcontroller. An experimental implementation of a speech-command-interface is built to evaluate this new technique's possibility and its concrete performance in an interactive real-time application. A speech-command-interface is not a complete voice recognition system. So it shouldn't perceive all the words defined inside a dictionary. Instead, only a small set of words are chosen for a specific application. Besides, any AI model, in general, is a probabilistic system. That means it couldn't work most of the time but not as consistently as the physical

interfaces, which is, in contrast, deterministic. A physical emergency stop switch is preferable to a voice command for a more secure mechanism. As a matter of fact, a speech-command interface should not replace the whole human-machine interface for most applications but be integrated as an enhanced feature. Therefore, this work aims not to improve the model or the algorithm but to apply the state-of-the-art method of speech recognition to create an interface that is usable and easy to integrate as a part of an HMI.

1.3 Structure

Chapter 2 describes the methodology and theory on the topics of speech recognition, machine learning, and AI development for embedded systems. Before getting started with signal processing methods, we learn about some phonetics fundamentals to understand the structure of speech created by humans. We continue examining two of the most well-known speech features with this knowledge: the spectrogram and the MFCC. The central algorithm used by both methods is the FFT and the Mel filterbank in audio processing. Later we need a complement AI algorithm to build a full speech recognition model. For this, we choose the CNN model with the Google Command Set as our base approach. Finally, we discuss the newly emerging technology of TinyML or embedded ML. In this section, we made a preview of the use of TensorFlow Lite as well as the optimization methods it applies.

Chapter 3 is our proposed design for the speech-command interface. Unfortunately, only a few available development hardware kits can run ML applications at the time of this work. For this reason, we keep researching the state-of-the-art to examine the constraints we may encounter when developing an embedded ML application. Furthermore, we also analyze the specifications of the available development kits on which we want to deploy our experimental speech-command interface's design. Finally, at the end of this chapter is a brief overview of the requirements applying to the SCI.

Chapter 4 describes the implementation of SCI. First, we explain the use of software frameworks and tools in developing modern embedded ML applications to further optimize the resource usage on microcontrollers. Next, we document the steps we made to train and deploy ML models on the targeted platform. In this section, we explain all the parameters we used in detail base on experiments and researches. Finally, in the end, we also illustrate the basic functionality of the SCI briefly.

Chapter 5 is the evaluation of our test application with SCI. For this, we estimate our AI model and make a comparison with the state-of-the-art models. We also include a demonstration of SCI in action at the end, along with its concrete performance in real-time application.

2 Theory

2.1 Speech recognition with machine learning

2.1.1 Phonetics fundamentals

Before getting started with signal processing methods, it is necessary to understand the human's speech structure. Sound is constructed by the vibration of an acoustic wave transmitted through a medium such as gas, liquid, or solid. Humans create sounds with a particular vocal organ, the vocal tract, which acts as a resonator. It consists of two parts known as the nasal tract and the oral tract, which indicate where the sounds are created.

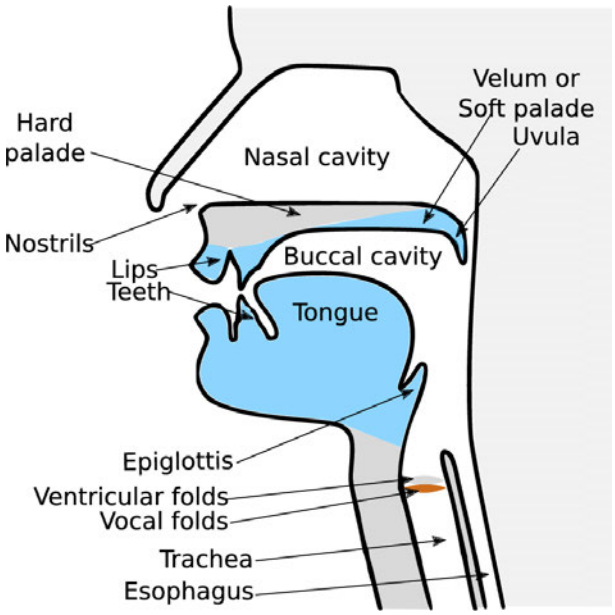

Figure 2.1: The vocal tract [Andrade-Miranda, 2017]

As the air goes through these tracts, it vibrates and creates the spoken speech. Speech sounds or phones are elemental sounds, which can be voiced or voiceless. For

example, in the English language, the voiced sounds or phones are [b], [g], [d], as well as all the English vowels. Unvoiced sounds include [p], [t], [k], and others. People also classify these sounds into two main classes: vowels and consonants. They are possibly represented with symbols adapted from the Roman alphabet used in dictionaries as pronunciation instructions. The International Phonetic Alphabet (IPA), first developed in 1988, is the standard representation for transcribing the world's language. The APRAbet, a subset of IPA, is widely used for the American-English languages. [Jurafsky und Martin, 2000, chap. 25]

ARPAbet Symbol	IPA Symbol	Word	ARPAbet Transcription	ARPAbet Symbol	IPA Symbol	Word	ARPAbet Transcription
[p]	[p]	parsley	[p aa r s l iy]	[iy]	[i]	lily	[l ih l iy]
[t]	[t]	tea	[t iy]	[ih]	[ɪ]	lily	[l ih l iy]
[k]	[k]	cook	[k uh k]	[ey]	[eɪ]	daisy	[d ey z iy]
[b]	[b]	bay	[b ey]	[eh]	[ɛ]	pen	[p eh n]
[d]	[d]	dill	[d ih l]	[ae]	[æ]	aster	[ae s t axr]
[g]	[g]	garlic	[g aa r l ix k]	[aa]	[ɑ]	poppy	[p aa p iy]
[m]	[m]	mint	[m ih n t]	[ao]	[ɔ]	orchid	[ao r k ix d]
[n]	[n]	nutmeg	[n ah t m eh g]	[uh]	[ʊ]	wood	[w uh d]
[ng]	[ŋ]	baking	[b ey k ix ng]	[ow]	[oʊ]	lotus	[l ow dx ax s]
[f]	[f]	flour	[f l aw axr]	[uw]	[u]	tulip	[t uw l ix p]
[v]	[v]	clove	[k l ow v]	[ah]	[ʌ]	butter	[b ah dx axr]
[th]	[θ]	thick	[th ih k]	[er]	[ɝ]	bird	[b er d]
[dh]	[ð]	those	[dh ow z]	[ay]	[aɪ]	iris	[ay r ix s]
[s]	[s]	soup	[s uw p]	[aw]	[aʊ]	flower	[f l aw axr]
[z]	[z]	eggs	[eh g z]	[oy]	[oɪ]	soil	[s oy l]
[sh]	[ʃ]	squash	[s k w aa sh]				
[zh]	[ʒ]	ambrosia	[ae m b r ow zh ax]				
[ch]	[tʃ]	cherry	[ch eh r iy]				
[jh]	[dʒ]	jar	[jh aa r]				
[l]	[l]	licorice	[l ih k axr ix sh]				
[w]	[w]	kiwi	[k iy w iy]				
[r]	[r]	rice	[r ay s]				
[y]	[j]	yellow	[y eh l ow]				
[h]	[h]	honey	[h ah n iy]				

Figure 2.2: The APRAbet

Unlike wild animals, humans can produce a large number of sounds forming a spoken language or, in general, human speech. A word is a smallest meaningful unit of a speech. Therefore, one of the essential missions of a speech processing system is

recognizing spoken words. A word can be a single syllable or a combination of multiple syllables. On the other hand, a syllable consists of consonants and vowels, the smaller parts of the speech. So basically, a word is just a list of phones spoken in order of time — in figure 2.3 showing how we can separate sounds when analyzing the sound waves. Each word is transcribed using the APRAbet.

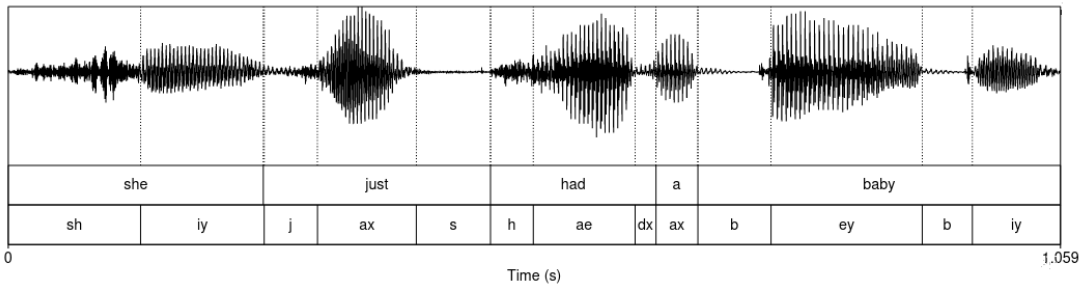

Figure 2.3: A waveform of the sentence “She just had a baby” with APRAbet transcription [Jurafsky und Martin, 2000]

2.1.2 Audio signal processing fundamental - extract speech features

2.1.2.1 Audio signal

In signal processing, we often record sounds and illustrate them as waveforms in time series. Then, we translate air vibration into electrical signals using sensors like microphones. The most common way of audio signal acquisition and encoding is using PDM and PCM. PDM is a form of modulation converting analog signals into a high-frequency stream of 1-bit digital samples. The relative density of the pulses represents the input signal’s amplitude. Figure 2.4 illustrates a converted sine wave in PDM format. A large cluster of 1s corresponds to a high or positive amplitude value when a large cluster of 0s would correspond to a low or negative amplitude value, and alternating 1s and 0s would correspond to a zero amplitude value.

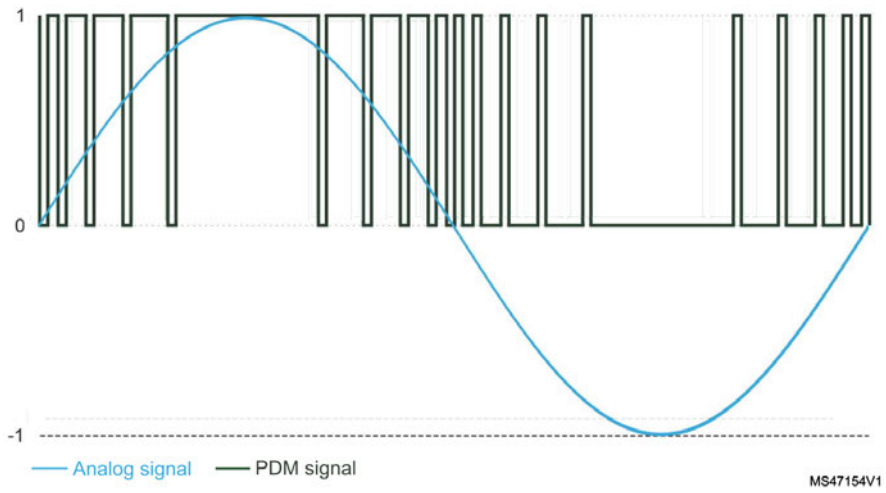

Figure 2.4: PDM signal [STMicroelectronics, 2019]

PDM signals are later encoded into PCM format representing digitalized signal as a stream of floating-point values, also known as data samples. A PCM stream has two basic properties: the sampling rate f_s and the bit depth. These properties help to reconstruct the original analog signal. In conclusion, a chain of a PDM and PCM acts as an ADC used for converting and streaming audio. PCM values are also a part of well-known audio file formats, for example, the WAVE file format, used in many computer applications.

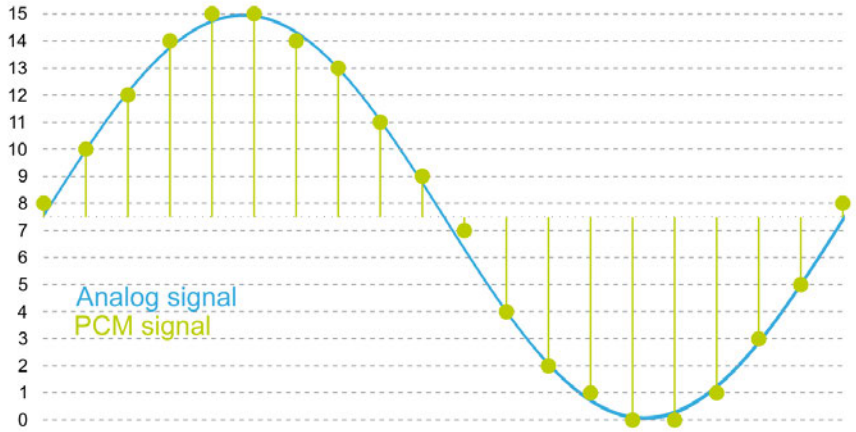

Figure 2.5: PCM signal [STMicroelectronics, 2019]

2.1.2.2 FFT and spectrogram

Figure 2.3 is an example signal of a spoken sentence. We can easily recognize the vowels and consonants pronounced. However, this signal is not always the same for every person because different people have different accents when saying the same sentence. There are two characteristics, which characterize human accents: pitch and loudness, and the speech recognition process should ignore them. The pitch indicates how humans perceive the sound vibration frequency if it is spoken in a higher or lower tone. On the other hand, the loudness is equivalent to the sound's energy. The human ears are more sensitive to lower pitch sounds, so we often need a pre-emphasis filter at the foremost to amplify higher pitched sound.

We later use the Fourier Transformation to analyze the sound wave's power spectrum to extract more information in the frequency domain. One way to apply Fourier Transformation on audio signals is the FFT, which is widely used for digital spectral analysis. In Speech processing, it is the most crucial algorithm for extracting information. FFT is a recursive, iterative and less complex process with a lower computation time than the straightforward DFT method, respectively $O(n \log n)$ and $O(n^2)$. The formel and the pseudo-code algorithm of FFT are as follows [Cormen u. a., 2009]:

- FFT formel

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{2.1}$$

- with

$$\omega_n^{kj} = e^{2\Pi i/n} \tag{2.2}$$

- FFT matrix formel

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} \tag{2.3}$$

- pseudo-code RECURSIVE-FFT(a)

```
1  n = a.length                          // n is a power of 2
2  if n == 1
3    return a
4  ω_n = e^(2Πi/n)
5  ω = 1
6  a^[0] = (a_0, a_2, ..., a_(n-2))
7  a^[1] = (a_1, a_3, ..., a_(n-1))
8  y^[0] = RECURSIVE-FFT(a^[0]))
9  y^[1] = RECURSIVE-FFT(a^[1]))
10 for k = 0 to n/2 - 1
11   y_k = y_k^[0] + ω y_k^[1]
12   y_(k+(n/2)) = y_k^[0] + ω y_k^[1]
13   ω = ω ω_n
14 return y                  // y is assumed to be a column vector
```

Another advantage of FFT is that the inverse FFT can be done with almost the same algorithm. The inverse matrix, for instance, is extracted by replacing every ω_n^{kj} with ω_n^{-kj}/n . In both cases, n or n_{FFT} is called FFT length.

- inverse FFT formel

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} y_j \omega_n^{-kj} \tag{2.4}$$

- with

$$\omega_n^{-kj} = e^{-2\Pi i/n} \tag{2.5}$$

- inverse FFT matrix formel

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \omega_n^{-2(n-1)} \\ 1 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \omega_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{2-(n-1)} & \omega_n^{-3(n-1)} & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{2.6}$$

- pseudo-code RECURSIVE-iFFT(y)

```
1  n = y.length                          // n is a power of 2
2  if n == 1
3    return y
4  ω_n = (1/n) e^{-2Πi/n}
5  ω = 1
6  y^[0] = (y_0, y_2, ..., y_{n-2})
7  y^[1] = (y_1, y_3, ..., y_{n-1})
8  a^[0] = RECURSIVE - iFFT(y^[0]))
9  a^[1] = RECURSIVE - iFFT(y^[1]))
10 for k = 0 to n/2 - 1
11   a_k = a_k^[0] + ω a_k^[1]
12   a_{k+(n/2)} = a_k^[0] + ω a_k^[1]
13   ω = ω ω_n
14 return a                  // a is assumed to be a column vector
```

The FFT is very popular for its great computation benefits and is implemented and optimized on most hardware platforms and programming languages as a DSP module [Liu u. a., 2019]. Firstly, we divide the sound waves into short-time frames or windows, often from $20ms$ to $40ms$ long. Then we calculate each frame's power spectrum using FFT. Finally, we save all results in a particular diagram called a spectrogram, which contains all power spectrums lying next to each other in chronicle order. The frequency range and the signal length are constant. Therefore, a spectrogram can be represented as an image. The magnitude of a frequency range f_i at the n_j window is, therefore, represented as a pixel p_{ij}. We can determine the dominating frequencies with a higher energy level based on the highlighted lines or figures. These are called formants [Jurafsky und Martin, 2000]. Following is the spectrogram of the word "forward" and "backward." The formants are highlighted in red color, corresponding to spectral peaks. The lowest formant F0 is related to the pitch of the sound. If the spoken pitch is raised, these formants will grow upwards. We can see some similarities between the twos spoken terms from the image because the second part of the word "backward" is pronounced the same as the word "forward," particularly the last voiceless phone [d]. The formants in this section are more distinctive than the rest. We can extract these

repeated patterns from the resulted figure by applying image processing methods. For example, TensorFlow's standard audio recognition method uses these spectrograms as speech features (see [Sainath und Parada, 2015]). In practice, these spectrograms can be further transformed into MFCC, which is a more practical speech feature.

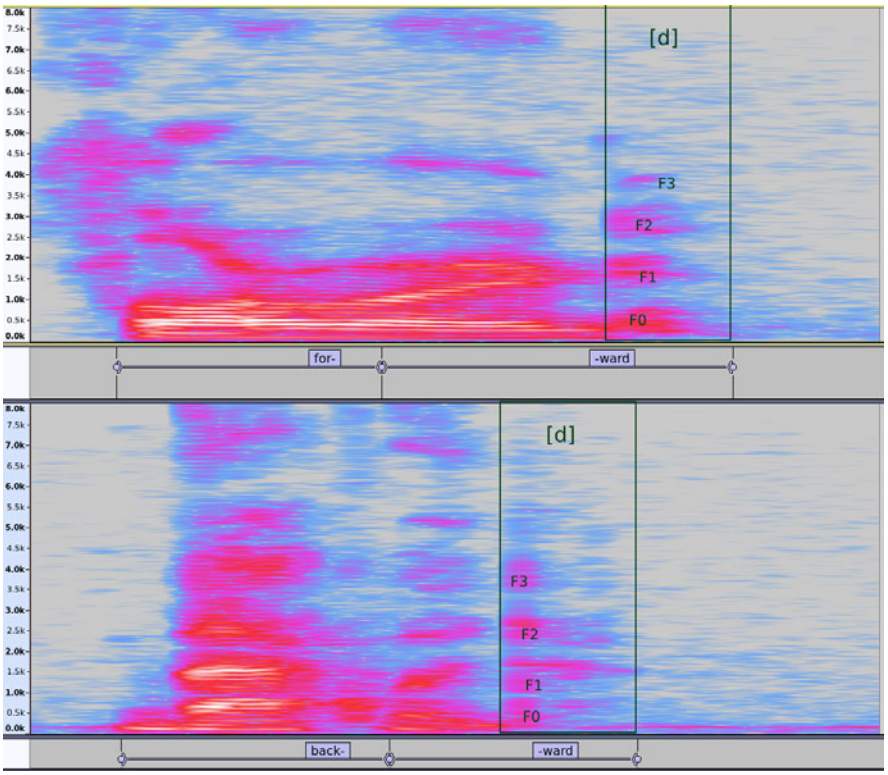

Figure 2.6: Spectrogram - "forward" and "backward"

2.1.2.3 MFCC

The MFCC is the method of extracting voice features based on how humans hear things. The main idea is to use a Mel filterbank to map the measured frequency into what humans perceive with hearing organ abilities. Our ears have about 15000 hairs inside the basilar membrane, and these hairs act as a band-pass filter receiving a particular narrow frequency range. The human's hearing range is between $300Hz$ and $3000Hz$. A Mel filterbank consists of triangular band-pass filters, typically between 20 - 40 filters, simulating the basilar membrane functionality. The filter bands grow wider as the frequency range rises, so the lower frequencies become more discriminative than

the higher frequencies, which is more transparent. This is comparable to how we, the human, hearing things as we are more sensitive to sound in lower tone. The result is another power spectrum called the Mel spectrum. The formants on the Mel scale are contrast and easier to identify. This process is identical to cropping and zooming on the region of interest when processing images.

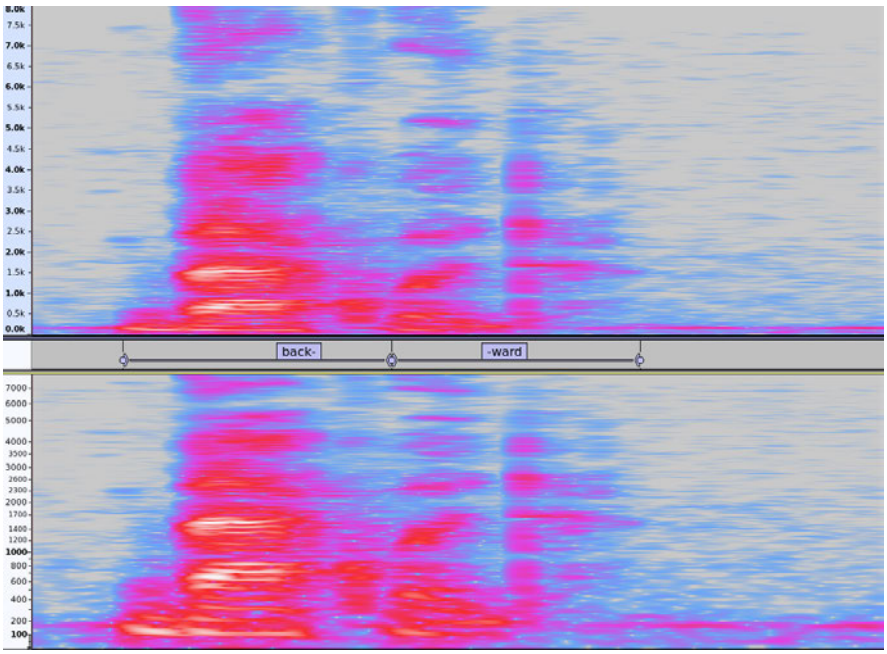

Figure 2.7: Spectrogram - below is in the Mel power spectrum

We construct the Mel filterbank as follows. First, we need equations to convert from frequency to Mel scale and vice versa.

$$m = Mel(f) = 1125 \ln(1 + \frac{f}{700})[Mel] \tag{2.7}$$

$$f = f(m) = 700(e^{m/1125} - 1)[Hz] \tag{2.8}$$

We initially evaluate the maximal frequency in the Mel scale using equation 2.7. Then we have to calculate a sequence of linear spaced values in the Mel scale based on how many filters we want to have. So, if we have n filters, we have to calculate $n + 2$ values, as each triangular filter is defined with a set of three values. Next, we convert this

sequence to regular frequencies using equation 2.8. The result is another sequence with logarithmically spaced values.

In the following example, we calculate a Mel filterbank of 40 for a range up to $4000Hz$. Firstly, we determine the maximal Mel frequency equivalent to the band limit of $4000Hz$, which is $2142.26Mel$. The full linear spaced Mel frequencies are as follows:

```
[    0.            52.25041791   104.50083582   156.75125372   209.00167163
  261.25208954   313.50250745   365.75292536   418.00334327   470.25376117
  522.50417908   574.75459699   627.0050149    679.25543281   731.50585071
  783.75626862   836.00668653   888.25710444   940.50752235   992.75794026
 1045.00835816  1097.25877607  1149.50919398  1201.75961189  1254.0100298
 1306.26044771  1358.51086561  1410.76128352  1463.01170143  1515.26211934
 1567.51253725  1619.76295515  1672.01337306  1724.26379097  1776.51420888
 1828.76462679  1881.0150447   1933.2654626   1985.51588051  2037.76629842
 2090.01671633  2142.26713424]
```

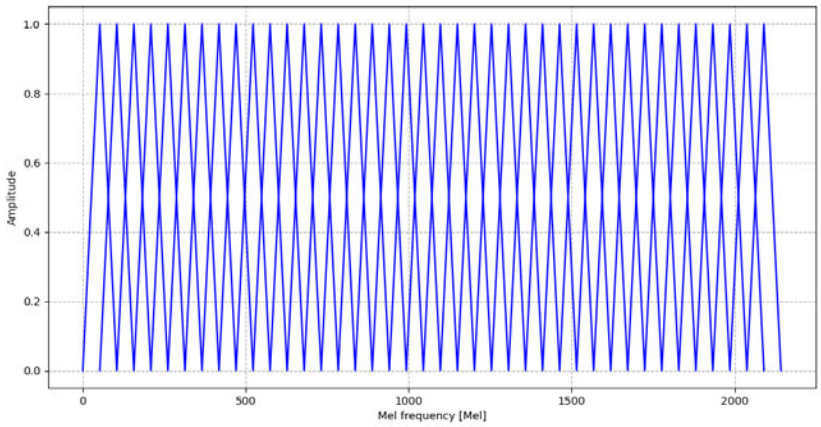

Figure 2.8: Mel linear spaced filterbanks

After converting, the new converted frequencies in Mel scale are as follows:

```
[    0.            33.27818895   68.13843198   104.65594038   142.90950098
  182.98164618   224.95883198   268.93162452   314.99489549   363.2480268
  413.79512499   466.74524587   522.2126298    580.31694812   641.18356143
  704.94378996   771.73519696   841.70188548   914.99480927   991.77209843
 1072.19940064  1156.45023851  1244.70638395  1337.15825037  1434.00530347
 1535.45649162  1641.73069664  1753.05720605  1869.67620777  1991.83930831
 2119.81007563  2253.86460776  2394.29212854  2541.39561156  2695.49243387
 2856.91506071  3026.01176279  3203.14736774  3388.70404717  3583.08214122
 3786.70102232  4000.          ]
```

When plotting on the frequency domain, the result is as in the following figure.

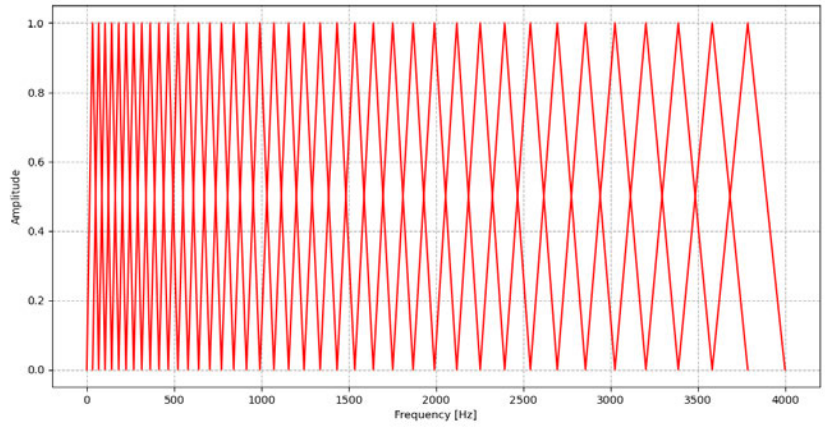

Figure 2.9: Converted Mel filterbank with 40 filters and frequency range upto $4000Hz$ [Fayek, 2016]

Calculating and ploting mel_filterbanks in Python

```
import numpy as np
import matplotlib.pyplot as plt

def Mel(f):
    return 1125*np.log(1+f/700)

def F(m):
    return 700*(np.exp(m/1125)-1)

def plot_filterbank(frequency_values, xlabel='Frequency [Hz]', ylabel='Amplitude', line_style='r'):
    amplitude = np.zeros(len(frequency_values[1:]))
    for i, a in enumerate(amplitude):
        amplitude[i] = i%2
    plt.plot(frequency_values[1:],amplitude, line_style)

    amplitude = np.zeros(len(frequency_values[:-1]))
    for i, a in enumerate(amplitude):
        amplitude[i] = i%2

    plt.plot(frequency_values[:-1],amplitude, line_style)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True, linestyle='--')
    plt.show()

n = 40
f_max = 4000
m_max = Mel(f_max)
n_frequency_val = n+2

m = np.linspace(0, m_max, n_frequency_val)

f = F(m)

plot_filterbank(f)
plot_filterbank(m, xlabel="Mel frequency [Mel]", line_style='b')
```

The next step is to calculate the cepstrum. Cepstrum is the reverse of the first four letters in the word "spectrum." We achieve this by applying inverse DFT to the logarithm of the estimated signal spectrum, also known as the power spectrum. We can use either inverse FFT or, more often, DCT as an alternative method. Sometimes DCT is more desirable as DCT outputs can contain more information on amounts of energy which may help to increase the classification performance [Gupta u. a., 2013]. As a result, we will have a compressed list of coefficients as the final representation of the MFCC. Typically, we select the coefficients from 2 up to 13 for the end outcome. The first one is related to the formant F0 or speaking pitch and shall be ignored. The other coefficients correspond to the higher frequency range outside the scope of regular speech and are also considered redundant.

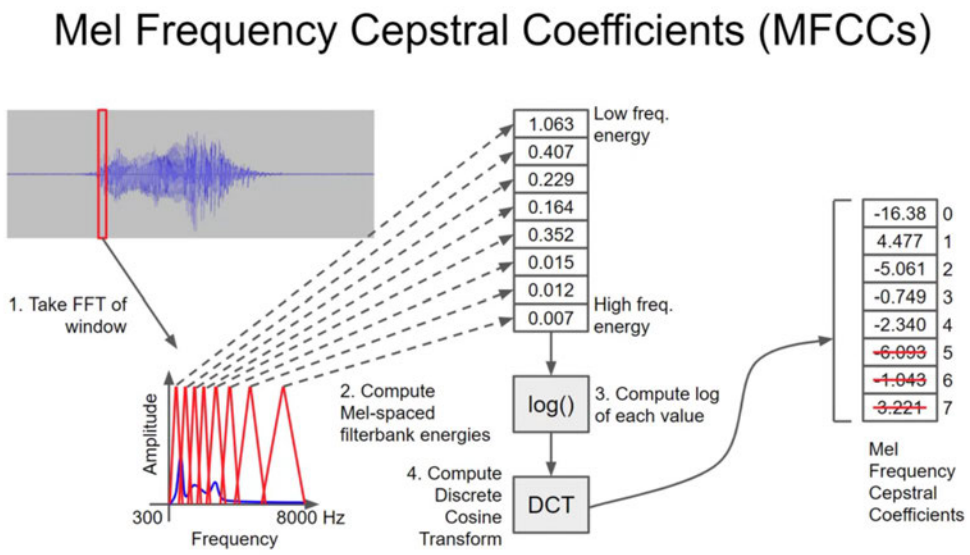

Figure 2.10: MFCC calculation steps [Shawn Hymel, 2020]

In conclusion, most signal processing chains for speech feature extraction are similar as in the following figure. However, this model could be slightly different for a variety of applications. With the MFCC, we are able to generalize the large structure of sound into highly uncorrelated and representable coefficient numbers. These features are suitable for applications processing speech and music models, which are sounds with

fixed patterns. The resulted features are saved in a 3D tensor or picture. The next step is to feed these features into our deep learning model for classification.

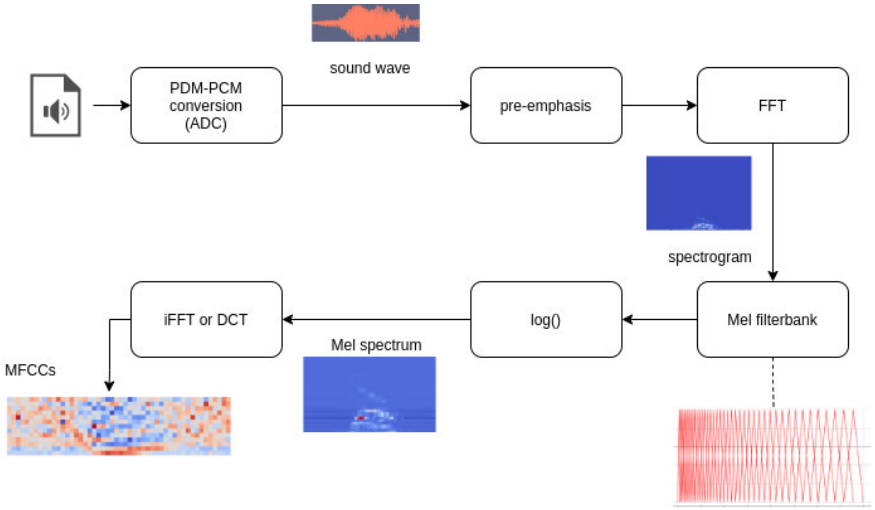

Figure 2.11: speech signal processing pipe

2.1.3 Speech recognition ML model

2.1.3.1 Deep learning with CNN [Goodfellow u. a., 2015]

The modern method we will use to determine and classify speech in this work is deep learning with neural networks. As mentioned in previous sections, the extracted features are represented as pictures, so the learning process is as similar as in image processing in computer vision. The most popular way to classify images is using convolutional neural networks. For example, a CNN-based model consists of layers classified as input/output layers, convolutional layers, subsample layers, and fully connected layers. A convolutional layer consists of 2D filters with the same kernel size, acting as the pattern finder or feature learner. The number of filters is called layer depth. Together with the striding step sizes, it defines the number of total overlapped windows. A subsampling layer, often a pooling layer, is a particular layer that helps to downsample the data to prevent overfitting and reduce model complexity. Finally, a fully connected layer contains only weights and biases and is often located at the end before activating the final classification at the output. CNN models are considered as a black box, which learns automatically from complex datasets. The more complex the model is, the more effective it learns. An ML engineer or researcher has to configure the attributes to find

out a model complicated enough to determine all the desired features and, at the same time, optimize enough to deploy on the target platform. The CNN model's complexity is defined by the layer number, the number of filters in each layer, and the attributes of each layer.
In figure 2.12 is the architecture of the classic LeNET5 model, one of the classic CNN. This model contains all of the mentioned basic components of the CNN.

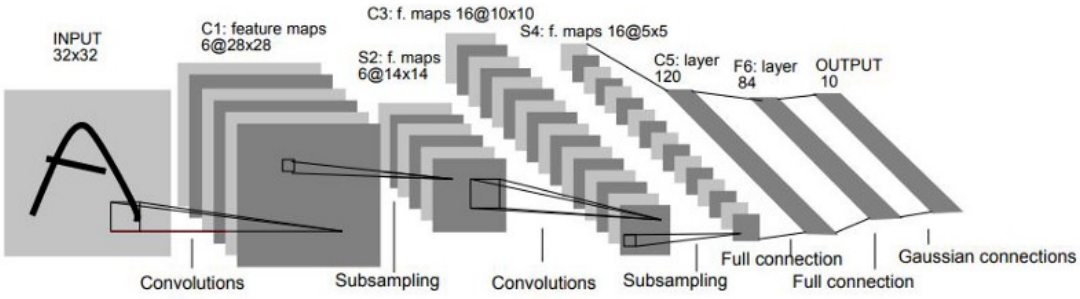

Figure 2.12: LeNET5 - handwriting classification [LeCun u. a., 1998]

2.1.3.2 Dataset

The dataset used in this thesis is the Google Speech Command Set Version 2. This dataset contains audio records of 35 common English words used as commands for robotics or automation applications. The recorded words are listed in Table 2.1. Each audio file is a 1-second-long WAVE-formated file with the sample data encoded in 16-bit mono-channel PCM values. The resolution is 16-bit, which means a floating-point of -1.0 is equivalent to -32768, and 1.0 equals +32767 in linear scale with type signed integer. The sample rate is $16kHz$. This is equivalent to a maximum frequency range of $8kHz$, according to Nyquist-Shannon-Sampling-Theorem. These properties must coincide when implementing on the targeted device to retain data consistency when recording raw input. This dataset is widely tested in various research and is proven to be efficient.

Word	Number of Utterances
Backward	1,664
Bed	2,014
Bird	2,064
Cat	2,031
Dog	2,128
Down	3,917
Eight	3,787
Five	4,052
Follow	1,579
Forward	1,557
Four	3,728
Go	3,880
Happy	2,054
House	2,113
Learn	1,575
Left	3,801
Marvin	2,100
Nine	3,934
No	3,941
Off	3,745
On	3,845
One	3,890
Right	3,778
Seven	3,998
Sheila	2,022
Six	3,860
Stop	3,872
Three	3,727
Tree	1,759
Two	3,880
Up	3,723
Visual	1,592
Wow	2,123
Yes	4,044
Zero	4,052

Table 2.1: How many recordings of each word arepresent in the dataset? [Warden, 2018]

2.2 Embedded Machine Learning

2.2.1 TinyML preview

Embedded ML, a subset of Edge AI, is the new technology that enables the implementation and deployment of ML on embedded systems, particularly microcontroller systems. TinyML is the name of the foundation supporting development progress on embedded ML, sometimes is also referred to as the technology itself. It is a fast-growing engineering field involving developing hardware, algorithms, and software capable of on-device data analytics at low power. The emerging of this technology associates deeply with the development of TensorFlow, a popular framework for developing ML applications. As a result, embedded systems are now a promising targeted platform for deploying ML.

2.2.2 TensorFlow lite

TensorFlow plays a crucial role in developing embedded ML, as mentioned in the book "TinyML." The core part of a TinyML application is the inference API built upon the TensorFlow Lite/micro-kernel. TensorFlow Lite kernel is used for deploying ML models on mobile and edge devices, namely on smartphones or on single-board computers like the Raspberry Pi or the Jetson Nano. Its subset is the micro-kernel, which is recently ported for the microcontrollers. In addition, the core functions and operators of the standard API are reimplemented upon the CMSIS-NN software library provided by ARM. Many optimizations have been done to shrink the algorithms, networks, and models' complexity and resource down to merely $100kB$. One of the applied techniques is quantization. The weights and biases by default are represented as 32-bit floating-point numbers. As the FPU on microcontrollers is very slow in inefficient compared to modern computers, it is a best practice to rescale these values in new types, particularly in 8-bit integer. This is equivalent to a 75% reduction in memory size. Moreover, the CPU also performs math operators on integers much faster. There is a minimal accuracy loss, but this performance exchange is still worthwhile. Without it, it is impossible to run ML on those MCUs. Figure 2.13 illustrates the actual prediction values between the quantized and unquantized models. The patterns are almost identical, showing that the quantized model can be used in real applications.

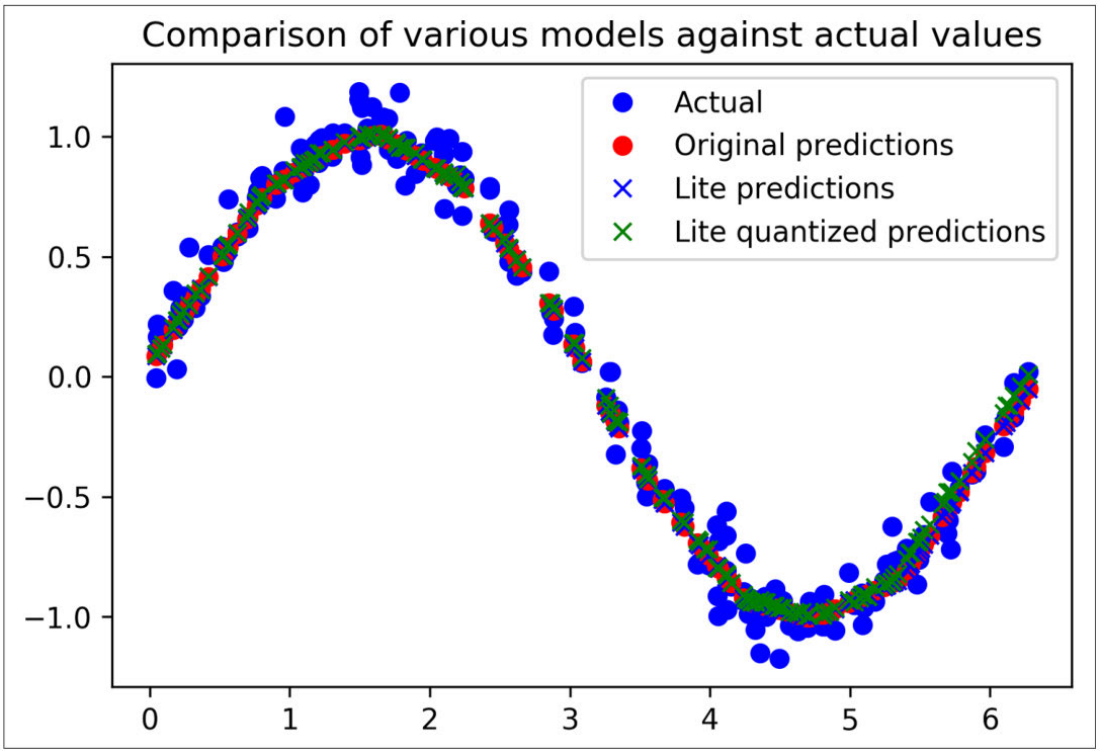

Figure 2.13: A sine model prediction: normal vs quantized [Warden und Situnayake, 2019]

2.2.2.1 tflite converter and interpreter

In this section, we will take a look at how TensorFlow Lite works on microcontrollers. First is the converter or TOCO. This program lets the trained model reconstruct into a particular space-efficient format, a FlatBuffer, for memory-constrained devices. The converted model is also optimized with quantization and saved in a file with an extension "tflite." This file is a C source code but is encoded as a raw byte array. To test this, we can use some software tools like the famous "xxd" on Unix/Linux PC to dump the file's content into a readable format like Unicode (see more in [Warden und Situnayake, 2019, chap. 4]). For this reason, we can compile this file and load it directly into memory as every C/C++ program does.

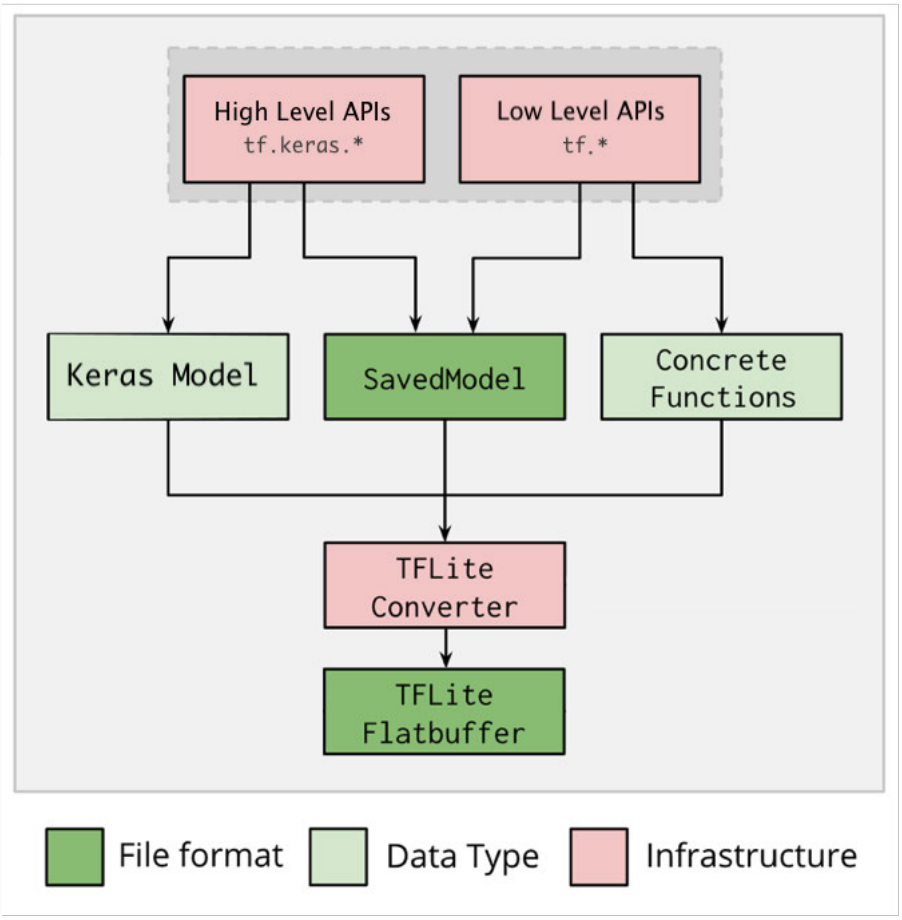

Figure 2.14: TensorFlow Lite Converter [official TensorFlow documentation, 2021]

However, we still need an interpreter to run inference. The reason is, inside this special model-C-file, there are some excessively long serializable arrays of bytes (up to tens of kBs), which are the model data arrays containing the tensors, weights, and operators that need to be parsed or mapped into C++ class or struct at runtime. This makes the model memory efficient. So, whenever we want to update our model, we need to replace this file with a newly converted one and recompile the code. The

inference then occurs after we create an interpreter and call a function through it. The steps to use TensorFlow Lite on microcontrollers is as follows. Only necessary steps are listed. Steps with helper functions are not included.

After training

- export the model to lite-model using TOCO (file with .tflite)
- convert this model into a C/Cpp file

Deploying on MCU

- mapping models by creating a Model pointer pointing to the model data array

```
1    // Map the model into a usable data structure.
2    //This doesn't involve any copying or parsing,
3    //it's a very lightweight operation.
4    const tflite::Model* model = ::tflite::GetModel(g_sine_model_data);
```

- Creating an AllOfResolver - a particular class - resolving mathematics operators for transforming inputs and outputs.

```
1    // This pulls in all the operation implementations we need
2    tflite::ops::micro::AllOpsResolver resolver;
```

- Define a tensor area - this is a memory area used by the NN operations, including the memory for loading the model and operators. We may need to do many trials to find a minimum or optimized memory usage.

```
1    // Create an area of memory to use for input, output, and intermediate arrays.
2    // Finding the minimum value for your model may require some trial and error.
3    const int tensor_arena_size = 2 * 1024;
4    uint8_t tensor_arena[tensor_arena_size]
```

- Create an interpreter instance

```
1    // Build an interpreter to run the model with
2   tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,tensor_arena_size, error_reporter);
```

- Now we can call the methods of the interpreter to invoke the inference process, including:
  - allocating Tensor memory area

```
1        // Allocate memory from the tensor_arena for the model's tensors
2        interpreter.AllocateTensors()
```

  - assigning input/output buffer

```
    // Obtain a pointer to the model's input/output tensor
    TfLiteTensor* input = interpreter.input(0);
    TfLiteTensor* output = interpreter.output(0);

    // Provide an input values
    // e.g: a 2D-matrix [[1 2 3] [4 5 6]]
    input->data.f[0] = 1.;
    input->data.f[1] = 2.;
    input->data.f[2] = 3.;
    input->data.f[3] = 4.;
    input->data.f[4] = 5.;
    input->data.f[5] = 6.;
```

– invoke - running inference

```
    // Run the model on this input and check that it succeeds
    TfLiteStatus invoke_status = interpreter.Invoke();
```

– getting output values after several inference invokes

```
    float value;

    // Run inference on several more values and confirm the expected outputs
    input->data.f[0] = 1.;
    interpreter.Invoke();
    value = output->data.f[0];

    input->data.f[0] = 3.;
    interpreter.Invoke();
    value = output->data.f[0];


    input->data.f[0] = 5.;
    interpreter.Invoke();
    value = output->data.f[0];
```

3 Requirement and Design

3.1 Design conception

3.1.1 Usecase preview and general design

Besides diving into ML training processes, we also design a system used by the human-machine-interaction. As mentioned in section 1.2, the objective of a speech-command interface is to enhance the overall HMI. The device or gadget equipped with an SCI must be able to recognize human voice commands and do some action reactively. Because any voice recognition system is probabilistic, we should construct the SCI as a separated embedded module. Therefore, it should run on its isolated hardware and software, also known as driver or firmware, in an as compact or minimal format as possible. The most successful ML applications always have a microprocessor specializing in executing ML algorithms, such as the TPU or tensor-processing unit developed by Google. This kind of microprocessor is referred to as NPU or neural processing unit and is an excellent example of how we can design and build such a system. This design concept is suitable for classification problems as the outcome is as simple as only the predicted class. Besides, the main application running on external devices is not interfered with by heavy ML processes. Accordingly, an SCI module must compose of at least three elements: a microphone for signal recording, a microcontroller for signal processing and inferencing, and finally, a communication interface and protocol for outputting results to external devices. The microphone and the microcontroller can be seen as a single speech-processing unit as the microphone drivers are usually provided inside the targeting MCU's software packages. This approach helps reducing development time, and the developer can focus on ML algorithms. A message-based communication protocol ensures that the main application is more secure and interactive as it can safely ignore or miss any prediction event. Physical or deterministic commands are always preferred to voice commands.

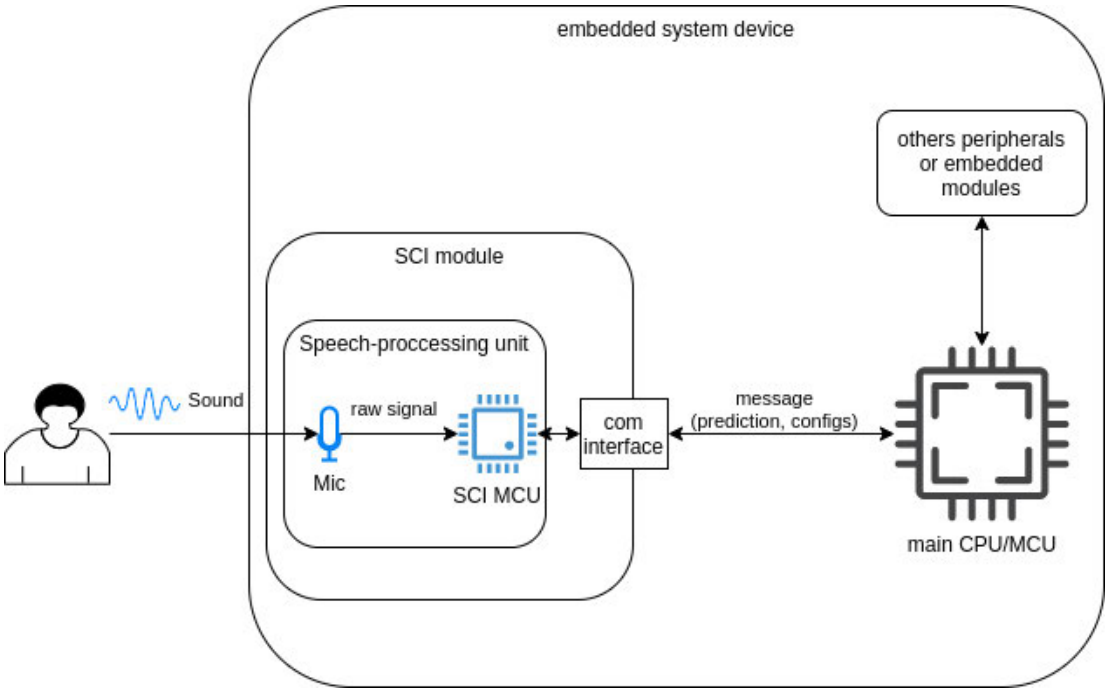

Figure 3.1: SCI sample design

3.1.2 State of the art - keyword spotting on microcontroller systems

This section will have a short review of some speech ML models aiming at resource-limited devices. At the time of this work, many researchers and engineers have made efforts to build speech recognition models targeting microcontroller systems. As a critical requirement, the primary criteria for evaluating these models are the calculation complexity measured in MOps (mega operations) and the consuming memory size in kB (kilobytes). Some of the best-performance NN-based models are listed in Table 3.1. These models are built with MFCC as extracted features, except the spectrogram-based DNN model. The number of to be predicted class is ten, those including the following classes: "Yes", "No", "Up", "Down", "Left", "Right", "On","Off", "Stop", and "Go". There are also two special classes like "__Unknown__" and "__background_noise__" (or also referred to as "Silence"). "__Unknown__" categorized words are randomly selected from the unselected labeled words.
The most light-weighted model would consume about 40kB of memory when operating inference only (ignoring memory consumption by feature extraction). Moreover,

most models except the DNN one would take at least $3MOps$, that is to say, for a minimal ARM® Cortex™-M4 CPU running at about 60 MHz, it would take at least $0,05s$ ($3MOps/60Mhz$) for each inferencing session.
Some popular development boards with computation specifications are listed in Table 3.2. Linking to the requirements in Table 3.1, we can conclude that a deep-learning-based speech recognition application can run on a microcontroller with a core architecture not weaker than an ARM® Cortex™-M4.

Table 3.1: Summary of best neural networks from the hyperparameter search. The memory required for storing the 8-bit weights and activations is shown in the table [Zhang u. a., 2018]

NN model	S(80KB, 6MOps)			M(200KB, 20MOps)			L(500KB, 80MOps)		
	Acc.	Mem.	Ops	Acc.	Mem.	Ops	Acc.	Mem.	Ops
DNN	84.6%	80.0KB	158.8K	86.4%	199.4KB	397.0K	86.7%	496.6KB	990.2K
CNN	91.6%	79.0KB	5.0M	92.2%	199.4KB	17.3M	92.7%	497.8KB	25.3M
Basic LSTM	92.0%	63.3KB	5.9M	93.0%	196.5KB	18.9M	93.4%	494.5KB	47.9M
LSTM	92.9%	79.5KB	3.9M	93.9%	198.6KB	19.2M	94.8%	498.8KB	48.4M
GRU	93.5%	78.8KB	3.8M	94.2%	200.0KB	19.2M	94.7%	499.7KB	48.4M
CRNN	94.0%	79.7KB	3.0M	94.4%	199.8KB	7.6M	95.0%	499.5KB	19.3M
DS-CNN	94.4%	38.6KB	5.4M	94.9%	189.2KB	19.8M	95.4%	497.6KB	56.9M

Table 3.2: Typical ARM-Cortex™-M based microcontroller development platform [Zhang u. a., 2018]

Arm Mbed™ platform	Processor	Frequency	SRAM	Flash
Mbed LPC11U24	Cortex-M0	48 MHz	8 KB	32 KB
Nordic nRF51-DK	Cortex-M0	16 MHz	32 KB	256 KB
Mbed LPC1768	Cortex-M3	96 MHz	32 KB	512 KB
Nucleo F103RB	Cortex-M3	72 MHz	20 KB	128 KB
Nucleo L476RG	Cortex-M4	80 MHz	128 KB	1 MB
Nucleo F411RE	Cortex-M4	100 MHz	128 KB	512 KB
FRDM-K64F	Cortex-M4	120 MHz	256 KB	1 MB
Nucleo F746ZG	Cortex M7	216 MHz	320 KB	1 MB

3.1.3 Conception

3.1.3.1 Choosing a feature extraction method

For feature extraction, these are two implementation methods according to section 2.1.2. We can use either spectrogram or MFCC as speech features illustrated in form

of an image. The image's resolution is equivalent to the number of input's parameters passed into the NN and is calculated as follow:

$$r = h \cdot w \tag{3.1}$$

with:

- w: image's width - number of frames
- h: image's height

We calculate w and h as in the following equations.

$$w = n_{frame} = (t_{window} - t_{frame})/t_{stride} + 1) \tag{3.2}$$

t_{window}, t_{frame} and t_{stride} are the timing window lengths. For Spectrogram, h is the number of FFT length:

$$h = n_{fft} = 2^n \tag{3.3}$$

For MFCC, h is the number of MFCCs we want to keep. $h = 13$ is feasible for most application with MFCC.
Using the spectrogram method will result in a bigger image with a denser resolution. As a result, our model's size would grow faster as we add more complexity to improve its performance. On the other hand, MFCC would take much longer to extract features as it is a further step in a signal processing chain. However, the output will be simplified with about a few hundred parameters. An example of extracting features of a single word "go" is demonstrated as follows in Table 3.3 and figure 3.2 as well as 3.3. The processing time and RAM usage are slightly affected by the total number of words of classes. Eventually, MFCC is more favorable and proven to be the more successful in automatic speech recognition in many works and practices.

Table 3.3: Benchmarking: Spectrogram vs MFCC

speech feature	**processing time [ms]**	**peak RAM usage [kB]**	**Number of classes**
spectrogram	77	27	4
spectrogram	81	28	16
spectrogram	81	28	19
MFCC	320	23	4
MFCC	368	26	16
MFCC	384	28	19

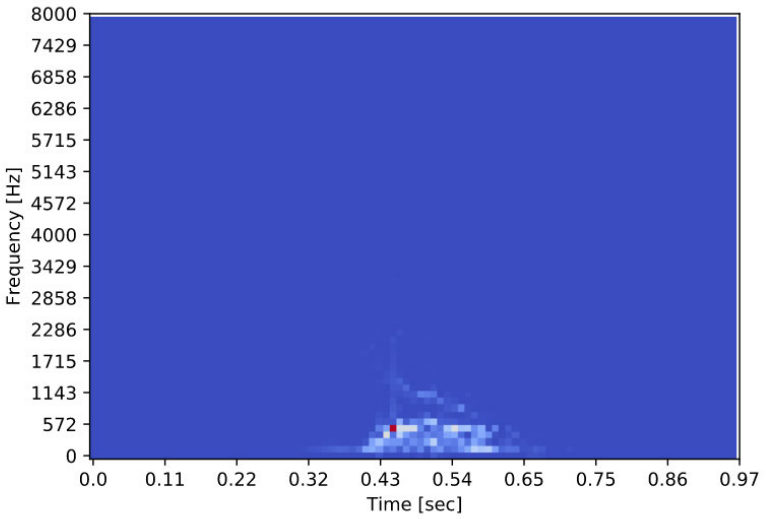

Figure 3.2: Spectrogram speech feature with the resolution of [128 x 50]

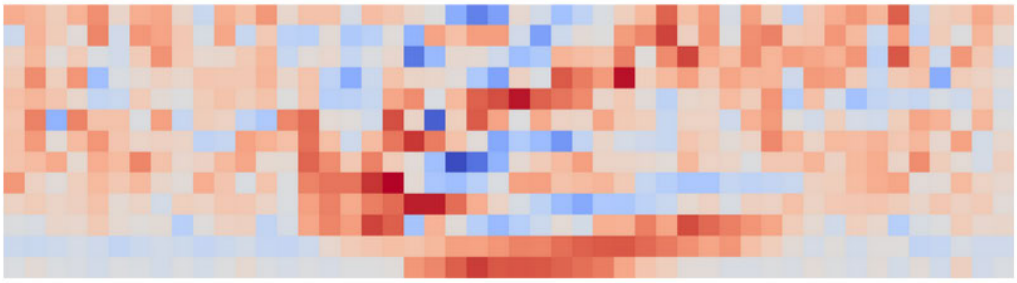

Figure 3.3: MFCC speech feature with the resolution of [13 x 50]

3.1.3.2 Choosing a NN model

There are many good developed models as mentioned in 3.1.2. The problem is that the TensorFlow-Lite kernel for microcontroller only supports some limited operations or architected layers, these are defined in this kernel header file "micro_ops.h". It means that some more advanced models like LTSM-NN would not be able to be applied yet. For this reason, this thesis will focus on developing a simple CNN consisting of the most used layers (Dense-, Dropout-, Conv-, Reshape-, Flaten-,...) in order to minimize the model's complexity. As the MCUs have very constrain resources, the more complex the model is, the less reactive our application is. The requirement on overall performance is set as low as the standard from Google-DNN-Model with an accuracy of about 84%. This accuracy is efficient for most applications.

```
1  /* Copyright 2019 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 ==============================================================================*/
15 #ifndef TENSORFLOW_LITE_MICRO_KERNELS_MICRO_OPS_H_
16 #define TENSORFLOW_LITE_MICRO_KERNELS_MICRO_OPS_H_
17
18 #include "edge-impulse-sdk/tensorflow/lite/c/common.h"
19
20 namespace tflite {
21 namespace ops {
```

```
22 namespace micro {
23
24 // Forward declaration of all micro op kernel registration methods. These
25 // registrations are included with the standard `BuiltinOpResolver`.
26 //
27 // This header is particularly useful in cases where only a subset of ops are
28 // needed. In such cases, the client can selectively add only the registrations
29 // their model requires, using a custom `(Micro)MutableOpResolver`. Selective
30 // registration in turn allows the linker to strip unused kernels.
31
32 TfLiteRegistration* Register_ABS();
33 TfLiteRegistration* Register_ADD();
34 TfLiteRegistration* Register_ARG_MAX();
35 TfLiteRegistration* Register_ARG_MIN();
36 TfLiteRegistration* Register_AVERAGE_POOL_2D();
37 TfLiteRegistration* Register_CEIL();
38 TfLiteRegistration* Register_CIRCULAR_BUFFER();
39 TfLiteRegistration* Register_CONV_2D();
40 TfLiteRegistration* Register_CONCATENATION();
41 TfLiteRegistration* Register_COS();
42 TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
43 TfLiteRegistration* Register_DEQUANTIZE();
44 TfLiteRegistration* Register_EQUAL();
45 TfLiteRegistration* Register_FLOOR();
46 TfLiteRegistration* Register_FULLY_CONNECTED();
47 TfLiteRegistration* Register_GREATER();
48 TfLiteRegistration* Register_GREATER_EQUAL();
49 TfLiteRegistration* Register_LESS();
50 TfLiteRegistration* Register_LESS_EQUAL();
51 TfLiteRegistration* Register_LOG();
52 TfLiteRegistration* Register_LOGICAL_AND();
53 TfLiteRegistration* Register_LOGICAL_NOT();
54 TfLiteRegistration* Register_LOGICAL_OR();
55 TfLiteRegistration* Register_LOGISTIC();
56 TfLiteRegistration* Register_MAXIMUM();
57 TfLiteRegistration* Register_MAX_POOL_2D();
58 TfLiteRegistration* Register_MEAN();
59 TfLiteRegistration* Register_MINIMUM();
60 TfLiteRegistration* Register_MUL();
61 TfLiteRegistration* Register_NEG();
62 TfLiteRegistration* Register_NOT_EQUAL();
63 TfLiteRegistration* Register_PACK();
64 TfLiteRegistration* Register_PAD();
65 TfLiteRegistration* Register_PADV2();
66 TfLiteRegistration* Register_PRELU();
67 TfLiteRegistration* Register_QUANTIZE();
68 TfLiteRegistration* Register_RELU();
69 TfLiteRegistration* Register_RELU6();
70 TfLiteRegistration* Register_RESHAPE();
71 TfLiteRegistration* Register_RESIZE_NEAREST_NEIGHBOR();
72 TfLiteRegistration* Register_ROUND();
73 TfLiteRegistration* Register_RSQRT();
74 TfLiteRegistration* Register_SIN();
75 TfLiteRegistration* Register_SOFTMAX();
76 TfLiteRegistration* Register_SPLIT();
77 TfLiteRegistration* Register_SQRT();
78 TfLiteRegistration* Register_SQUARE();
79 TfLiteRegistration* Register_STRIDED_SLICE();
80 TfLiteRegistration* Register_SUB();
81 TfLiteRegistration* Register_SVDF();
82 TfLiteRegistration* Register_UNPACK();
83 TfLiteRegistration* Register_L2_NORMALIZATION();
84 TfLiteRegistration* Register_TANH();
85
86 }  // namespace micro
87 }  // namespace ops
88 }  // namespace tflite
89
```

```
90 #endif  // TENSORFLOW_LITE_MICRO_KERNELS_MICRO_OPS_H_
```

3.1.3.3 Hardware and software selection

Programming and developing on microcontrollers is often very tricky. Before writing the codes, we have to consider the requirement of the targeted hardware and how we should build our software. When it comes to choosing a hardware kit or microcontroller, the CPU frequency and memory specifications are critical for every application. Applications processing a large amount of data, measured in kilos of Bytes, require at least a CPU of a few tens of kHz and a RAM with a minimum of $100kB$ of RAM. Read-only storage is not so critical as modern MCU often have at least a few MB of flash ROM. On the subject of software, should we write a bare-metal code or not? Writing bare-metal code, primarily written in C, is a low-level method of programming, which means our program will only run on the chosen OEM's hardware. The code is, therefore, more compact but is not reusable when we change the hardware. Modern embedded software nowadays is no more considered bare-metal. They are often built on top of SDKs, abstract libraries, or a complete framework. Many of these are written in C++ instead of C. The most well-known frameworks in recent years are Arduino and MbedOS, and both are C++-based. For this reason, TensorFlow Lite is also developed in C++ and aims for these two frameworks.

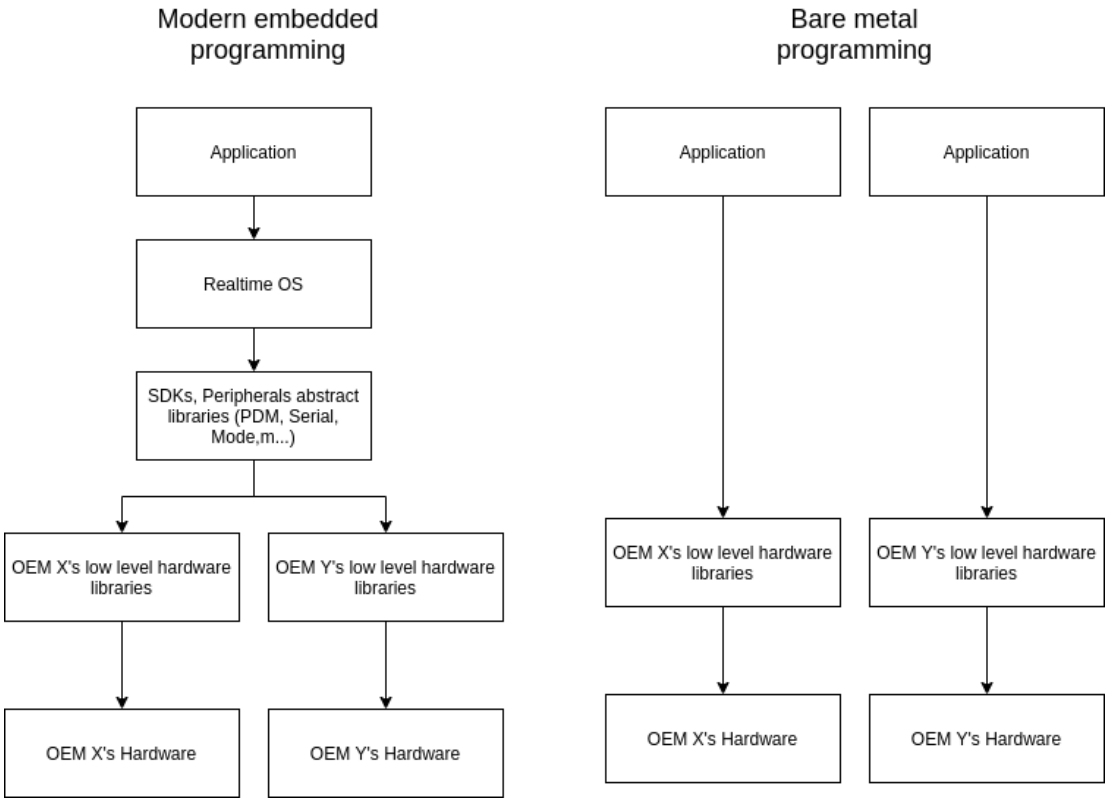

Figure 3.4: Embedded programming components - modern vs bare metal

At present, there are only a few development boards capable of realizing a speech recognition application. Among those boards, one of the most popular is the Arduino Nano 33 BLE Sense. This board is built with an integrated nRF52840-microcontroller based on ARM Cortex-M4 architecture. Another comparable choice is the B-L475E-IOT01A Discovery kit from the semiconductor manufacturer STM, which also has an ARM Cortex-M4-based MCU. The main differences between those two are CPU frequency, memory specifications, and the supported software packages. The Arduino board has more significant memory and is developed with the help of the Arduino SDK, which provides a more abstract and easy-to-use coding library. Therefore, this board is theoretically easier to use and may run a more complex model because of the bigger RAM. The most significant disadvantage is that it is harder to optimize code and hardware usage because the code almost always runs consecutively in a super loop. On the other hand, the board from STM has a better CPU frequency and is capable of

programming with MbedOS. MbedOS is the official SDK from ARM, providing a more mature programming interface and libraries. It is packed with more features, for example, the internal built real-time operating system, memory management, or abstract various peripherals drivers. These libraries, for example, the audio and microphone driver, are more challenging to set up and use but are very popular among professionals. In addition, it provides many options for optimization as well as the ability to write programs, which are event-driven or can run subtasks or threads simultaneously. An overview comparison between the two is in table 3.4.

Table 3.4: Hardware development kit specifications

Specification	**Arduino Nano 33 BLE Sense**	**B-L475E-IOT01A**
Processor	nRF52840	STM32L475
ARM Architecture	Cortex M4	Cortex M4
CPU Frequency	64kHz	80kHz
ROM	1MB	1MB
RAM	256kB	128kB
software SDK	Arduino	MbedOS
Microphone	MEMS Microphome	MEMS Microphome
Audio driver	Arduino's PDM	OEM Hardware kit's audio BSP
Application program type	single-process with super-loop	Realtime OS program
Onboard debugger	No	Yes
Optimization	No	intern DMA (faster DSP and audio acquisition)

Arduino Nano 33
BLE Sense

B-L475E-IOT01A

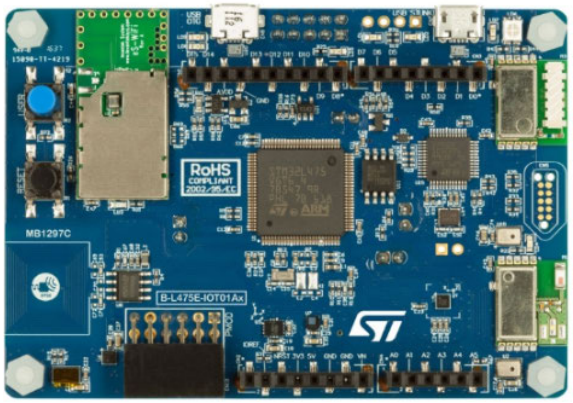

Figure 3.5: Hardware development kit

Both development kits from Arduino and STM are very similar in terms of hardware specifications. However, the STM board is more suitable and more efficient as the main focus of this work is to implement a speech-command interface that is reactive enough to use in real life. In addition, STM boards have more hardware acceleration methods, for example, the DMA, which provides more optimization for the program. Professionals also recommend boards with MbedOS support as the NN processing kernel on microcontrollers runs better on these boards.

3.1.3.4 Communication interface bus and application command interface

At the end of the inference process, we need to have a communication interface in order to output our prediction to a later process or system. As transport service, the following are the most common among embedded bus systems:

Table 3.5: Common embedded system bus

	I2c	**SPI**	**CAN**	**UART**
connection topology	bus, master-slave	serial master-slave	bus, star serial, ...	serial serial
transfering rate	100 kbit/s up to 400 kbit/s	limited by master clock	up to 1MBit/s	standard 9600 115200,..

The speech-command interface should run continuously and be able to output multiple predictions. For this reason, the demand for possessing communication lines is considerably high. Using a serial interface is, therefore, more appropriate. On the other hand, the prediction is sent out in message format, so the CAN bus and UART are better choices. UART is more favorable for ease of development between the twos as it is more straightforward and flexible.

Eventually, we need an abstraction interface for formatting messages sent from the SCI module. In networking, AT commands or Hayes command set is very popular. It is used as a common message-based or command-based communication between networking devices or modem with host devices like separated CPU or MCU. The message syntax is easy to understand as a simple request-response process and fully customizable. Standard commands are synchronous and always have a confirmation response, either with "OK" or "ERROR." Unsolicited or asynchronous message/command is also supported. Unsolicited commands do not require a request from host. Figure 3.6 and 3.7 illustrate the interaction between an AT module and the external host systems basing on the Hayes-command-set behaviors.

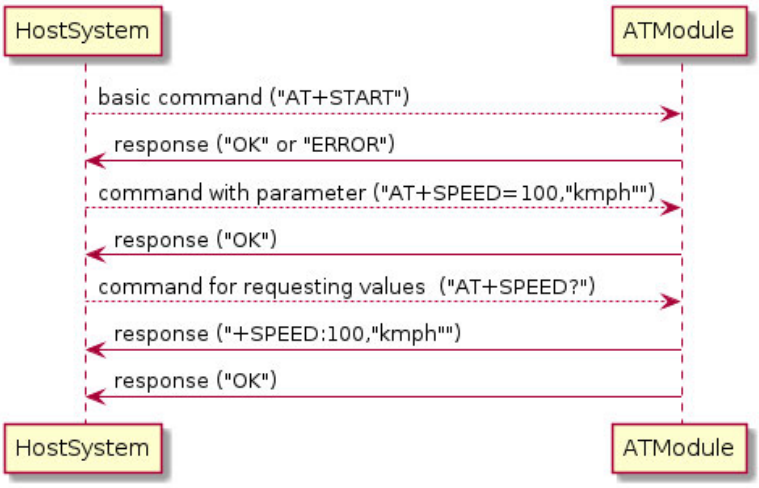

Figure 3.6: Synchronous AT command communication as a pair of request-response

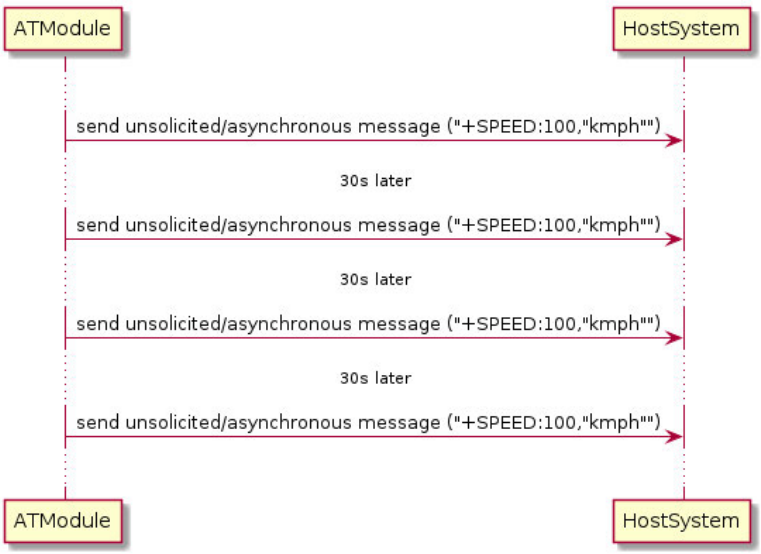

Figure 3.7: Asynchronous AT command communication - e.g. periodic message

3.2 Requirement

Combining all fundamental knowledge of speech processing, ML as well as the design proposal in previous sections, we summarize the requirements for the speech-command interface in this thesis as follows:

- Signal sample rate: at least $8kHz$
  We set the maximal signal frequency band limit B as $4000Hz$ according to the human's hearing range ($300Hz$ up to $3000Hz$). According to Nyquist–Shannon sampling theorem, the band limit must be at least half of the sampling frequency f_s so that the signal can be reconstructed back to the original, preventing information losses. In computer science, we often work with audio signal sample rates of $8kHz$, $16kHz$, and $32kHz$.

- Number of words: at least 10
  We build a speech-command interface upon a multiple-keywords-spotting system. The number of classified words must be at least ten to be comparable to the state-of-the-art similar ML models. The choice of words is spontaneous, as we will not define a specific application of SCI in this thesis.

- Response time: less than $1s$
  We set the maximal response time for each prediction as $1s$. Longer than this may be too slow and unusable for real-life applications. This duration includes digital signal processing time and the ML model inferencing time.

- Model accuracy: at least 80%
  Our model must be comparable to the standard DNN-model, whose performance is 84.6%.

- Memory consumption: maximal $64kB$ of RAM
  We chose the STM board for development, which has a limited RAM of $128kB$. As the main focus of the application is speech processing, we want to dedicate half of the memory resources to this task. Still, this is not precisely the feasible restriction, as many MbedOS libraries also have a large memory footprint that causes memory overflow. Our goal is to have a model as small as possible and at the same time perform comparably to the standard model. We calculate the memory consumption by getting the total sum of the DSP and inference peak RAM usage.

- Functionality
  Our SCI must be able to send out predictions by requests through a standard AT command. Furthermore, continuous asynchronous predictions as unsolicited messages are also supported. On the other hand, the host system using the SCI must be able to tune the predicting performance by, for example, setting the classification probability threshold, in which it decides if the class is qualified as the "GOOD" result or not.

4 Implementation

4.1 Modern Embedded ML development

As pointed out previously, developing an embedded application is very hard because of the poor portability based on the targeted platform. In addition, training and inferencing are two distinguished development processes needing many efforts to implement. We already have many software platforms and frameworks for training AI. In the case of inferencing, the TensorFlow lite still in early growing as its first release of the TensorFlow Lite micro-kernel in May 2019. As a result, it lacks productivity tools that connect training pipelines to deployment platforms and tools. Furthermore, deployed models on microcontrollers still consume much memory. For instance, a simple keyword spotting model may use up to tens of kB. This is considered quite intense for a microcontroller, not to mention the fact that, for an actual application, the model is much larger.

Measuring memory consumption on runtime is also tricky, requiring the skills of an experienced engineer or a specific development tool. Therefore, we will use another framework build on top of TensorFlow Lite to optimize further memory usage and performance on the chosen target in this work. Founded in 2019,- Edge impulse is a modern framework supporting developers, engineers, and domain experts solving embedded ML's problems. At the time of this thesis, it provides a set of development tools focusing on building and deploying ML models related to images and speech processing, and motion detection. What makes Edge Impulse the leading on the market is the EON compiler, which won the TinyML foundation's best innovation of the year in 2021. Instead of using the interpreter and the converted model, it is the extra step of compiling the model into more C++ codes and minimizing the interpreter's use. As a result, many model's metadata are hardcoded, many unused operators are also removed. The EON compiler does this automatically. For this, NN operators and buffers now use up to 55% less RAM and 35% less ROM, according to Edge Impulse. Figure 4.1 showing the benchmarking comparison of running NN using the EON compiler

versus the ones with the standard TensorFlow Lite. While there is a clear difference in memory usage, the computing latency remains the same between the twos.

Edge Impulse EON vs TensorFlow Lite for Microcontrollers

	RAM (KiB)	ROM (KiB)	Latency (M4F 80MHz)	Latency (M7 216MHz)
Continuous gestures TFLite (fully connected 33x20x10x4)	3.8	27.3	1 ms.	1 ms.
Continuous gestures EON (fully connected 33x20x10x4)	1.5	15.1	1 ms.	1 ms.
	60.5%	44.7%		
Audio scene recognition TFLite (1D CNN, 30x10)	12.8	60.5	22 ms.	6 ms.
Audio scene recognition EON (1D CNN, 30x10)	7	39.5	22 ms.	6 ms.
	45.3%	34.7%		
Keyword spotting TFLite (2D CNN, 10x5 + FC 12)	16	51.4	55 ms.	14 ms.
Keyword spotting EON (2D CNN, 10x5 + FC 12)	10.7	32.4	55 ms.	14 ms.
	33.1%	37.0%		
Image recognition TFLite (Transfer Learning, MobileNetV2 0.05, 32x32 grayscale)	130.6	292.7	-	40 ms.
Image recognition EON (Transfer Learning, MobileNetV2 0.05, 32x32 grayscale)	70.2	164.2	186 ms.	40 ms.
	46.2%	43.9%		
Image recognition TFLite (Transfer Learning, MobileNetV2 0.35, 96x96 RGB)	440.7	766.5	-	420 ms.
Image recognition EON (Transfer Learning, MobileNetV2 0.35, 96x96 RGB)	297.0	577.5	-	420 ms.
	32.6%	24.7%		

Figure 4.1: EON compiler benchmarking [Jan, 2020]

The framework generates C++ code from trained models, including the SDK for inferencing and feature extraction corresponding to the training session. For this reason,

the implementation will focus on choosing appropriate parameters for feature extraction, model architecture, and building an application running inference and outputting results through the communication interface to the host device.

4.2 Training a speech recognition model

4.2.1 Preparing the dataset

Before diving into the ML training process, we have to prepare the dataset first. We use the same method used by researchers as mention in the section 3.1.2, with an extra step. As recommended in this paper [McMahan und Rao, 2017], we mix all the labeled words with the background noises to improve the noise tolerance of the model. The whole dataset is separated into three subsets of training, validation, and testing set with a ratio of 6:2:2 respectively. Moreover, our model is a bit more complex as it contains 13 words comparing to the standard of 10.

4.2.2 Configuring feature's parameters and tuning model's parameters

The first step is to configure the feature extraction layer, which generates MFCC features as images. Based on the speech-command dataset from Google, our task is to predict only a single word as a command. Therefore, a window of $1s$ is reasonable for input audio signals. The sliding frame should be between $20ms$ and $40ms$, as mentioned in Chapter 2. The striding length can be set lower or equal to window length. Lower striding length means that we will have overlapped windows preventing information loss, but on the other hand, it also makes the model bigger. For instance, the image resolution calculated with equation 3.1 is more significant because the number of overlapped windows is more than non-overlapped.

The FFT length n_{fft} must be a power of 2, so the possible value is 128, 256, 512, etc. The bigger the n_{fft} is, the better contrast the output images is, as more information is extracted. A summary on model's performance by different n_{fft} values is listed on table 4.1. Except when n_{fft} is as low as 128, MFCCs features are identical for n_{fft} bigger than or equal to 256. The model's accuracy with an FFT length of 512 has an overall improvement of circa 3% compared to when n_{fft} equals 256.

Table 4.1: Resources comsumption based on n_{fft}

n_{fft}	**processing time [ms]**	**RAM usage [kB]**
128	225	16
256	278	19
512	384	27
1024	600	42

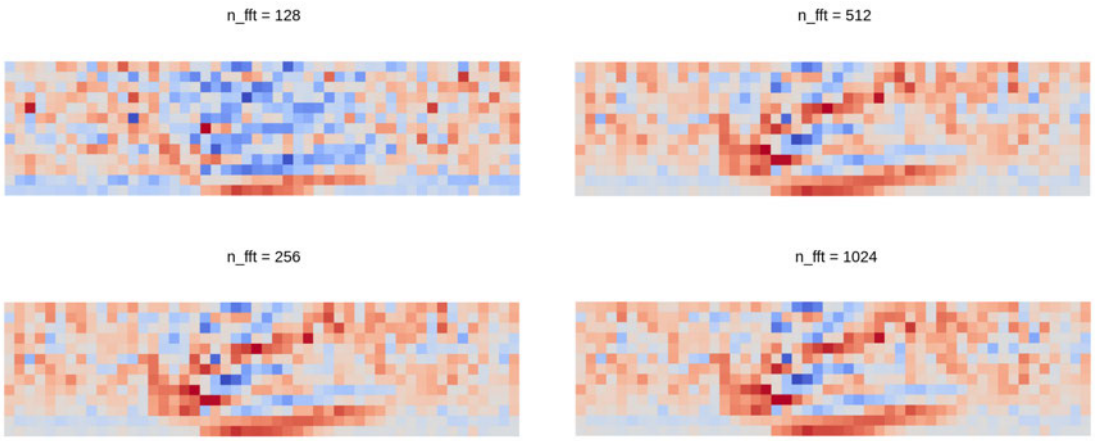

Figure 4.2: MFCCs by different n_{fft} with "go" as an example

The sample rate is $16000Hz$, so the band limit is set as $8000Hz$. We will keep 13 MFCCs from a Mel filterbank of 40, as this is suitable for most applications, including in this paper [Zhang u. a., 2018]. To summarize, the parameter applied in this thesis is as follows in Table 4.2. Using equation 3.1, we will have a feature with a resolution of 13x50 with a total number of 650 features.

Table 4.2: Feature extraction's parameter overview

parameter	**value**
window size	$1s$
framesize	$0.02s$
striding length	$0.02s$
nfft	512
Mel filterbank length	40
MFCC length	13

The second step is to tune the model hyperparameter, including the learning rate, batch size, and epoch number. These parameters affect the training session's performance. We set the learning rate as low as 0.005 and the epoch number as high as 200 for a finer training curve. A small batch size of 512 is, after trials, feasible for a fast learning speed without deteriorating the dataset.

The final step is to train the model after defining the input figure. As mentioned in chapter 3.1.3.2, we will focus on an uncomplicated CNN model based on the most basic layers. For learning the features, we have the convolutional layers. The most used kernel size for the CNN filters is 3x3, 5x5, or sometimes 7x7. Any larger kernel size may diminish the model's performance. Still, a smaller kernel size also means more calculations to perform. Another possible approach is to use 1D-CNN instead. This method is proven to be effective for processing time-series data or time-varying signals [Srinivasamurthy, 2018]. The most significant advantage of a 1D network, according to [Kiranyaz u. a., 2019], is the simple and compact configuration, as the kernel size and striding size are both defined with just a number instead of 2. The kernel strides only in one direction, which means no nested loop is required for iterating through the inputs. As a result, the 1D convolution operator executes only scalar multiplications and additions, making the algorithm faster and lighter. For this reason, 1D CNN is recommended for real-time application and works well on low-cost hardware. In this thesis, we will use 1D CNN as the primary learning layer. In between, we will have some pooling layer to downsample the outputs further. Dropout layers are also used to avoid overfitting [Srivastava u. a., 2014].

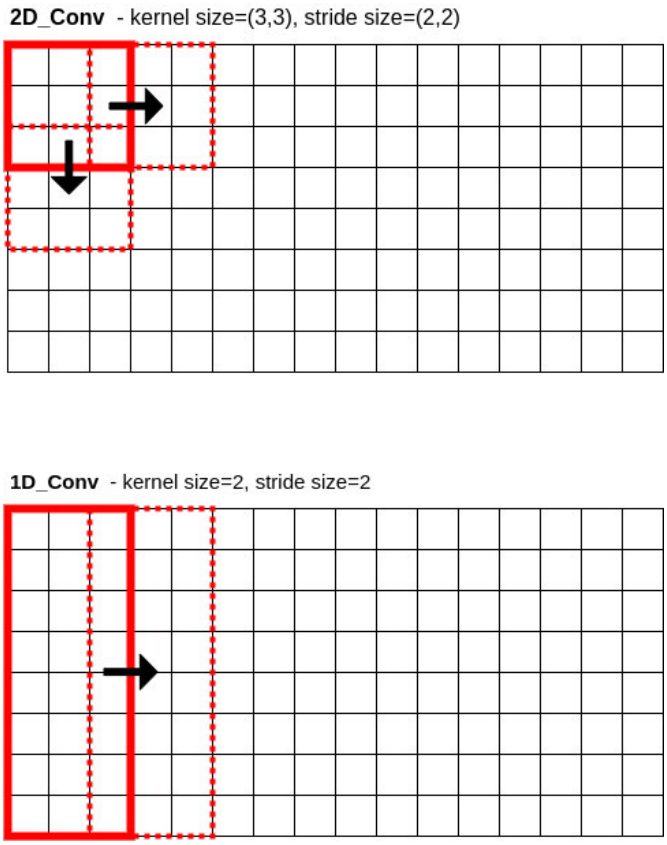

Figure 4.3: 1D vs 2D convolution

4.3 Running inference

After training and generating C++ code, The final step is to build an SCI application and deploy the model on targeted hardware platform. The compiled model contains the C++ inference and DSP SDK as well as the AI model structure in C++ code.

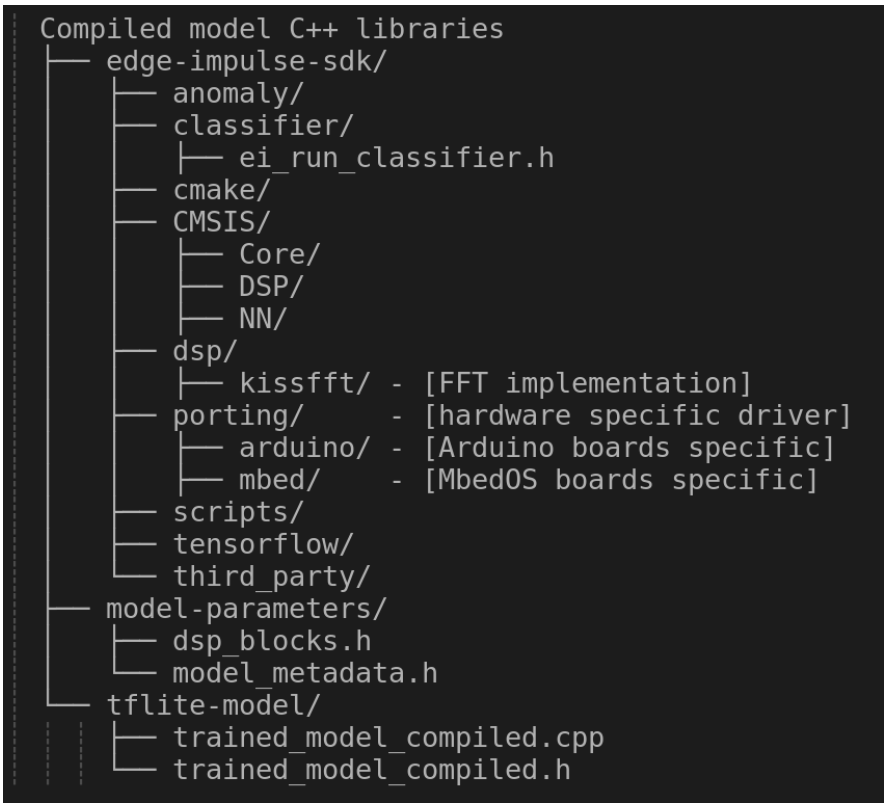

Figure 4.4: Compiled C++ SDK library folder structure

We import these codes into our source and compile them all together with the main program. Our main program is as simple as the following flow chart. After defining the needed commands, the program will listen to the host system's requests and handle them internally.

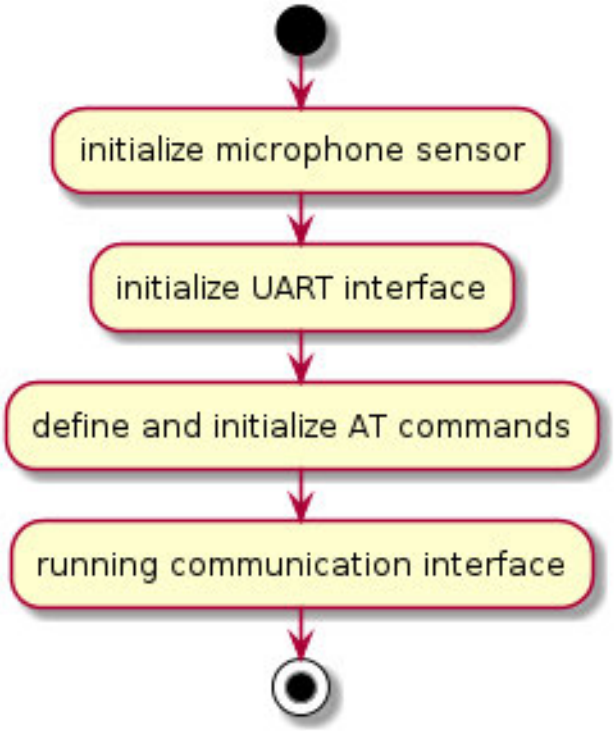

Figure 4.5: Main program's flow chart

Each command corresponds to a function of the SCI. We describe the most important functionalities as follows:

- AT+CLASSLIST: sending out a list of prediction classes/words
- AT+PTHRES=<float>: get/set prediction probability threshold. If a class is more than the set value, it is considered "GOOD" and sent out to the host system.
- AT+PFILTER=<0:false or 1:true>: choose if the output contains all classes or just the best or the only "GOOD" class.
- AT+RUNSINGLE: get a single prediction
- AT+RUNCONT: run prediction continuously. Results are sent out as unsolicited messages.
- +UPCLA: classification output from SCI (normal or unsolicited)

4.3.1 Running single prediction

When the host system request only a single prediction, the SCI applies the signal processing pipe from end to end sequentially. Firstly, it records the audio samples into a data buffer. This buffer contains the PCM data or discrete audio signal. Then all the data are passed through the inference pipes to extract the MFCC features and, finally, the classification results. By default, we will filter out all class outputs, except the class with the highest rating and is considered as "GOOD."

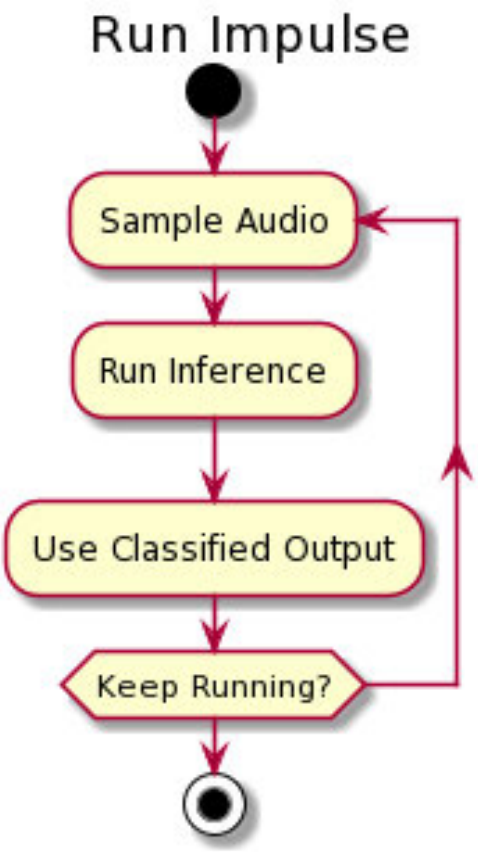

Figure 4.6: Flow chart: Run single prediction

4.3.2 Running inference continuously

In the case of continuous inferencing, things get a little bit more complicated. We have to keep recording audios while running inference at the same time. On most

STM32 boards, with the help of the DMA, audio buffers are accessed directly without interrupting CPU operations. We divide the standard $1s$ window into n smaller blocks, for example, by four blocks. By doing this, we hardly missed any event. We get the input audio by concatenating the newest n blocks. However, doing this means it may consume more memory because, at a time, several inferences are running.

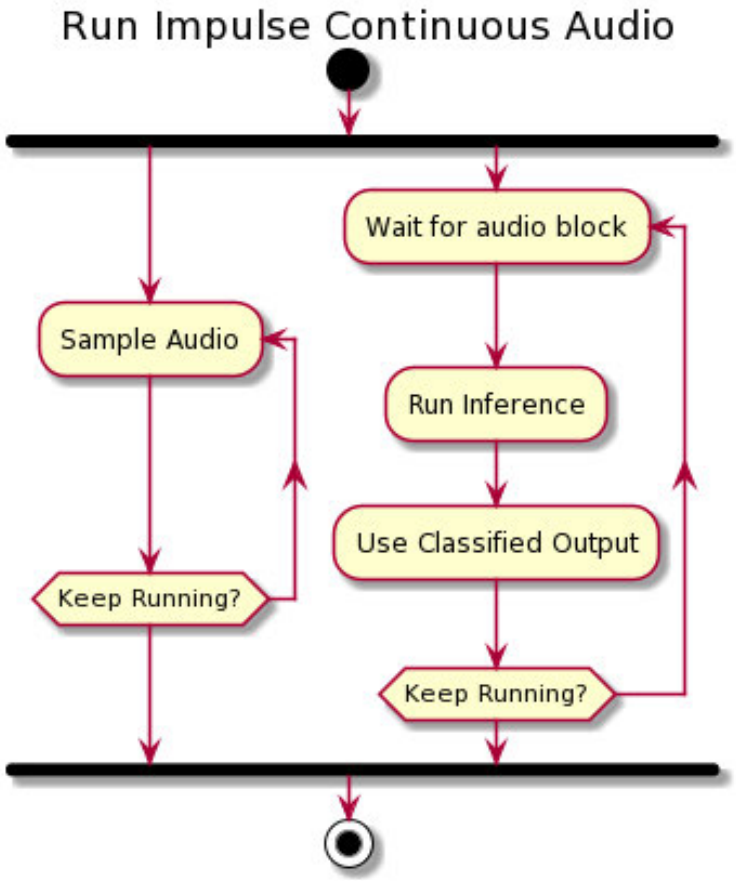

Figure 4.7: Flow chart: Run continuous inference

5 Experiment and Result

5.0.1 Final ML model

After carrying out the development steps in chapter 4, we have accomplished a model with an accuracy of 85%, which is comparable to the standard Google model CNN-2. Table 5.1 also lists other well-known models on the same dataset for comparison. Our SCI's rankings are among all at number 4 of accuracy and rank 1st on the memory benchmark.

Table 5.1: ML models comparison

NN Architecture	**Accuracy [%] - rank**	**Memory [kB] - rank**
DNN [Chen u. a., 2014]	84.3 - 6	288 - 4
CNN-1 [Sainath und Parada, 2015]	90.7 - 1	556 - 6
CNN-2 [Sainath und Parada, 2015]	84.6 - 5	149 - 3
LSTM [Sun u. a., 2016]	88.8 - 2	26.0 - 2
CRNN [Arik u. a., 2017]	87.8 - 3	298 - 5
SCI	85.0 - 4	13.8 - 1

We have built our model as simple as in the figure 5.2 illustrated. With this architecture, we have optimized the complexity and, at the same time, achieved a decent prediction accuracy. The model is also not overfitting by validation.

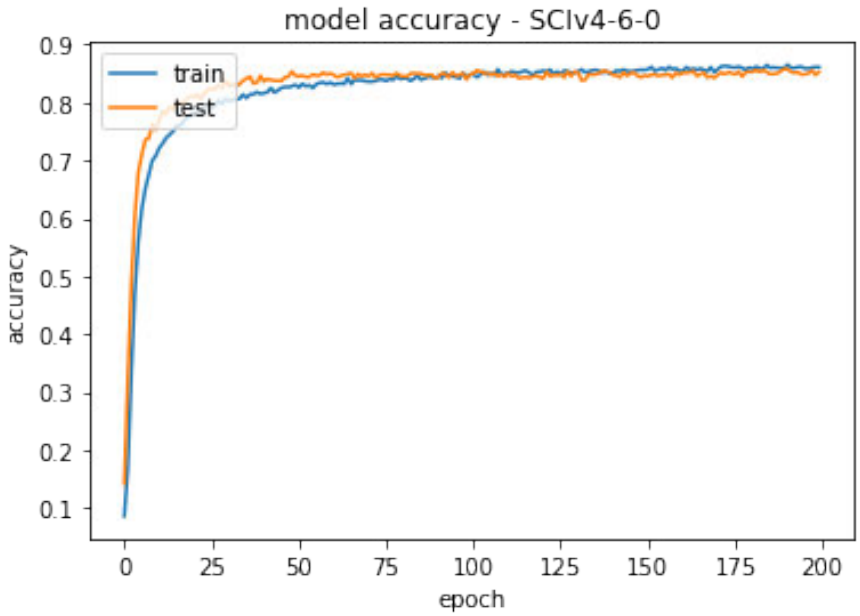

Figure 5.1: SCI model accuracy

```
Epoch 200/200
28/28 - 6s - loss: 0.4864 - accuracy: 0.8538 - val_loss: 0.4999 - val_accuracy: 0.8546
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape_3 (Reshape)          (None, 50, 13)            0
_________________________________________________________________
conv1d_21 (Conv1D)           (None, 50, 64)            2560
_________________________________________________________________
dropout_9 (Dropout)          (None, 50, 64)            0
_________________________________________________________________
conv1d_22 (Conv1D)           (None, 50, 64)            12352
_________________________________________________________________
conv1d_23 (Conv1D)           (None, 50, 64)            12352
_________________________________________________________________
max_pooling1d_9 (MaxPooling1 (None, 25, 64)            0
_________________________________________________________________
dropout_10 (Dropout)         (None, 25, 64)            0
_________________________________________________________________
conv1d_24 (Conv1D)           (None, 25, 32)            6176
_________________________________________________________________
conv1d_25 (Conv1D)           (None, 25, 32)            3104
_________________________________________________________________
conv1d_26 (Conv1D)           (None, 25, 32)            3104
_________________________________________________________________
max_pooling1d_10 (MaxPooling (None, 13, 32)            0
_________________________________________________________________
conv1d_27 (Conv1D)           (None, 13, 16)            1552
_________________________________________________________________
max_pooling1d_11 (MaxPooling (None, 7, 16)             0
_________________________________________________________________
dropout_11 (Dropout)         (None, 7, 16)             0
_________________________________________________________________
flatten_3 (Flatten)          (None, 112)               0
_________________________________________________________________
y_pred (Dense)               (None, 15)                1695
=================================================================
Total params: 42,895
Trainable params: 42,895
Non-trainable params: 0
```

Figure 5.2: SCI model summary

```
# model architecture
model = Sequential()
model.add(Reshape((int(input_length / 13), 13), input_shape=(input_length, )))
model.add(Conv1D(64, kernel_size=3, activation='relu', padding='same'))
model.add(Dropout(0.4))
model.add(Conv1D(64, kernel_size=3, activation='relu', padding='same'))
model.add(Conv1D(64, kernel_size=3, activation='relu', padding='same'))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.4))
model.add(Conv1D(32, kernel_size=3, activation='relu', padding='same'))
model.add(Conv1D(32, kernel_size=3, activation='relu', padding='same'))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Conv1D(32, kernel_size=3, activation='relu', padding='same'))
model.add(Conv1D(16, kernel_size=3, activation='relu', padding='same'))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.4))
model.add(Flatten())
model.add(Dense(classes, activation='softmax', name='y_pred'))
```

Figure 5.3: SCI model architecture made with TensorFlow

Figure 5.4 is the final confusion matrix showing classification results. The class labeled as "__Unknown__" is the worse performing class because it contains randomly sampled words and not correctly labeled. Furthermore, this AI model often misclassifies the word "Off" with the two words "Up" and "On ." This shows that our model resembles how a person senses these words as they sound pretty identical to our ears. False predictions also occur when classifying the other group of the identical words, namely the "Go"-"Down"-"No" group as well as the word pair "Yes"-"Left."

	_NOISE	_UNK	BACK	DOWN	FORW	GO	LEFT	NO	OFF	ON	RIGHT	STOP	UP	VISUAL	YES
_NOISE	99.2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.4%	0.4%	0%	0%
_UNKNOWN	3.9%	56.3%	2.6%	5.2%	9.6%	3.1%	1.3%	2.2%	4.4%	2.6%	3.5%	1.7%	1.7%	1.3%	0.4%
BACKWARD	2.5%	2.1%	86.8%	0.8%	0.8%	0.8%	1.7%	0%	0.8%	0%	0.4%	0.4%	2.9%	0%	0%
DOWN	1.6%	2.8%	0%	89.9%	0.4%	0.8%	0.4%	2.8%	0%	0%	0.4%	0.4%	0.4%	0%	0%
FORWARD	0.7%	2.6%	0.4%	0%	88.5%	2.6%	0%	0%	2.6%	2.2%	0%	0%	0%	0.4%	0%
GO	3.7%	2.5%	0%	6.2%	1.2%	80.2%	0%	3.3%	1.2%	0%	0%	0.4%	0.8%	0.4%	0%
LEFT	0.8%	5.9%	0.4%	0.8%	0.8%	0.4%	85.7%	0%	0.4%	0%	0%	0.4%	0%	0.4%	3.8%
NO	0.9%	4.7%	0%	5.2%	0.4%	5.2%	0.9%	79.7%	1.3%	0%	0%	0.9%	0%	0.4%	0.4%
OFF	0.9%	0.9%	0%	0%	1.4%	1.4%	0.5%	0%	88.3%	0.9%	0%	0.5%	5.1%	0%	0%
ON	0.9%	4.8%	0%	0.9%	3.0%	0%	0.4%	0%	6.1%	82.6%	0%	0%	1.3%	0%	0%
RIGHT	2.1%	6.2%	0.4%	0%	0%	0.8%	2.9%	0%	0%	0%	86.7%	0.4%	0%	0%	0.4%
STOP	1.8%	2.3%	0%	3.2%	0%	0.9%	0%	0%	0.9%	0%	0%	88.2%	2.7%	0%	0%
UP	1.6%	1.6%	0.4%	0.8%	0%	0.8%	0.8%	0%	13.8%	0.4%	0%	2.0%	78.0%	0%	0%
VISUAL	0.4%	5.4%	1.3%	0.4%	0.4%	1.3%	2.1%	0%	0%	0%	0%	0.4%	0%	87.0%	1.3%
YES	2.0%	2.7%	0%	0.8%	0.4%	0%	3.1%	0.8%	0%	0%	0%	0%	0%	0.4%	89.8%
F1 SCORE	0.89	0.55	0.90	0.84	0.87	0.81	0.86	0.84	0.79	0.87	0.91	0.90	0.81	0.91	0.92

Figure 5.4: SCI classification result as confusion matrix

5.0.2 Inferencing - SCI in action

Following is the logged output from the SCI in action. We build a simple test by connecting the development board to a PC acting as a host system for SCI. For this, we use some utility software for accessing the UART communication through the USB. Host system's messages or requests begin with a specific character ">." It is essential to notice the actual inferencing performance. By the case of single inference, the total response time is circa $650ms$. After triggering a request, SCI will open a $1s$ window for recording and then return a prediction. In the following example, starting from line 46, SCI sends the output in a class name pair with its probability in continuous running mode. In this case, we receive the prediction almost immediately. The more blocks we divide the audio input window, the more sensitive SCI is and more RAM when operating. If the SCI is too sensitive, it may send out multiple identical predictions right after each

other. This can be prevented on the side of the host system or internally by SCI by changing the threshold value or reduce the number of inferencing blocks.

The total RAM usage is $41kB$, respectively $27kB$ for feature extraction (see table 4.1) and circa $14kB$ for inferencing ML model (see table 5.1).

```
1
2  > AT+HELP
3  AT+HELP - Lists all commands
4  AT+RESET - Reset the system
5  AT+RUNSINGLE - Run a single prediction
6  AT+RUNCONT - Run the impulse continuously
7  AT+CONFPOUTPUT= - set prediction output format (1 parameter)
8  AT+CONFPOUTPUT? - get prediction output format
9  AT+PTHRES= - set prediction probability threshold (1 parameter)
10 AT+PTHRES? - get prediction probability threshold
11 AT+PFILTER= - config get only good prediction (affect only SCI FORMAT) (1 parameter)
12 AT+DOUTPUTEN= - enable debug output (1 parameter)
13 AT+CLASSLIST - enable debug output
14 OK
15
16 > AT+RUNSINGLE
17 [DEBUG] Recording
18 [DEBUG] Recording OK
19 [DEBUG] Starting inferencing
20 [DEBUG] Predictions (DSP: 447 ms., Classification: 102 ms., Anomaly: 0 ms.):
21     _noise: 0.00000
22     _unknown: 0.00391
23     backward: 0.00000
24     down: 0.00000
25     forward: 0.99609
26     go: 0.00000
27     left: 0.00000
28     no: 0.00000
29     off: 0.00000
30     on: 0.00000
31     right: 0.00000
32     stop: 0.00000
33     up: 0.00000
34     visual: 0.00000
35     yes: 0.00000
36 OK
37
38 > AT+RUNSINGLE
39 +UPCLA=stop,0.99609,GOOD
40 OK
41
42 > AT+PFILTER=1
43 OK
44
45 > AT+RUNCONT
46 +UPCLA=backward,0.99609
47 +UPCLA=backward,0.99609
48 +UPCLA=backward,0.90234
49 +UPCLA=off,0.90234
50 +UPCLA=forward,0.81250
51 +UPCLA=forward,0.83984
52 +UPCLA=left,0.87891
53 +UPCLA=stop,0.82422
54 +UPCLA=up,0.93750
55 +UPCLA=down,0.91016
56 +UPCLA=backward,0.99609
57 +UPCLA=backward,0.99609
58 +UPCLA=backward,0.99609
59 +UPCLA=left,0.81641
60 +UPCLA=visual,0.99609
```

```
61   +UPCLA=visual,0.99609
62   +UPCLA=visual,0.99219
63   +UPCLA=up,0.82031
```

6 Conclusion

6.1 Summary

In this thesis, we have successfully built an experimental speech-command interface on a microcontroller. The implementation includes defining and training a speech recognition ML model and deploying this model on the targeted platform built upon a microcontroller. Our model uses the MFCC as speech features as it is commonly used and proven to be effective in many applications. MFCC feature is represented as a 3D tensor or picture, so the model we choose is similar to image processing using a CNN-based model. Instead of a standard 2D convolutional kernel, we apply the 1D convolution operator for extracting information from input since this method helps to reduce the model size even more without losing much performance. Parallel to the training process, we have also created an embedded software running on our development kit, the B-L475E-IOT01A from STM. We use another framework built upon the standard TensorFlow Lite for microcontrollers to optimize memory usage further. The model after training is compiled into C++ codes in the form of a native library easily imported to any C++ project. In the end, we have achieved a tiny 1D-Conv model consuming minimal RAM usage of $13,8kB$ and perform just as well as the standard DNN model. Our model has an accuracy of 85%, and the total memory usage in RAM is $41kB$, including the RAM needed for DSP. With this small footprint, we expect our model to be deployable on most microcontrollers systems with architecture not lower than the ARM-Cortex™-M4. Considering the compactness in model design with 1D CNN and the light-weighted memory usage, the model of SCI fulfills the requirements well and is feasible in many use cases. By design, we built the SCI as an individual speech processing module with its own hardware and software specifications, interfacing with the external host system through a serial communication or UART. We use the AT command as the application message protocol for outputting the speech command prediction. The host system may consume these messages for its application.

6.2 Discussion

Building a speech recognition application requires deep knowledge of phonetics and signal processing. The nature of human speech is also sophisticated; therefore, the amount of computation is costly. For most applications, accuracy is not always the most crucial requirement as speech may not be used as the only method to help humans interact with devices. However, in real life, the prediction precision is expected as high as at least 80% to be usable. This thesis is just an experimental implementation of a speech-command interface on microcontrollers, and one of our aims is to evaluate the possibility of the new technology, TinyML. The era of embedded ML is becoming more and more exciting after the first publication of the "TinyML" book in 2019. Since then, many efforts and achievements have been made, providing more tools and more optimization, making AI applications on microcontrollers more secure, more robust, and easier to get started. On the software side, we have used in this work the EON compiler, which won the TinyML foundation's best innovation of the year in 2021. making the AI model more compact, and consume less memory on MCUs. On the hardware side, the new ARM-Cortex™-M55 architecture is very promising. It is designed particularly for embedded AI, accelerating NN operator and inference speed for up to 15 times better than the most powerful embedded kernel ARM-Cortex™-F7, along with the benefits of energy efficiency.

In conclusion, after evaluating the use of TinyML in our trial design for SCI, our opinion is that embedded ML is still not a fully mature technology. However, it will be more active in both academic and practical trials in the future. We expect to see more presence of TinyML applications in everyday life very soon, considering its fast-growing pace in development.

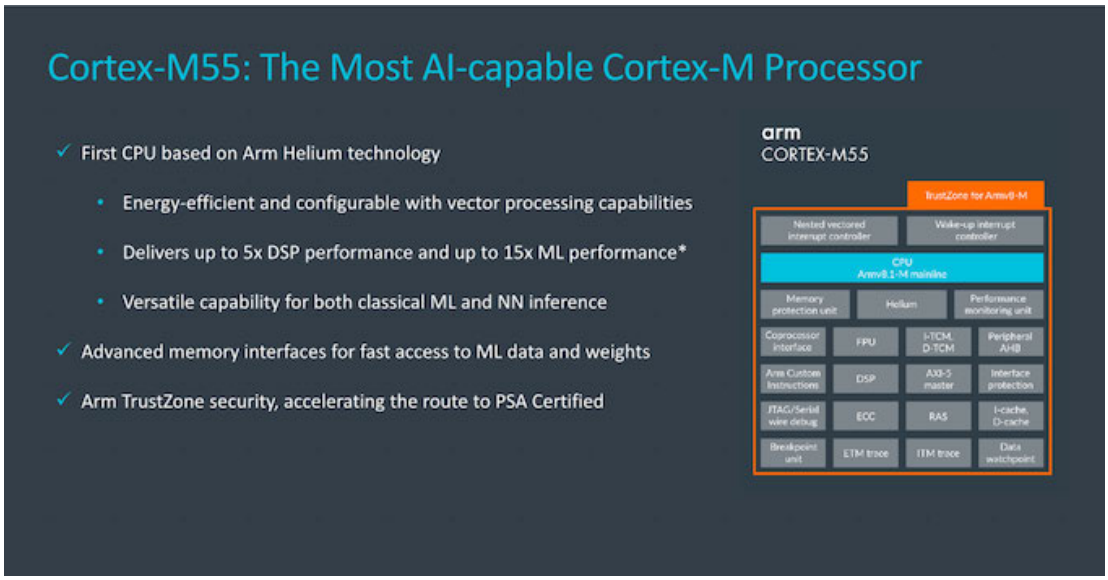

Figure 6.1: An overview of the new ARM-Cortex™-M55 [Frumusanu, 2020]

Bibliography

[Andrade-Miranda 2017] ANDRADE-MIRANDA, Gustavo: *Analyzing of the vocal fold dynamics using laryngeal videos*, Dissertation, 06 2017

[Arik u. a. 2017] ARIK, Sercan O. ; KLIEGL, Markus ; CHILD, Rewon ; HESTNESS, Joel ; GIBIANSKY, Andrew ; FOUGNER, Chris ; PRENGER, Ryan ; COATES, Adam: *Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting*. 2017

[Chen u. a. 2014] CHEN, Guoguo ; PARADA, Carolina ; HEIGOLD, G.: Small-footprint keyword spotting using deep neural networks. In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2014), S. 4087–4091

[Cormen u. a. 2009] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L. ; STEIN, Clifford: *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. – ISBN 0262033844

[Fayek 2016] FAYEK, Haytham M.: *Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between*. 2016. – URL https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html

[Frumusanu 2020] FRUMUSANU, Andrei: *Arm announces Cortex-M55 Core and Ethos-U55 microNPU*. 2020. – URL https://www.anandtech.com/show/15494/arm-announces-cortexm55-core-and-ethosu55-micronpu

[Goodfellow u. a. 2015] GOODFELLOW, I. ; BENGIO, Yoshua ; COURVILLE, Aaron C.: Deep Learning. In: *Nature* 521 (2015), S. 436–444

[Gupta u. a. 2013] GUPTA, Shikha ; JAAFAR, J. ; FATIMAH, W. ; BANSAL, A.: FEATURE EXTRACTION USING MFCC. In: *Signal & Image Processing : An International Journal* 4 (2013), S. 101–108

[Jan 2020] JAN, Jongboom: *Introducing EON: Neural Networks in Up to 55% Less RAM and 35% Less ROM*. 2020. – URL https://www.edgeimpulse.com/blog/introducing-eon

[Jurafsky und Martin 2000] JURAFSKY, Dan ; MARTIN, James H.: Speech and language processing - an introduction to natural language processing, computational linguistics, and speech recognition. In: *Prentice Hall series in artificial intelligence*, 2000

[Kiranyaz u. a. 2019] KIRANYAZ, S. ; AVCI, Onur ; ABDELJABER, Osama ; INCE, T. ; GABBOUJ, M. ; INMAN, D.: 1D Convolutional Neural Networks and Applications: A Survey. In: *ArXiv* abs/1905.03554 (2019)

[LeCun u. a. 1998] LECUN, Y. ; BOTTOU, L. ; BENGIO, Yoshua ; HAFFNER, P.: Gradient-based learning applied to document recognition, 1998

[Liu u. a. 2019] LIU, Weiqiang ; LIAO, Qicong ; QIAO, F. ; XIA, W. ; WANG, Chenghua ; LOMBARDI, F.: Approximate Designs for Fast Fourier Transform (FFT) With Application to Speech Recognition. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66 (2019), S. 4727–4739

[McMahan und Rao 2017] MCMAHAN, Brian ; RAO, Delip: *Listening to the World Improves Speech Command Recognition*. 2017

[Sainath und Parada 2015] SAINATH, T. ; PARADA, Carolina: Convolutional neural networks for small-footprint keyword spotting. In: *INTERSPEECH*, 2015

[Shawn Hymel 2020] SHAWN HYMEL, Alexander Fred-Ojala: *Introduction to Embedded Machine Learning*. 2020. – URL https://www.coursera.org/learn/introduction-to-embedded-machine-learning?

[Shi u. a. 2016] SHI, Weisong ; CAO, Jie ; ZHANG, Quan ; LI, Youhuizi ; XU, Lanyu: Edge Computing: Vision and Challenges. In: *IEEE Internet of Things Journal* 3 (2016), Nr. 5, S. 637–646

[Srinivasamurthy 2018] SRINIVASAMURTHY, Ravisutha S.: Understanding 1D Convolutional Neural Networks Using Multiclass Time-Varying Signals, 2018

[Srivastava u. a. 2014] SRIVASTAVA, Nitish ; HINTON, Geoffrey E. ; KRIZHEVSKY, A. ; SUTSKEVER, Ilya ; SALAKHUTDINOV, R.: Dropout: a simple way to prevent neural networks from overfitting. In: *J. Mach. Learn. Res.* 15 (2014), S. 1929–1958

[STMicroelectronics 2019] STMICROELECTRONICS: *Application Note: Interfacing PDM digital microphones using STM32 MCUs and MPUs*. 2019

[Sun u. a. 2016] SUN, Ming ; RAJU, Anirudh ; TUCKER, George ; PANCHAPAGESAN, Sankaran ; FU, Gengshen ; MANDAL, Arindam ; MATSOUKAS, Spyros ; STROM, Nikko ; VITALADEVUNI, Shiv: Max-pooling loss training of long short-term memory networks for small-footprint keyword spotting. In: *2016 IEEE Spoken Language Technology Workshop (SLT)* (2016), Dec. – URL http://dx.doi.org/10.1109/SLT.2016.7846306. ISBN 9781509049035

[Warden 2018] WARDEN, Pete: *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018

[Warden und Situnayake 2019] WARDEN, Pete ; SITUNAYAKE, Daniel: TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers, 2019

[Zhang u. a. 2018] ZHANG, Yundong ; SUDA, Naveen ; LAI, Liangzhen ; CHANDRA, Vikas: *Hello Edge: Keyword Spotting on Microcontrollers*. 2018

Glossar

BSP Board support package

CNN Convolutional neural network

DCT Discrete Cosinus Transform

DFT Discrete Fourier Transformation

DL Deep learning

DMA Direct memory access

DNN Deep Neural network

DSP Digital signal processing

FFT Fast Fourier Transformation

FPU floating-point processing unit

HMI Human-Machine-Interface

ICT Information and communications technology

IoT Internet of things

LSTM-NN Long short-term-memory neural network

MCU Microcontroller

MFCC Mel Frequency Cepstral Coefficients

MFE Mel-Filterbank Energie

ML Maschine learning

NN Neural network

OEM Original equipment manufacturer

PCM Pulse Code Modulator

PDM Pulse Sensity Modulator

RAM Random access memory

SCI Speech-Command-Interface

SDK Software development kit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

[redacted]

[redacted]

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: [redacted]

Vorname: [redacted]

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:
Implementation of a Speech-command-interface on Microcontroller with TinyML

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der -bitte auswählen- ist erfolgt durch:

[redacted] [redacted] [redacted]

Ort Datum