UCLouvain

epl

École polytechnique de Louvain

Implementation and characterization of a machine learning algorithm for bat detection running on a low-power microcontroller

Author: **Thomas ANTOINE**
Supervisor: **David BOL**
Reader: **Martin ANDRAUD, Laurent JACQUES, Pol MAISTRIAUX**
Academic year 2023–2024
Master [120] in Electrical Engineering

Abstract

Biodiversity has long been a concern for environmental authorities, and bat species have not been spared from the global biodiversity loss. Acoustic surveys are the main method for censuses of bat faunas, but their use has remained limited, partly due to their cost and the difficulties of call detection. Several algorithms using machine learning to detect bat calls have been developed and have shown great performance. However, most of them are designed for computers and cannot be implemented on low-power embedded devices for real-time operation.

In this work, an algorithm for bat detection inspired by state-of-the-art algorithms is designed and implemented for a platform based on a Cortex-M4. Multiple design choices are made to speed up computation, including the use of mel-scaled filterbanks and quantization of the model. The dataset to train the algorithm is collected in Louvain-la-Neuve with the device, and a teacher model is used to generate annotations. This dataset is used to train the CNN in the algorithm and to optimize various parameters.

The resulting system requires 11.47 ms to process 9.8 ms of audio, while consuming 7.45 mA in execution and 115 µA in standby. Taking the sensing duty cycle into account, the proposed system is estimated to have an average precision between 60.29 % and 69.55 % on the call detection task, and has been validated in the field to be able to detect bat calls. Performance improvements based on hardware modifications and possible applications of the proposed solution are discussed.

Acknowledgements

I would like to thank Prof. David Bol and Pol Maistriaux for their guidance, advice and constant availability throughout the year. Through regular meetings, their views and expertise have helped me to progress in my work, to surpass my preconceived ideas and to explore avenues I would not have thought of without them.

Thanks to Nicolas Brusselmans for his prompt answers to my questions about the platform and to Souley Djadjandi for his help in re-soldering the battery pack to the board.

This work would not have been possible without Bat Detective, which served as the main source of inspiration for the algorithm, and BatDetect2, which proved to be a great tool for detecting bat calls.

I am very grateful to my friends and family for their support. Thanks to Marine Boxus, Gauthier Galez and Maxime Vanliefde for their help in accessing recording sites. Thanks to my sister, Charlotte Antoine, for her proofreading of the whole thesis. Finally, I would like to thank my parents, who have supported me throughout my studies and have always been there in times of need.

DeepL Write [1] was used in the writing of this document.

Contents

Introduction **1**

1 System overview **7**
1.1 Main board . . . 8
1.2 Ultrasonic AFE and microphone . . . 9
1.3 Memory storage . . . 11
1.4 Power budget . . . 12

2 Dataset creation **15**
2.1 Data collection through the UART . . . 15
2.1.1 Setup . . . 15
2.1.2 Result . . . 15
2.1.3 Discussion . . . 16
2.2 Data collection through the microSD card . . . 17
2.2.1 microSD card write throughput increase . . . 17
2.2.2 Continuous recording setup . . . 18
2.3 Supply noise analysis and attenuation . . . 18
2.3.1 Measurements setup . . . 19
2.3.2 Results . . . 21
2.3.3 Discussion . . . 21
2.4 Annotation . . . 22
2.4.1 Methodology . . . 22
2.4.2 Selection of the teacher model . . . 22
2.5 On-site measurements . . . 23
2.5.1 Setup . . . 23
2.5.2 Results . . . 25
2.5.3 Discussion . . . 28

3 Algorithm design and implementation **31**
3.1 State of the art of bat detection algorithms . . . 31
3.2 Chosen algorithm . . . 33
3.2.1 Problem setting . . . 33
3.2.2 Feature extraction pipeline . . . 33
3.2.3 Machine learning model architecture . . . 35

3.3 Implementation of a detection algorithm 36
3.3.1 Feature extraction pipeline 36
3.3.2 Quantization of inference model 38
3.3.3 Embedded inference 39
3.4 Parameter selection 39
3.4.1 Validation metrics 39
3.4.2 Feature extraction emulation 41
3.4.3 Label generation 42
3.4.4 Methodology 43
3.4.5 Results 45

4 Characterization **49**
4.1 Power consumption 49
4.1.1 Setup 49
4.1.2 Results 49
4.1.3 Discussion 49
4.2 Full system simulation 51
4.2.1 Setup 51
4.2.2 Results 52
4.2.3 Discussion 53
4.3 Full system validation 53
4.3.1 Setup 53
4.3.2 Results 54
4.3.3 Discussion 55

Conclusion **57**

A Data of the algorithm optimization **65**

B Code of the pipeline for generation of the features **69**

C Emulation of the pipeline for generation of the features **73**

List of Acronyms

ADC Analog to Digital Converter

AFE Analog Front-End

AI Artificial Intelligence

AP Average Precision

AUC Area Under the Curve

CMSIS Common Microcontroller Software Interface Standard

CNN Convolutional Neural Network

CPU Central Processing Unit

DMA Direct Memory Access

FFT Fast Fourier Transform

FN False Negative

FP False Positive

FPR False Positive Rate

FPU Floating-Point Unit

GPIO General Purpose Input/Output

HAL Hardware Abstraction Layer

IDE Integrated Development Environment

LDO Low DropOut regulator

MCU Micro-Controller Unit

MSFB Mel-Scaled Filterbank

PSD Power Spectral Density

RAM Random Access Memory

RF Radio Frequency

ROC Receiver Operating Characteristic

RTC Real-Time Clock

SD Secure Digital

SIMD Single Instruction Multiple Data

SPI Simple Peripheral Interface

TN True Negative

TP True Positive

TPR True Positive Rate

UART Universal Asynchronous Receiver-Transmitter

List of Figures

1 Separation of the acoustic signal into windows and resulting spectrogram. (Hann window, overlap of 25%, window size of 512 samples at 300 kHz) . . . 2
2 Two representations of the Mel filters. (10 filters, from 0 kHz to 20 kHz, window size of 512 samples at 40 kHz of sample frequency) 3
3 Different types of bat calls. [19] 4

1.1 High-level block schematic of the platform. 7
1.2 Photos of the platform with and without extension for bat monitoring. . . . 8
1.3 Preliminary ultrasonic free field response of the microphone normalized to 1kHz. [42] 9
1.4 Schematic of the AFE. 10
1.5 Simulated transfer function of the filtering and amplifying chain. [35] 10
1.6 microSD card pinout. [43] 11
1.7 Software stack of the microSD card. [35] 11
1.8 SPI link schematic. 11

2.1 Two examples of audio signal transmitted via UART. 15
2.2 Throughput of SD write operation. 18
2.3 Spectrogram of the noise recorded by the ADC when no other sound is present. 18
2.4 Supply tree of the main board. 19
2.5 Supply tree of the extension board. 19
2.6 Setup for external supply of the microSD card. 20
2.7 Measurement setup of the noise on the microphone signal. Measuring V_{dd}. . 20
2.8 PSD of V_{dd} without external supply. 21
2.9 PSDs of the different nodes of the amplification chain without external supply. 21
2.10 Difference between the PSDs of V_{out} and V_{min}. 21
2.11 Comparison of the PSDs of the noise recorded on the ADC with and without external supply. 21
2.12 Setup at the Louvain-la-Neuve site. 24
2.13 Setup at the Aubange site. 24
2.14 Histogram of call probabilities when no bat is recorded. 26
2.15 Histogram of call probabilities when bats are recorded. 26
2.16 Histogram of call durations when bats are recorded. 26
2.17 Comparison of bat detection probability based on duration. 27
2.18 Histogram of call durations of calls kept in the dataset. 27

2.19 Histogram of distances between two calls kept in the dataset. 27
2.20 Histogram of detected species with bat recorded and threshold of 0.6. . . . 27
2.21 Distribution of the lower and higher frequency bounds of calls kept in the dataset. 28

3.1 Full spectrogram generation pipeline. 33
3.2 Hann and Hamming windows for a window of size 512. 34
3.3 Frequency response of Hann and Hamming windows. 34
3.4 Time measurements of data formatting over unroll factor. Buffer of size 512. 36
3.5 Time measurements of CMSIS `arm_rfft_q15` function over buffer size. . . . 36
3.6 8 segments piecewise linear approximation of the logarithm function. 37
3.7 Time measurements of third stage over loop-unrolling factor. Buffer of size 512. 38
3.8 Time measurements for mel-scaled filtering over loop-unrolling factor. Buffer of size 512 and 64 MSFBs. 38
3.9 Example of precision and recall curves. 40
3.10 Example of TPR and FPR curves. 41
3.11 The two cases to test for call in a window. Upper window is the point from which we begin to consider that it contains a call, lower window is where we stop. 43
3.12 Distribution of models over metrics and normalized execution time with optimality boundary. 45
3.13 Distribution of time taken by each step of computation. 47
3.14 Execution time of the different layers of the chosen model. 47
3.15 Performance curves of the quantized chosen model. 48

4.1 Measurement setup for the power consumption. 50
4.2 Precision-recall curves for different values of $S_{\text{spec,hop}}$. For clarity, average between higher and lower bound of the precision is used. 52
4.3 AP against $S_{\text{spec,hop}}$. 52
4.4 Timestamp of calls predicted in real time and calls detected with 0.6 probability in recording. 54

A.1 AUC against AP on measured models. 66

List of Tables

1.1 Current consumption according to data sheets. 13

2.1 Functionality of transmission via UART with different baud-rates. 16

2.2 Comparison of existing models with reported performance, availability and output information. 23

2.3 Summary of recording sessions. 25

3.1 Comparison of computation times for logarithm computation for a buffer of size 512. 37

3.2 Diagnostic testing diagram. [64] 39

3.3 Error done on computation of the features at different points of the pipeline. 41

3.4 Variables used in the search for model optimum and their meaning. 43

3.5 List of symbols for computation time estimation with their meaning. 44

3.6 Chosen model parameters and results. 46

3.7 Performance of the model in different situations. 48

4.1 Result of power consumption measurement for different modes of the system. 50

A.1 Full data from the model optimization. Chosen model in green. 65

A.2 Tested models execution time. 67

Introduction

Context

Biodiversity has long been of concern to environmental authorities; its importance for the short- and long-term wellbeing of humanity, whether it is through alimentation, medicine, economics or any other factor arising from the symbiosis between humans and the rest of the natural world, has been recognized long ago. However, the rise in human expansion has led to an increase in habitat destruction [2]. To date, the main cause of biodiversity loss has been the human exploitation of ecosystems. Climate change is likely to exacerbate the situation as it will lead to perturbations in habitats with unpredictable consequences. These new challenges will require new measures to mitigate biodiversity loss, as currently protected areas may become inhabitable for endangered species [3].

According to IBPES [4], the global extinction rate of species is now tens to hundreds of times higher than the average rate of the last 10 million years, and this increasing trend shows no sign of stopping. The emergence of human activity as an influential actor in the system formed by the Earth and all its components has had a destabilizing effect. Throughout modern civilization, the Earth has remained in a stable "Holocene-like" interglacial state, ideal for human development. However, humanity's overstepping of several planetary boundaries, the limits within which the planet is able to remain in a stable state, puts us at risk of facing an unprecedented change in planetary state [5].

Bat species have not been spared from global biodiversity loss: of the 1326 bat species on the IUCN Red List, 324 are known to be in decline, we lack data to determine the status of 704 other species [6]. Moreover, bats have shown promise as bio-indicators because of characteristics such as their global distribution or their effectiveness as surrogate taxon [7], which is a group of organisms whose species richness can represent that of the rest of the biosphere in the same area [8].

A significant part of the scientific community has therefore pushed for the monitoring of bat populations in order to gain useful information about ecosystems. It has been carried out in three main ways: roost surveys, mist nests and acoustic surveys. While mist nests and roost surveys provide more reliable data, they are more time-consuming and stressful for bats than acoustic surveys, which is why this method is the main method used for censuses of bat faunas [9]. However, acoustic surveys of bats have remained limited due to a number of factors, including their cost, leading to difficulties in the systematic monitoring of bats [7].

State of the art

Developments in bat acoustics began in the late 1970s. At that time, the aim was to bring bat calls into the audible spectrum so that they could be recorded using an inexpensive

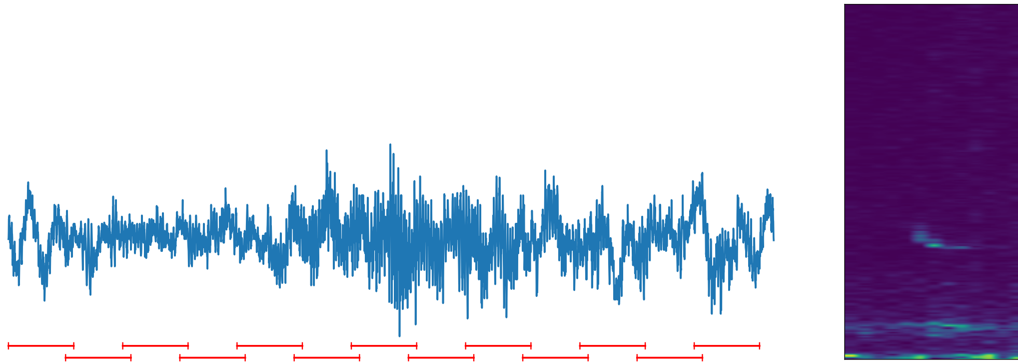


Figure 1: Separation of the acoustic signal into windows and resulting spectrogram. (Hann window, overlap of 25%, window size of 512 samples at 300 kHz)

device such as a cassette recorder. Two types of recording equipment were then produced: the heterodyne system and the time expansion system. The heterodyne system works by transposing a narrow band of high frequencies into the audible spectrum. This means that bats will not be heard if their calls are out of the band to which the device is calibrated. The time expansion system records the sound at a high sampling frequency and plays it back at a lower sampling frequency. This means that the full spectrum is audible. However, unlike the heterodyne system, the time expansion system cannot record continuously because it plays back the sound slower than it records it. The calls recorded will therefore depend on when the user starts the device and how long it can record before playing back [10].

With the advent of digital systems, other methods could be used to record audio into memory for later processing. However, as digital memory was expensive, only a compressed representation of the signal could be recorded. One such representation is the zero-crossing method, which tracks the number of times the audio signal crosses an arbitrary threshold level (called zero). This method returns the value of the primary frequency in the signal in a given window, resulting in a small file size [11]. However, as costs have come down, solutions that store the full spectrum or raw audio signal have appeared on the market [12].

Automated detection and species classification of bat calls soon became an active area of research to replace the tedious and repetitive manual process. The first methods were based on the extraction of call features from spectrograms, which were then processed by machine learning algorithms [13, 14]. Spectrograms are a way of representing the frequency components of a signal and their evolution over time. They are obtained by dividing the audio signal into windows and applying a discrete Fourier transform to the samples in each window. The windows can overlap, which can help to better represent the frequency content of the signal. In most cases, each sample window is multiplied by a shaping window (e.g. Hann, Hamming) to reduce spectral leakage. Figure 1 shows the separation of an audio clip into windows and the corresponding spectrogram.

Further transformations can then be applied to the spectrograms. A common practice is to transform the signal into Mel-Scaled Filterbanks (MSFBs). This is done via a series of triangular filters (usually called Mel filters, shown in Figure 2) spaced non-linearly along the mel-scale [15]. The mel-scale is defined by the formula

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{1}$$

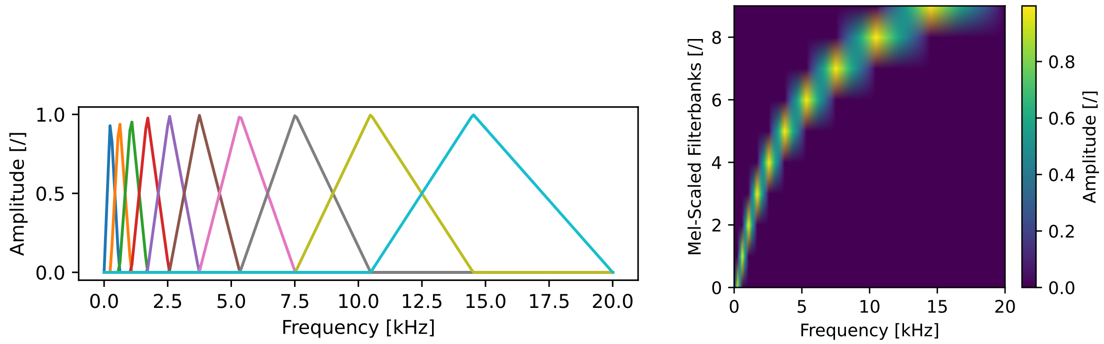


Figure 2: Two representations of the Mel filters. (10 filters, from 0 kHz to 20 kHz, window size of 512 samples at 40 kHz of sample frequency)

to follow human perception, which gives more weight to the variation of low frequency sounds compared to high frequency sounds [16]. When applied to a spectrogram, this series of triangular filters can be implemented as a matrix multiplication.

MSFBs can then go through a discrete cosine transform to obtain the mel frequency cepstral coefficients. This method has been used in speech detection because of its ability to decorrelate features [15], but it has been less successful in the case of bat calls compared to MSFBs [17].

Bat calls can be divided into several structures based on their frequency contents. There are three basic call structures that serve as building blocks for more complicated ones. Constant Frequency (CF) calls have frequencies that do not change over the duration of the call. Frequency Modulated (FM) calls contain frequencies spanning a range over 5 kHz, while calls that vary over a range under 5 kHz are called Quasi-Constant Frequency (QCF) calls. Figure 3 contains a spectrogram illustrating different types of bat calls. Figure 1 contains an FM-QCF call from a common pipistrelle.

These different call structures are used by different bat species depending on their environment and anatomy. For example, FM-CF-FM calls are typical of horseshoe bats in Europe. With an audition that is very sensitive to a narrow bandwidth, those bats are specialized in this call structure, which allows them to use the Doppler effect to detect insects based on the movement of their wings [18]. FM calls are expensive but give the bat a very accurate picture of its environment. Those calls are therefore more likely to be used by forest-dwelling species, since the high density of obstacles and preys forces the bats to make quick decisions based on their environment. QCF calls have a longer range but are less accurate and are therefore used more by species that live in large, clear airspaces. FM-QCF calls offer a compromise between FM and QCF calls and are used, for example, by species that are used to tree lines on open land, such as the common pipistrelle. These structures are therefore an important key for species identification, even though other factors used by experts include the rhythm, the regularity, the frequency of maximum intensity of the call, the terminal frequency of the call, the width of the frequency band and the duration of the call [18].

Over time, developments in machine learning in other fields led to the conclusion that directly learning from spectrograms could produce better results than those obtained by an expert decision algorithm based on an engineered call feature extraction [20]. Since then, several machine learning algorithms have been tried for bat call detection [21], species classification [17, 22–24], or both at the same time [25–30].

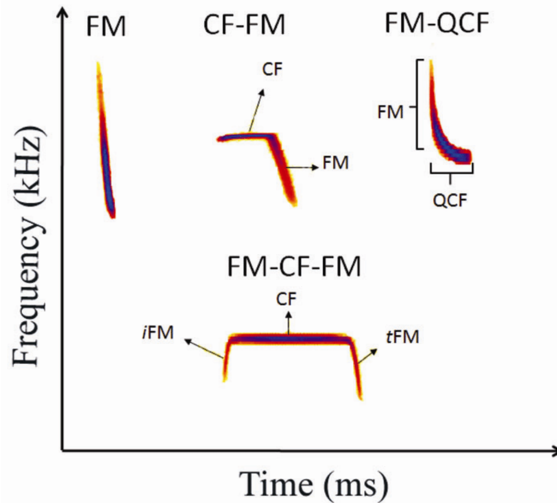


Figure 3: Different types of bat calls. [19]

One of the innovations that has led to these improvements is the introduction of supervised deep Convolutional Neural Networks (CNNs) [21], a type of neural network that contains at least one convolutional layer [31]. A convolutional layer works by convolving a kernel with the input elements. This is usually done with multiple kernels, resulting in as many outputs. The weights of the kernels are then trained to maximize the performance of the model. As convolution is a linear operation, a non-linear layer is usually placed after a convolutional layer to allow the model to learn non-linear functions. A pooling layer can also be placed after the non-linear layer to reduce the dimensionality of the data. There are several types of pooling layers; a common one is the max pooling layer, which works by dividing the input into blocks of equal sizes and only keeping the maximum of the elements in each block. Another common practice is to end the model with dense layers, which work like matrix multiplications in which the elements of the matrix are trained weights.

More recently, recurrent neural networks, another type of neural networks in which the execution of the model takes into account past executions, have shown good performance in species classification [28, 30] but have not shown significant improvements in call detection compared to the case in which the recurrent part of the system is removed [30].

Contribution

As Artificial Intelligence (AI) has become an important tool for problem-solving, the need for efficient computing solutions has arisen. Cloud-based solutions have been very successful, but have several drawbacks. The most important one in our case is their narrow bandwidth, which creates a limit to the throughput of input data that the systems can accept [32]. TinyML [33] is an alternative paradigm in AI, which does the inference directly at the edge level instead of delegating the work to servers far away from the sensor node. It aims at low-cost systems with low power consumption and the ability to work offline.

TinyML offers several solutions for embedded AI. On the hardware side, these solutions include IoT-ready microcontrollers, hardware accelerators through FPGAs and ASICs (e.g. Google's Edge Tensor Processing [34]). On the software side, techniques such as model quantization and compression can help to fit models within the constraints of a given hardware solution [33].

The task of bat detection and classification lends itself well to tinyML. The fact that the prediction is based on high frequency sounds leads to an equally high sampling frequency (e.g. 300 kHz). This high data throughput would require prohibitive power for wireless transmission. Storing the data for future treatment also increases power consumption (although usually less than data transmission) and can lead to a reduction in unattended operating time if the data fills the memory faster than the battery is depleted. On the other hand, applying inference on the device allows dramatically reducing the data throughput, which makes transmission or storage of the result possible while still keeping a low-power system.

In this work, a machine learning model with its signal processing pipeline is implemented to run on the sensor platform developed by Nicolas Brusselmans during his master's thesis [35], which includes a low-end MCU. This platform was initially designed for fire detection, but an extension for ultrasonic monitoring was designed to demonstrate the device's extensibility. The performance of the completed device is then analyzed and solutions to problems in the base system are presented.

The algorithm is inspired by state-of-the-art algorithms, while respecting the constraints imposed by the MCU used. The design of this algorithm is based on an optimization of its different parts in order to see how the physical limitations of the device limit the performance of the algorithm. This result makes it possible to link hardware choice and performance metrics by extending it to platforms with different computing power.

The model is trained on data collected directly from the device to account for non-idealities of the microphone. This data collection campaign was mainly carried out at a site near the lake of Louvain-la-Neuve. A pre-existing, publicly available model is used to annotate the data. The results of the data collection and the annotation strategy are discussed with their limitations.

A characterization of the resulting system is performed to determine the operational capabilities of the device in different use cases. Bottlenecks are also analyzed, leading to a discussion on the choice of additional hardware to improve performance.

Structure

This thesis is divided into 5 chapters.

Chapter 1 describes the hardware of the system on which the algorithm will be implemented.

Chapter 2 focuses on the generation of the dataset. It starts by explaining how the system had to be modified to allow data collection, then explains how the data is annotated, and finally presents the result of the data collection in the field.

Chapter 3 describes the choices made for the algorithm design and the rationale behind them. It also explains the implementation of the algorithm on the platform. Finally, the algorithm resulting from the optimization of the different hyperparameters is described.

Finally, chapter 4 characterizes the finished system in terms of its power consumption and bat detection capabilities, and discusses the results.

All the codes and data needed to reproduce results are available at the following URL:

`https://forge.uclouvain.be/forestme/forestme_bat`

Chapter 1

System overview

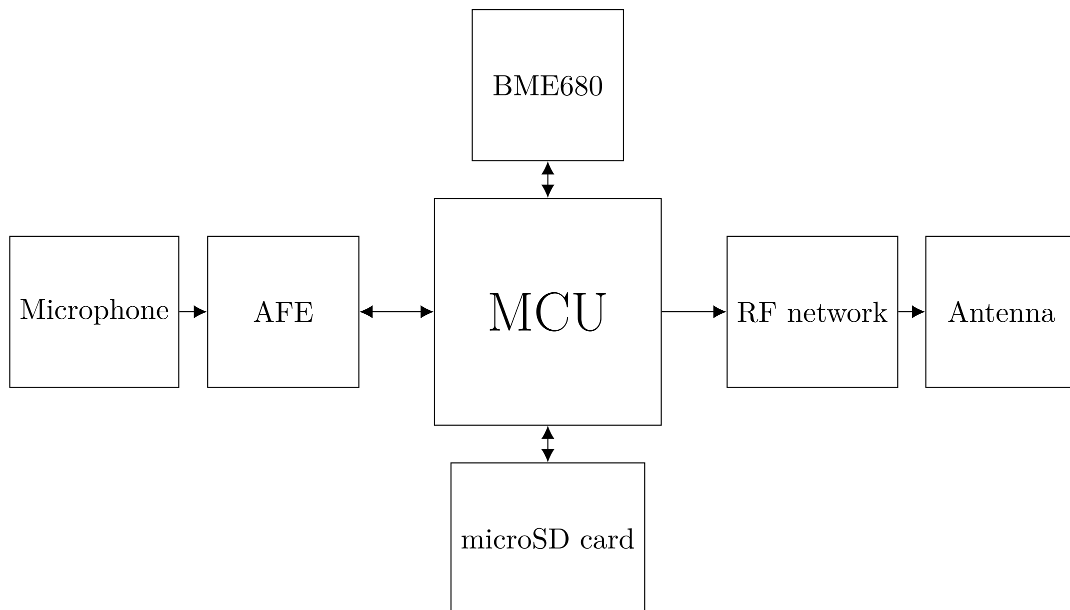
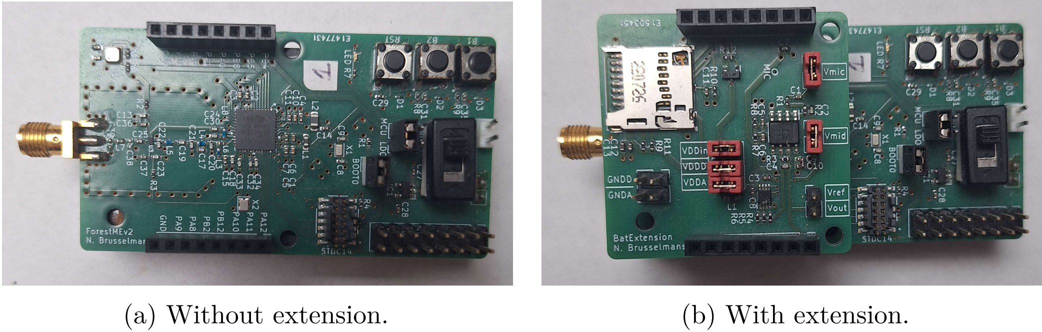


Figure 1.1: High-level block schematic of the platform.

The platform consists of two printed circuit boards. The first is the main board, which contains all the components associated with the MCU, the power supply, the radio and the gas sensor. Below this is a battery holder that can hold three AAA batteries. Taking this into account, the dimensions of the main board are 7.5 cm × 4 cm × 2.4 cm. A switch controls the power source. The two possibilities are to use the battery holder or another supply, that can be connected via two pins. The price of the printed circuit board with all the components is about 75 €. This cost is dominated by the cost of the printed circuit board itself, which is 25 €.

The second board is an extension of the main board, inserted on top via the connectors. Its dimensions are 4 cm × 4 cm, and it contains all the parts related to audio acquisition and memory storage on the microSD card. Figure 1.2 shows the platform with and without the extension, and the complete block diagram of the system is shown in Figure 1.1.



(a) Without extension.

(b) With extension.

Figure 1.2: Photos of the platform with and without extension for bat monitoring.

1.1 Main board

The platform is equipped with a STM32WLE5CCU, which contains a Cortex-M4, 256 kB of flash memory and 96 kB of Random Access Memory (RAM) [36]. The Cortex-M4 can run at up to 48 MHz and supports Single Instruction Multiple Data (SIMD) operations. SIMD operations allow speeding computations up by applying the same operation to multiple element at the same time. In the case of the Cortex-M4, the 32-bit wide arithmetic logic unit is used to treat multiple 16-bit or 8-bit element at the same time. This way, it is for example possible to do two 16-bit addition or four 8-bit addition in a single cycle [37]. The Common Microcontroller Software Interface Standard (CMSIS) is made by ARM to simplify software re-use and knowledge transfer from project to project [38]. One of the element of CMSIS is the libraries which allow for efficient implementation of commonly used functions. They use all capabilities of the CPU, using techniques like SIMD operations or loop unrolling extensively.

The MCU contains a Direct Memory Access (DMA) which allows various components (e.g. the ADC, the UART) to access memory without CPU intervention. A Real-Time Clock (RTC) keeps track of the current time, allowing strategies based on the time of day. This can be particularly useful in our application, as bats come out mainly at night.

The platform includes a female STDC14 interface. This allows the user to connect a debugger to the MCU. When used with the STLINK-V3MINIE, this creates a USB connection between a host computer and the MCU, which allows reading from the UART. The flash memory can be written to using the debugger and the software offered by STMicroelectronics. For this project, STM32CubeIDE will be used. It supports automatic compilation of the project, flashing the memory with validation of the result, debugging and package installation.

The platform is powered by 3 AAA batteries through a Low DropOut regulator (LDO) that stabilizes the voltage at $V_{dd} = 3.3\,\text{V}$. The LDO is a TPS7A03 from Texas Instruments [39]. It is designed to draw only 200 nA of quiescent current. This low ground current used by the regulator allows for high current efficiency when the system is in use.

The STM32WL5 was chosen for its support of LoRa modulations, allowing it to interface with the LoRa radio without additional hardware. The output power and spreading factor can be selected to control power consumption, transmission time and communication range. The ability to transmit data has been shown to be highly dependent on the location of the node. With a spreading factor of 7 and a maximum output power of 14 dBm, the maximum communication distance was between 400 m and 600 m depending on the location. In this configuration, a transmission of 16 bytes takes 52 ms and consumes 3.91 mJ at 3.3 V[35].

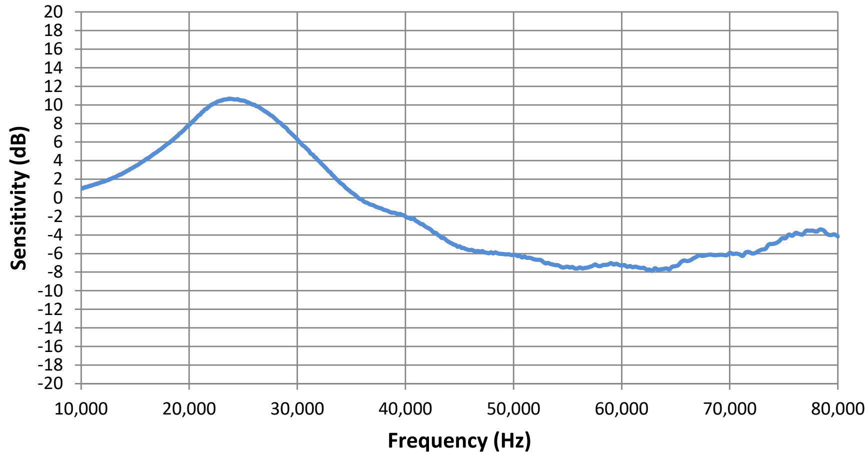


Figure 1.3: Preliminary ultrasonic free field response of the microphone normalized to 1kHz. [42]

A BGS12SN6 Radio Frequency (RF) switch from Infineon [40] in the matching network allows the antenna path to be connected to either the output or input path of the MCU.

As the original purpose of the platform was to monitor forest fires, it includes a gas sensor, the BME680. It can measure temperature, relative humidity, pressure and the concentration of volatile organic compounds. The concentration of volatile organic compounds is particularly useful in the case of fire detection, as it increases greatly when smoke comes into contact with the sensor, but the sensor needs to heat up to measure it, leading to increased power consumption (around 44.47 mW) [35]. We will not be using it in this project as we will be concentrating on the bat detection task. When not in use, the sensor should consume less than 1 µW [41].

1.2 Ultrasonic AFE and microphone

The microphone is a SPU0410LR5H-QB from Knowles [42], which contains a MEMS-based acoustic sensor, a low noise input buffer and an output amplifier [42]. It is not specifically designed for ultrasound recording but, as shown in Figure 1.3, it is capable of capturing frequencies well above the audible spectrum with a transfer function that remains relatively stable. It consumes at most 160 µA [42], which means a maximum consumption of 528 µW as we supply it at 3.3 V.

The Analog Front-End (AFE) is based on a two-stage amplification and filtering chain. A switch gives a choice between three resistor values, changing the value of R in Figure 1.4. The selected resistor can be 4.7 kΩ, 15 kΩ or 47 kΩ, giving a gain of 33.1 dB, 43 dB or 53.1 dB respectively, as shown in the transfer function in Figure 1.5. The gain can thus be selected in the software via two GPIOs.

The filter used is a bandpass filter. The lower cut-off frequency is 340 Hz and the upper cut-off frequency is 133 kHz. The bandwidth varies slightly with gain, but this variation should remain negligible. The transfer function of the AFE is shown in Figure 1.5.

V_{out} is sampled by the ADC of the MCU. It is able to record 12-bit samples at up to 2.5 MS/s. It accepts values between 0 V and V_{dd}. The AFE is designed such that V_{out} oscillates around $V_{\text{ref}} = V_{dd}/2$.

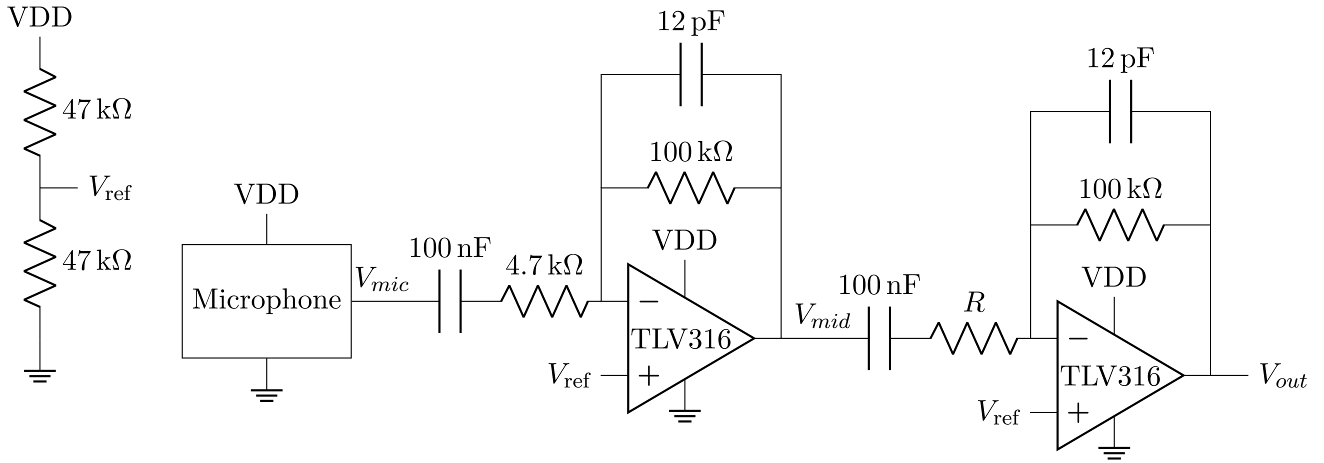


Figure 1.4: Schematic of the AFE.

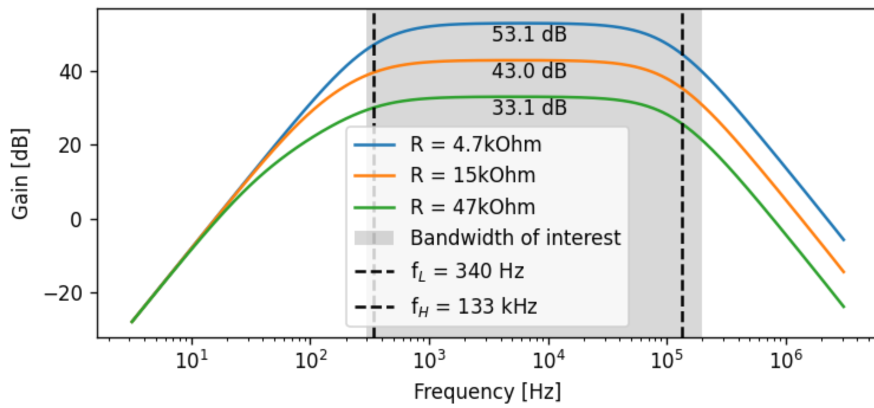


Figure 1.5: Simulated transfer function of the filtering and amplifying chain. [35]

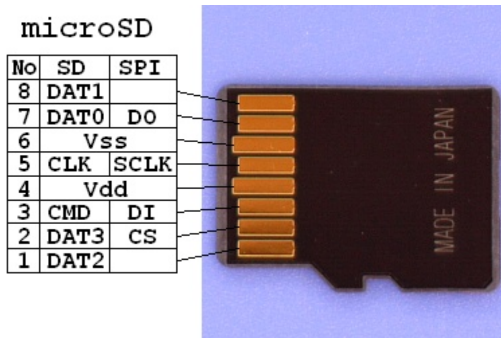


Figure 1.6: microSD card pinout. [43]

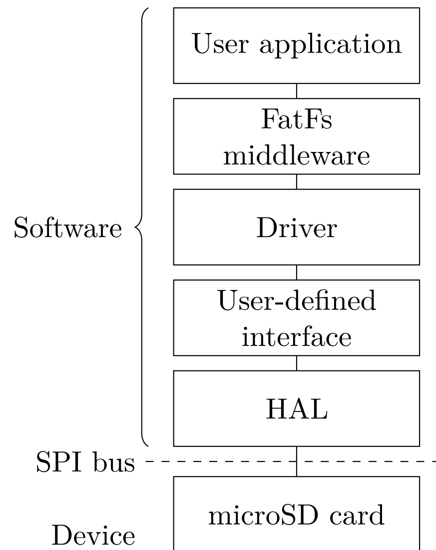


Figure 1.7: Software stack of the microSD card. [35]

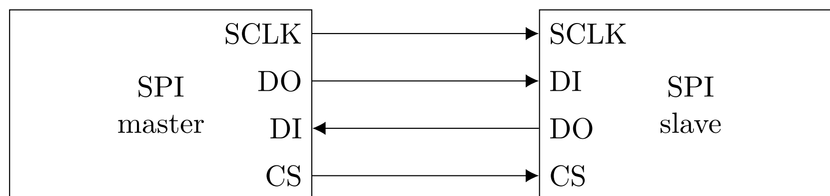


Figure 1.8: SPI link schematic.

1.3 Memory storage

The system includes a connector that allows a microSD card to be plugged in. To interface with it, SPI is used as a simpler alternative to the native operating mode. Figure 1.6 shows how the native pinout is used for the usage through SPI.

An SPI connection between a master and a single slave is shown in Figure 1.8. SCLK is the forwarded clock used for synchronization, which is generated by the master. CS is an active low signal used to select the slave for communication. For the microSD card, the SPI mode 0 should be used. This means that data is sampled on the rising edge of the clock and that data transfer starts as soon as CS goes low.

The software stack to interface with the microSD card is shown in Figure 1.7. The FatFs middleware contains the logic required to interface with the file system. FatFs is available as a middleware in STM32CubeIDE, allowing easy integration into the project. A driver [44] is necessary for the middleware to work. It provides a standard hardware-agnostic interface, and translates it to the functions specific to our hardware. The user-defined interface will determine the exact implementation details of the connection, such as the pins used by the SPI connection. The Hardware Abstraction Layer (HAL) is made up of the functions provided by STMicroelectronics to interface with the hardware. By calling them, the system is able to read or write to the SPI link.

One limitation from the software stack is that the FatFs middleware for the Integrated Development Environment (IDE) only supports Fat32 and a single partition. This limits our total storage to 32 GB. This will likely be the bottleneck in terms of recording time, as

it only allows just under 16 hours of recording at 600 kS/s with samples encoded on 16 bit.

The data throughput achievable on the SPI link depends on the clock and the microSD card used. There are several classes of microSD cards; in this project, a 32 GB SanDisk Ultra is used. It is a class 1 microSD card, which means that it is certified to work at a minimum of 1 MB/s. When the system was designed, a data rate limitation was observed. Indeed, the device could not write at more than 20 kB/s. This issue is addressed in Section 2.2.1.

The power consumption of microSD cards in SPI mode is not well documented, as many datasheets don't give a value or give a higher limit of 100 mA. Nevertheless, measurements show an average power consumption between 20 mA and 24 mA[45] with peaks well below 100 mA.

1.4 Power budget

The power consumption of the different components of the system based on the values from the datasheets of the different components is given in Table 1.1. The gas sensor is assumed in sleep mode as it is not used in this project. Based on these values, we can calculate different theoretical power consumptions for the whole device.

The first case that we will consider is the one in which the system is set up to write to the microSD card. In this configuration, the MCU is in run mode and all the components are active, which leads to a typical current consumption of 30.57 mA and a maximum current consumption of 108.89 mA. The main source of consumption is the microSD card with 78.5 % of the typical current consumption, followed by the MCU with 18.16 %. The consumption should only reach its peak during short spikes [45]. So the average power consumption should be 137.57 mW when the system is supplied at 4.5 V. We can compare this value with the 195.36 mW measured during the initial characterization of the system [35]. We can see that the consumption is slightly higher, but this could be explained by the fact that the microSD cards used in our baseline are of a different brand to those used in the original characterization.

Another case we can consider is one in which the MCU is set to handle the data coming from the microphone, but not to store it on the microSD card. The typical and maximum power consumption is then of 6.57 mA and 8.89 mA respectively. The MCU is now the dominant source of power, consuming 84.47 % of the total power. In this case, the whole system should typically consume around 29.57 mW and up to 40 mW when supplied at 4.5 V.

The last case we can look at is the one in which the system is idle. In this case, the power to the expansion board is cut off, leaving only the main board components powered. There are two possibilities, depending on whether the MCU is in Stop 1 mode or in standby. In Stop 1 mode, all clocks in the VCORE domain (digital peripherals, SRAM1 and SRAM2), the phase-locked loop, the multispeed internal RC oscillator and both the internal and external high-speed RC oscillators are disabled. The system is powered by a low-power regulator instead of the main regulator, at the cost of a longer wake-up time. Standby mode has all the same savings as Stop 1 mode, except that the entire VCORE domain is now switched off. This means that when the system wakes up, it behaves as if it had been reset, although it is still possible to keep the data in SRAM2 to have a persistent state between idle phases.

When the system is in Stop 1 mode, the typical current consumption is 104.55 µA and the maximum current consumption of 202.25 µA. When the system is in Standby mode,

the typical current consumption is 100.8 µA and the maximum current consumption is 181.7 µA. In both cases, the consumption of the RF switch dominates with 95.65 % of the typical consumption with the MCU in Stop 1 mode and 99.2 % of the typical current consumption with the MCU in standby. If the system is supplied at 4.5 V, this gives a typical power consumption of 470.48 µW in Stop 1 mode and 453.6 µW in standby.

If a switch had been added at design time to cut off the power supply to the RF switch when it is not being used for the radio, typical idle current consumption would have been less than 1 µA.

Component	Operating conditon	Current [mA]		
		Typ.	Max.	
MCU	Standby[1]	0.000445		[36]
MCU	Stop 1 mode[2]	0.0042	0.021	[36]
MCU	Run mode[3]	5.55	7.40	[36]
LDO		0.0002	0.00025	[39]
Gas sensor	Sleep	0.00015	0.001	[41]
RF switch		0.100	0.180	[40]
microSD card	Write	24	100	[45]
Microphone	V_{dd} =1.8 V	0.120	0.160	[42]
Operational amplifiers	Two amplifiers	0.800	1.15	[46]

Table 1.1: Current consumption according to data sheets.

[1] 25 °C, V_{dd} =3 V, SRAM2 and backup registers retained and RTC enabled clocked by LSE quartz
[2] 25 °C, V_{dd} =3 V, RTC enabled, clocked by LSI
[3] 25 °C, 48 MHz

Chapter 2

Dataset creation

2.1 Data collection through the UART

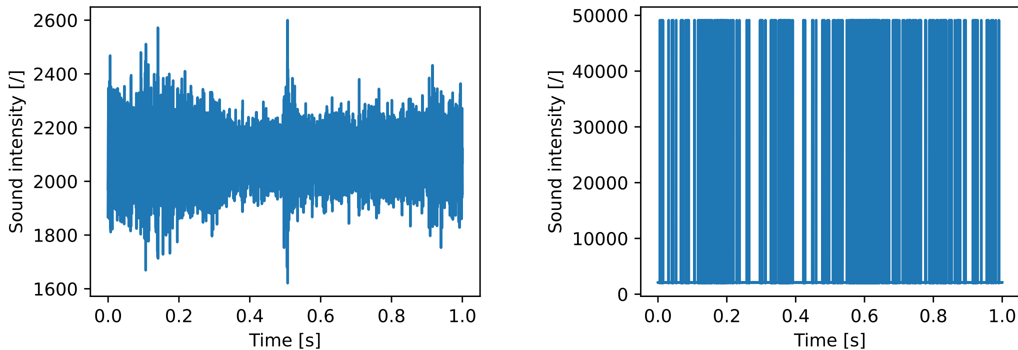
One way to collect data from the device is to use the debugger to get data from the UART. In this setup, a computer must be constantly connected to the platform to receive the data. The connection is done through the STLINK-V3MINIE debugger. This debugger creates a virtual COM port interface for the computer to read.

2.1.1 Setup

A 15-cs1050nb HP Pavilion laptop is used as the host computer. It runs `picocom` [47] to connect to the serial port, and the output of the program is written to a file using `tee`. `screen` [48] and `minicom` [49] could not be used, most likely because the required baud rate (4.8 Mbit/s) is greater than 4 Mbit/s, which is the maximum standard baud rate listed in `termbits.h` on the host computer. Therefore, `picocom` seems to be the only one of the three program which supports non-standard baud rates. The transfer is done at different baud rates to test functionality.

2.1.2 Result

Two behaviours can be observed in the resulting file, depending on the baud rate. Both are shown in the Figure 2.1. Either the sound signal is recorded correctly, or the signal is



(a) Baud-rate at 7.3728 Mbit/s.

(b) Baud-rate at 7 Mbit/s.

Figure 2.1: Two examples of audio signal transmitted via UART.

Throughput [MHz/s]	Correct transfer?
6.8	No
6.9	Yes
7	No
7.3728	Yes
7.5	No
8	No
12	No

Table 2.1: Functionality of transmission via UART with different baud-rates.

severely distorted, and the audio signal is only correct for a few moments. When converted to a WAV file and listened to, the original audio is barely audible behind the noise. Table 2.1 shows which of the two behaviours was observed in the different tested throughputs.

2.1.3 Discussion

The observed distortion can be explained by data loss. Since the samples are 16 bytes long, if a byte is skipped, the data is no longer aligned. For a given sample, its low order byte is then used as the high order byte and the high order byte of the next sample as the low order byte. This explains the high amplitude noise observed and the alternation between correct data transmission and high noise, the switch occurring when an odd number of bytes are skipped.

The exact cause of the data loss is not known. The STLINK-V3MINIE should support communication up to 16 Mbit/s [50]. The fact that it works for 7.3728 Mbit/s but not 7 Mbit/s indicates that the losses are not just due to the connection, which would place an upper limit on the achievable baud rate. As the frequency of the input clock of the UART is 48 MHz, the IDE indicates that any baud rate between 11.719 kbit/s and 16 Mbit/s is supported. Even if we consider that synchronization between clocks could be important, both 8 Mbit/s and 12 Mbit/s should have no problem being used by the platform as they are integer fractions of the frequency of the input clock. However, this is not observed. On the other hand, 6.9 Mbit/s and 7.3728 Mbit/s both work when they are not synchronized to the input clock.

The final possible source of loss is at the host computer level. As we are outside the standard baud rates, we cannot guarantee data integrity. It is up to the implementation of the driver whether a baud rate is supported or not. The exact specifications of the driver running on the host computer or those of the clock used by that driver could not be determined. However, this is the only part of the connection in which we are outside the supported usage. It is therefore likely to be a point of failure.

The value 7.3728 Mbit/s was chosen as 64 times 115.2 kbit/s, which is one of the most commonly used baud rates. The fact that this baud rate is one of the few that works may indicate a clock division problem with other baud rates.

The working system can be used to record data using one of the baud rates identified as working. However, the system obtained is not very practical. Its portability is very low as a laptop has to be permanently connected to the platform. This laptop has to be plugged in an outlet for long recording sessions, which further limits the recording possibilities. All

this led to the decision to privilege the microSD card as a means of data collection.

2.2 Data collection through the microSD card

2.2.1 microSD card write throughput increase

During the initial characterization of the device, it was found that the write operation to the microSD had a data rate of 20 kB/s. [35]. This is indeed what was observed when opening the project as it was. Activating the compiler optimizations to -Ofast brought this data rate up to about 110 kB/s. This is still not enough for audio recording.

```
//btx is the size of the buffer to send
for(UINT i=0; i<btx; i++) {
    HAL_SPI_Transmit(&SD_SPI_HANDLE, buff+i, 1, HAL_MAX_DELAY);
}
```

Code 2.1: Original write driver.

The driver used to interact with the microSD card is based on an open source repository [44]. The code used to write to the microSD card is given in Code 2.1. It repeatedly calls the `HAL_SPI_Transmit` function to send one byte at a time. However, if one looks at the definition of the function, they will see that the fourth argument which is set to 1 here is used to specify the size of the buffer to send. This means that Code 2.2 is actually exactly equivalent to Code 2.1. However, Code 2.2 should have better performance because it avoids repeating the initialization and de-initialisation of the `HAL_SPI_Transmit` function and the cost of the *for loop*.

```
//btx is the size of the buffer to send
HAL_SPI_Transmit(&SD_SPI_HANDLE, buff, btx, HAL_MAX_DELAY);
```

Code 2.2: Optimized write driver.

```
//btx is the size of the buffer to send
for(UINT i=0; i<btx; i+=TRANSFER_SIZE) {
    HAL_SPI_Transmit(&SD_SPI_HANDLE, buff+i, TRANSFER_SIZE, HAL_MAX_DELAY);
}
```

Code 2.3: Parametrized driver.

For analysis purposes, Code 2.3 is another equivalent version of the code with a parameter `TRANSFER_SIZE` used to control the transfer size, giving intermediate situations between the initial and optimized situations. Figure 2.2 shows throughput measurements of three different parts of the code: the content of the *for loop* which is a single SPI transfer of size `TRANSFER_SIZE`, the *for loop* which is a transfer of 512 B, and a full transfer of 25 kB. As can be seen, at high transfer sizes, both the transfer of 512 B and the single `HAL_SPI_Transmit` call go faster than should be physically possible given the clock on which the transfer is being done. This is actually because the only job of the CPU is to put the data into a buffer which will be sent via SPI. The CPU can then return to other tasks. The full 25 kB transfer has a throughput under the SPI clock, which is logical since most of the bytes must have been transmitted by the time we finish sending.

The optimized code has a full transfer throughput of 1.28 MB/s, which is greater than the 600 kB/s required for continuous sound recording. Since the clock allows for a maximum of around 1.43 MB/s, we now have a utilization of 85 % of the SPI link.

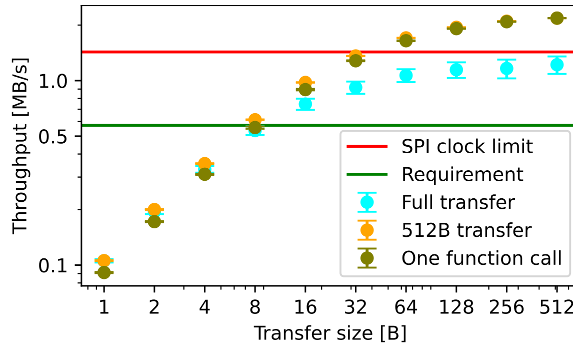


Figure 2.2: Throughput of SD write operation.

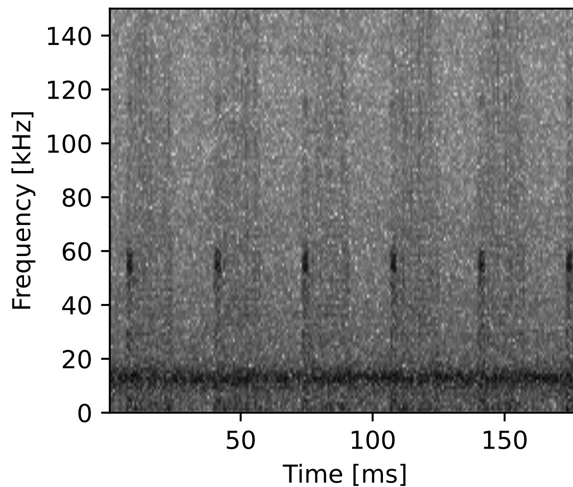


Figure 2.3: Spectrogram of the noise recorded by the ADC when no other sound is present.

2.2.2 Continuous recording setup

Given that the throughput is now sufficient for continuous recording, it can be implemented in software. To do so, the ADC is set to record continuously to a buffer via the DMA. We then activate interrupts when one half of the buffer is filled. When the interrupt is sent, the callback will write the half of the buffer that has finished sampling to the microSD card.

This double buffering allows for continuous sampling. Indeed, the ADC never stops writing to memory and the CPU is able to process data in parallel with the acquisition.

2.3 Supply noise analysis and attenuation

During the recording of the data from the microphone, a high frequency noise was recorded, even in the absence of any noise source. The noise is visible in Figure 2.3, contained between 52 kHz and 60 kHz. This noise is problematic because this is the frequency range in which bat calls occur. Moreover, the noise spikes can easily be mistaken for a call.

The noise spikes occur at a fixed period. This period is the same as the period at which we write to the microSD card: changing the size of the buffer also changed the periodicity

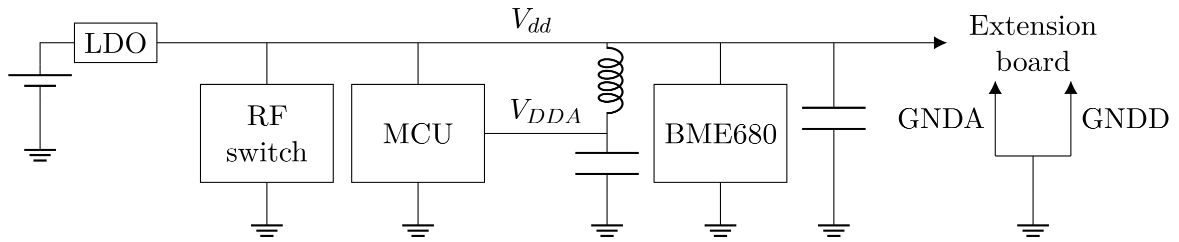


Figure 2.4: Supply tree of the main board.

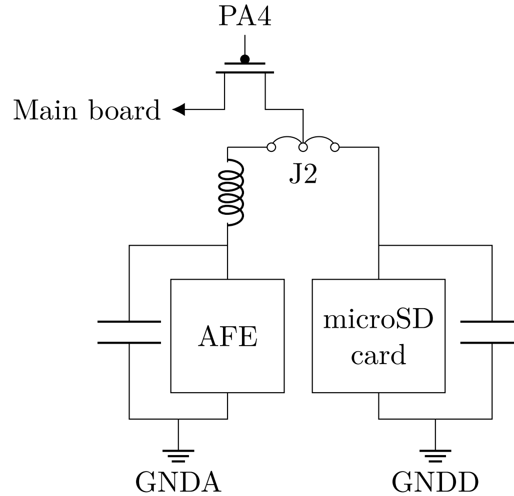


Figure 2.5: Supply tree of the extension board.

of the noise. This leads us to conclude that the source of the noise must be the microSD card.

Figures 2.4 and 2.5 show the supply tree of the entire platform. As we can see, the AFE supply and the microSD card supply come from the same connection to the main board, with a ferrite isolation to protect the AFE from high frequency noise.

2.3.1 Measurements setup

Measurements are made using an MSO 2024B oscilloscope as shown in Figure 2.7. It contains a low pass filter which automatically filters out unwanted noise. It is set to 140 kHz to filter out the high frequency noise and leave only the noise in the band of interest. All measurements are made with the ground pin as the reference voltage, measured on the main board probe.

An external supply has been added to the platform to try to eliminate supply noise from the microSD to the rest of the platform. It supplies the microSD card directly via a jumper as shown in Figure 2.6. It consists of a 9 V battery and a Power MB-V2 breadboard power supply which stabilizes the voltage at 3.3 V.

Measurements are taken for the platform with and without the external power supply to compare performance

The measurement nodes are V_{dd}, the voltage on the supply tree as shown in Figure 2.4, V_{ref}, V_{mic}, V_{mid} and V_{out}, the different nodes of the amplification and filtering chain of the AFE as shown in Figure 1.4.

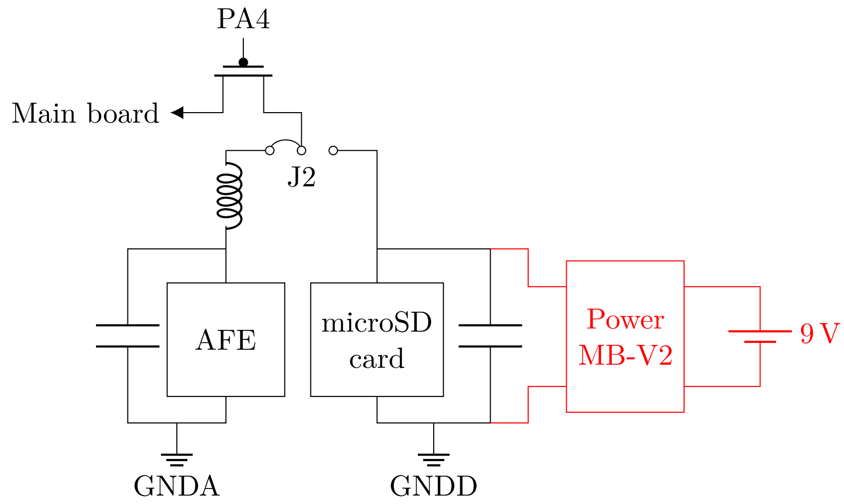


Figure 2.6: Setup for external supply of the microSD card.

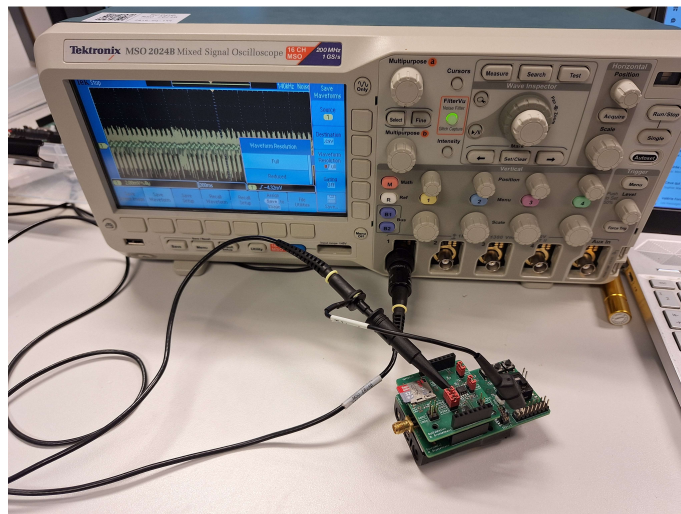


Figure 2.7: Measurement setup of the noise on the microphone signal. Measuring V_{dd}.

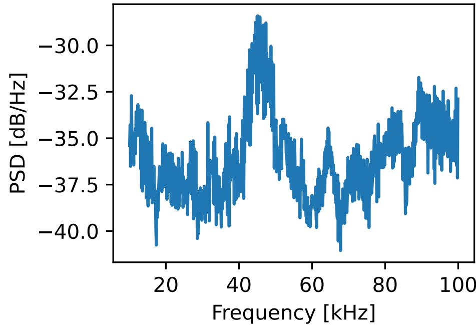


Figure 2.8: PSD of V_{dd} without external supply.

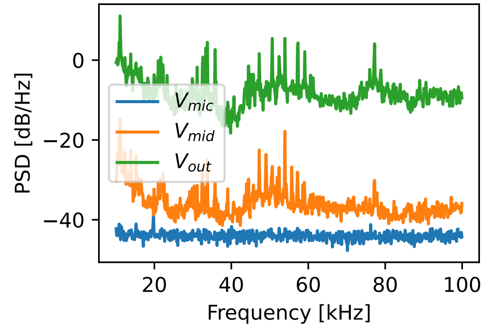


Figure 2.9: PSDs of the different nodes of the amplification chain without external supply.

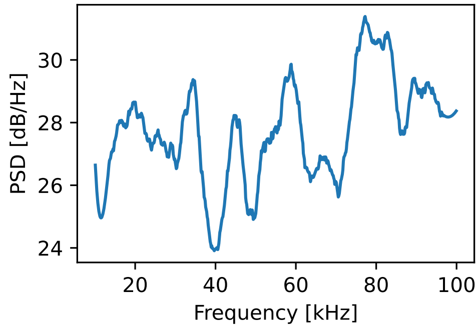


Figure 2.10: Difference between the PSDs of V_{out} and V_{min}.

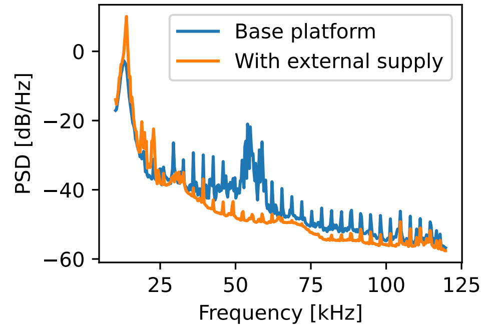


Figure 2.11: Comparison of the PSDs of the noise recorded on the ADC with and without external supply.

2.3.2 Results

Figure 2.8 shows the measurement of V_{dd} and Figure 2.9 shows the result of the measurements for V_{mic}, V_{mid}, V_{out} without external supply. As can be seen, the only signal measured on V_{mid} is the noise floor. No signal was measured at V_{ref} either.

In the case where the external supply was added, no signal was measured at any node. However, looking at the unfiltered data, the higher frequency noise was present in both cases.

2.3.3 Discussion

The disappearance of the noise at V_{dd} when the external supply is applied seems to indicate that this component of the noise is a supply noise. The higher frequency noise that does not disappear with the external supply is more likely to be capacitive crosstalk.

The fact that the noise does not appear on V_{ref} in either case shows that the ferrite is working and filtering the noise below the measurement threshold. The noise at V_{out} is coherent with an amplification of the noise at V_{mid}. Indeed, as show in Figure 2.10, there is a difference of about 27 dB between the two noises, which is close to 26.55 dB, the value of the amplification between V_{mid} and V_{out}. The variation in frequency of the difference

between the two PSDs could be due to a change in time of the signals, since the two PSDs were not measured exactly at the same time.

The conclusion is that although the ferrite attenuates the noise so that it is negligible on the supply of the AFE and microphone components, it is still present on the microphone signal. Its amplitude is small and therefore not detected by our measurement setup, but it is not negligible compared to the microphone signal. It is then amplified by the amplification chain, producing the noise seen from the ADC.

As shown in Figure 2.11, the external supply removes this supply noise and is therefore a satisfactory solution. To solve this problem at the design level, more filtering of the supply could be done so that the noise is sufficiently attenuated. Measures could also be taken to isolate the microSD card from the rest of the platform to prevent the noise from affecting other components. Another solution could be to directly support a power supply specific to the microSD card.

The resulting recording system is portable and easy to use. The system may crash when manipulated, possibly due to inconsistencies in the power supply of the microSD card. However, once set up, it can record for several hours without interruption.

2.4 Annotation

2.4.1 Methodology

One of the main difficulties in creating a dataset of bat calls is the annotation of sound clips to determine the ground truth. Several datasets have been created by the work of volunteers [21]. These volunteers were trained and asked to annotate the data. In our case, it would be difficult to use the same method.

Given the constraints under which we are working, we can expect the model to be much less complex than state-of-the-art algorithms. It follows that we can expect the performance to be much more modest and that creating a model that produces the same result as the state-of-the-art models would be an achievement. Therefore, we will annotate the data using an already established model and use this to create the ground truth of the model. The model used for annotation will be referred to as the teacher model.

This methodology is an example of knowledge distillation, a method of model compression where a student model is trained to imitate a teacher model, much bigger than the student [33].

In our case, it leads to two limitations. First, the teacher model was trained on data that did not come from the acquisition chain. This means that the model may not be as effective on our specific data, which reduces the quality of the annotations. Second, the performance recorded on the dataset is not comparable that of models tested on manually annotated data.

This choice of annotation method can be seen as shifting the goal from "detecting bats" to "giving the same answer as the teacher model", which should give approximately the same result if the teacher model is much better than what our model can achieve.

2.4.2 Selection of the teacher model

There are several factors to consider when choosing a teacher model.

- Its accuracy. The accuracy of the teacher model should be an upper bound on the accuracy of the final model.

Year	Paper		Output	Avail.	Metric
2018	Mac Aodha et al.	[21]	t_{start}, Pr(detect)	Yes	AP: 0.866-0.895
2019	Prince et al.	[15]			AUC: 0.975
2021	Beauvois et al.	[26]	t_{start}, Pr(detect), species	Yes	Prec: 0.84 Rec: 0.85
2021	Schwab et al.	[29]			Acc: 100%
2022	De Winter	[27]	t_{start}, Pr(detect), species		Prec: 0.62 Rec : 0.90
2022	Mac Aodha et al.	[30]	t_{start}, t_{end}, f_{min}, f_{max}, Pr(detect), species, Pr(species)	Yes	AP: 0.923-0.971
2022	Bellafkir et al.	[28]	t_{start}, Pr(detect), species		AP: 0.898-0.902

Table 2.2: Comparison of existing models with reported performance, availability and output information.

- The format of its output. If a model only outputs a start time of a call, it will be harder to know exactly in which time interval the call occurs.
- The availability of the model. Many models developed in the literature are not published, and those that may be of varying ease of use.

Table 2.2 summarizes information about some models that allow the detection of bat calls. Only three models are publicly available, BatDetect (Mac Aodha et al., 2018), batML (Beauvois and Dierckx, 2021) and BatDetect2 (Mac Aodha et al., 2022). The first two have GitHub repositories where the models can be downloaded and used via their frameworks. However, problems with package requirements have made them difficult to use. BatDetect is based on Python 2.7 which is deprecated. batML is based on BatMen [24], which is based on BatDetect but has ported its framework to Python 3.6. However, packages such as `nms` [51] still cause compatibility issues.

BatDetect2 also has a public repository, but it can also be installed on Python (version $\geq 3.8, < 3.11$) via pip, which greatly simplifies the dependency management. It has a simple command line interface for detection on many audio clips. It is the only model that outputs an end time for the call, which greatly simplifies the use of its output to create a dataset. Finally, it has excellent performance, outperforming all other models using the same metric and dataset.

All this led to the choice of BatDetect2 as the teacher model.

2.5 On-site measurements

2.5.1 Setup

We will use the board in the setup described in Section 2.3. The microSD card is supplied by the external battery via the voltage regulator which stabilizes the supply at 3.3 V. The rest of the platform is powered by the three AAA batteries. The system is set to record continuously on the microSD.

Three locations were used for the recording sessions. The first is a garden in Buzet, a small village in the province of Hainaut, Belgium. It was only used for testing purposes, such

Figure 2.12: Setup at the Louvain-la-Neuve site.

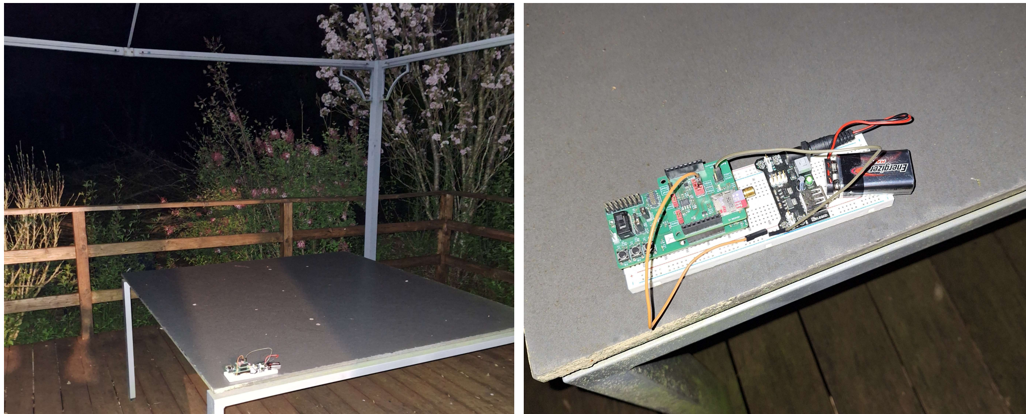


Figure 2.13: Setup at the Aubange site.

as checking that bats could be recorded by the device and that they were not hibernating. However, the recordings made there were still included in the final dataset.

The second location was at the window of a flat overlooking the lake in Louvain-la-Neuve. This site was chosen on the advice of Pr. Olivier Bonaventure, who had experience in bat monitoring and who indicated that bats could be recorded near the lake. This is the main recording site.

A final site was chosen to provide different data from the Louvain-la-Neuve site and to demonstrate the ability of the model to generalize to different environments. Based on reports from local residents about the visible presence of bats in the area, the site chosen was a garden in Aubange, a town in the province of Luxembourg, Belgium. The site and the recording setup are shown in Figure 2.13. Given the distance from the Louvain-la-Neuve site and the difference in environment (urban in Louvain-la-Neuve and more rural in Aubange), it was expected that the species distribution would be different and that, within a species, there would be variations in calls based on region and habitat.

2.5.2 Results

Location	Start time [DD/MM hh:mm]	Duration [hh:mm:ss]	Number of calls with probability >0.6
Buzet	14/03 23:11	00:19:25	0
Buzet	27/03 21:15	00:36:46	0
Louvain-la-Neuve	01/04 23:08	00:01:52	52
Louvain-la-Neuve	01/04 23:14	00:08:06	366
Louvain-la-Neuve	01/04 23:26	00:34:10	730
Louvain-la-Neuve	02/04 00:06	01:03:05	478
Buzet	06/04 23:08	00:26:19	56
Louvain-la-Neuve	11/04 21:09	00:27:13	1220
Louvain-la-Neuve	11/04 21:52	00:31:10	850
Louvain-la-Neuve	11/04 22:31	00:28:23	1180
Louvain-la-Neuve	11/04 23:01	00:45:14	2025
Louvain-la-Neuve	11/04 23:58	01:02:11	3486
Louvain-la-Neuve	12/04 01:04	00:59:59	2167
Aubange	21/04 01:12	00:32:51	0

Table 2.3: Summary of recording sessions.

A summary of the recording sessions is given in Table 2.3. Manual inspection of the audio recordings revealed that no calls were recorded in Aubange and during the first two recording sessions in Buzet. In the case of Aubange, this could be due to lower temperatures, as the night of 21 April was close to freezing, while both the 1st and 11th of April had temperatures around 10 °C. For Buzet, this could be explained by a low frequency of visits to the site and the fact that the hibernation period might not have ended yet.

Considering only these three sessions, the distribution of probabilities generated by BatDetect2 when used to detect calls is shown in Figure 2.14. As we can see, calls are detected up to a probability of 0.556. Of course, as there are no calls in the recording, accepting them would introduce false positives into our dataset. Listening to the sound clips detected as calls, it can be heard that the most probable detections are actually digital noise. This digital noise happens at random time instant. Its cause is most likely housekeeping tasks of the microSD which momentarily increase the time necessary to write to the card. This leads to deadlines being missed and data not being written, creating the high frequency noise heard.

It was chosen to use a threshold of 0.6 when creating the dataset. This gives a margin of error which allows to be confident that the calls considered are indeed bat calls. Given the low number of false positives between 0.5 and 0.6, the threshold could be slightly lowered. This would have resulted in more calls. However, as can be seen in Table 2.3, there are already many calls in the dataset when the threshold is set at 0.6. One thing to note is that only the most easily detectable calls are taken into account. This means that the test metrics will be optimistic as only with the simplest calls are used when training and testing our model.

For the sessions where calls were detected, Figure 2.15 shows the distribution of the

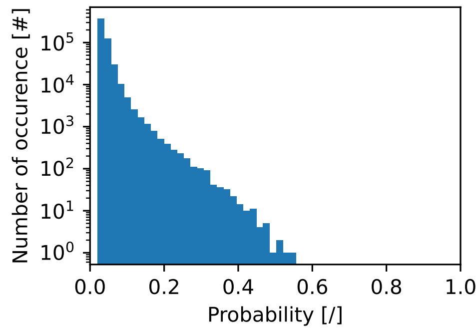


Figure 2.14: Histogram of call probabilities when no bat is recorded.

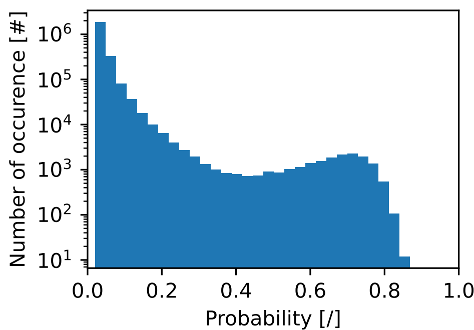


Figure 2.15: Histogram of call probabilities when bats are recorded.

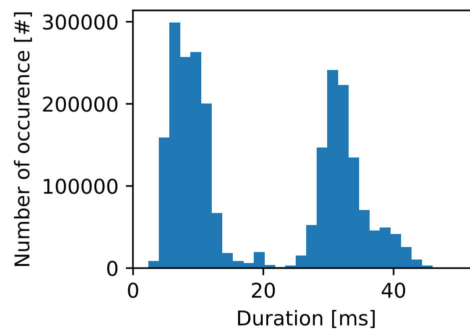


Figure 2.16: Histogram of call durations when bats are recorded.

detection probabilities. As can be seen, there is a maximum around a probability of 0.7, which is not present in Figure 2.14. This represents all detected calls. In Figure 2.16, we can see that the duration of the calls reported by BatDetect2 are divided into two lobes, the first concentrated around 7 ms and the second around 30 ms. If we look at the probabilities of each lobe, as shown in Figure 2.17, we can see that most detected calls have a duration under 22.5 ms. This suggests a possible additional way of sorting between true and false positives in the output of BatDetect2, which would be to take only those with a duration under 22.5 ms. We will not use this here as it would eliminate some high probability calls and require more work to validate the criterion.

Using the above threshold of 0.6 on the detection probabilities of BatDetect2, 12554 calls were recorded at the Louvain-la-Neuve site and 56 at the Buzet site. The recordings at Louvain-la-Neuve were made on two different nights. During the first night, 1626 bat calls were recorded for a total monitoring time of 3 hours, 7 minutes and 3 seconds. During the second night, 10928 calls were recorded over a period of 4 hours, 14 minutes and 10 seconds. This means that the first night had an average of one call every 6.9 seconds, while the second night had an average of one call every 1.4 seconds. The higher rate on the second night could be due to the later date, as bats could be slowly coming out of hibernation and becoming more active.

Considering only the calls with a probability above 0.6, Figure 2.18 shows that the durations of the calls are concentrated around 7 ms. This will be of interest later when training the model, as it will affect the optimal width of our spectrogram. Another similarly important information is the distribution of the lower and higher bound of the frequency

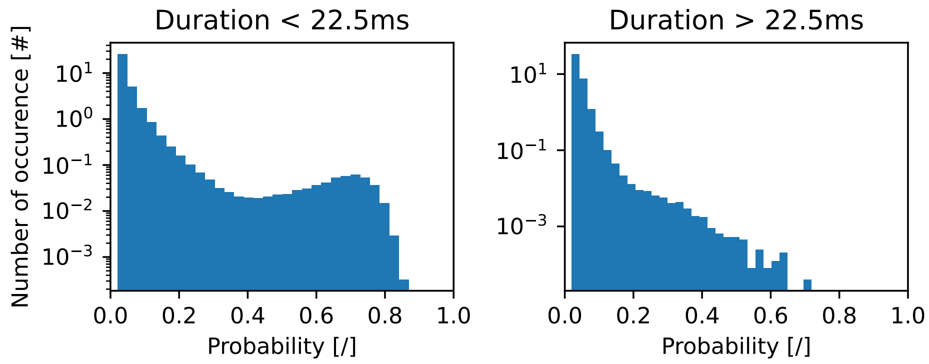
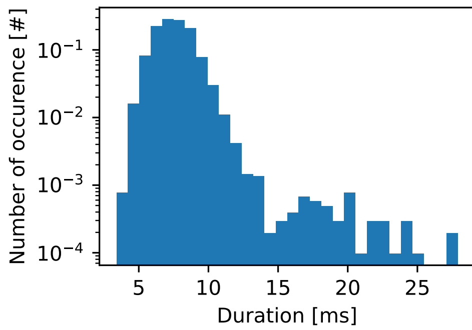


Figure 2.17: Comparison of bat detection probability based on duration.



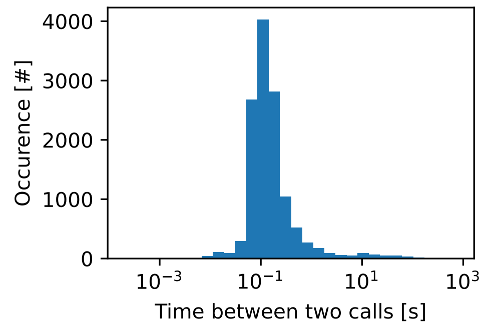


Figure 2.18: Histogram of call durations of calls kept in the dataset.

Figure 2.19: Histogram of distances between two calls kept in the dataset.

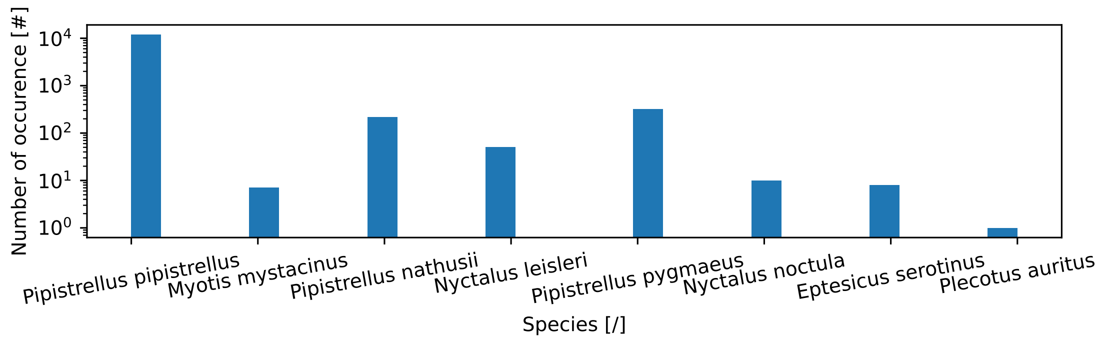
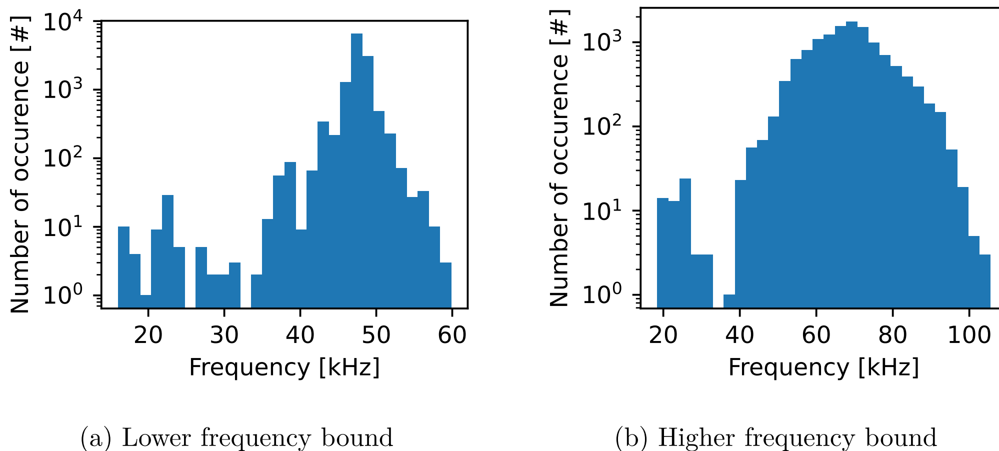


Figure 2.20: Histogram of detected species with bat recorded and threshold of 0.6.



(a) Lower frequency bound

(b) Higher frequency bound

Figure 2.21: Distribution of the lower and higher frequency bounds of calls kept in the dataset.

of the calls. As can be seen, most calls are comprised between 33 kHz and 105 kHz.

Figure 2.20 shows the distribution of species in the calls we kept. As can be seen, our dataset is dominated by calls of common pipistrelles (Pipistrellus pipistrellus). The second and third most common species are Soprano Pipistrelle (Pipistrellus pygmaeus) and Nathusius' Pipistrelle (Pipistrellus nathusii). These two are very close to the common pipistrelle and have more than an order of magnitude fewer recorded calls than the common pipistrelle. Note that all species reported by BatDetect2 are actually species present in Wallonia [52]. The fact that calls of common pipistrelles dominate our dataset could be a limitation, as our model could mainly learn the call of this species and be less efficient for other species.

Another statistic that can be analyzed on the annotations generated by BatDetect2 is the time between two calls. Figure 2.19 shows its distribution. We can see that most of the time there is less than one second between two calls. This means that most calls are consecutive calls from a single bat that is close enough to the sensor for its call to be recorded. A longer time between two calls could be due to the bat leaving the area around the sensor and no call being recorded until a bat (either the same or a different bat) comes close to the sensor again. This information informs us of the possibility of recording several calls at once. If we record one call, it is likely that others will follow closely.

2.5.3 Discussion

The methodology results in many calls being recorded and annotated at a low cost in both human and material terms. In just two nights of recording, more than 10 000 calls with a high level of confidence were recorded without the need for manual annotation.

However, there are several limitations that have been mentioned throughout this chapter. The first is that we are limited by the performance of BatDetect2, both in terms of the performance of our model and the quality of our dataset. Indeed, our ground truth will contain some false negatives and false positives, which will limit our overall performance and the evaluation of our performance.

A second limitation is that our dataset may be easier to classify than some others, such as Bat Detective. Our recording device was several metres above the ground, away from insects, and no other source of ultrasonic noise was detected. This means that bats

could be the only source of noise in the ultrasonic frequencies. This would lead to an easy classification, which would not generalize well to cases where other ultrasonic sources are present (e.g. frogs).

A third and final limitation comes from the limited setting of the recordings. Most of our dataset comes from a single site, with recordings made over a short period of time. There could therefore be a loss due to a change of season or a change of location.

In terms of the usability of the device for recording a dataset, the device is easy to use and portable. However, it is prone to crash. This can happen randomly at any point in the recording, but is more likely to happen at the start, at the end and when the device is transported. These crashes mean that the file appears empty on the microSD, even though data has actually been recorded. This was the case with the Aubange recording, where the device crashed at the end of the recording. However, it was still possible to recover the data. The method used was to create an image of the entire microSD card using `dd` [53]. The resulting file is then transformed into an audio clip, assuming that all the content of the image is audio data recorded by the system, and then decimating the file to obtain audio data playable by most software. The file obtained can then be listened to, which makes it possible to listen to all the data contained by the microSD card, even if they are not accessible from the file system.

Since all the blocks of the file system are aligned to an even number of bytes, all the audio data is correctly aligned throughout the listening. Moreover, this listening revealed that recordings do not directly overwrite data, even if the data has been deleted from the file system. This means that almost all the data ever recorded on the microSD card was still on the card, even though it had either been deleted from the file system or was not even registered in the file system due to a crash or reset of the system. This shows the ability to recover data from system failures and mishandling of the system.

Chapter 3

Algorithm design and implementation

3.1 State of the art of bat detection algorithms

On the subject of machine learning models for bat detection using full-spectrum features, Bat Detective [21], published by Mac Aodha et al. in 2018, was the first paper to tackle the subject and show a significant improvement over older algorithms. They evaluated their models against SonoBat [13], SCAN'R [54] and Kaleidoscope [55], which are three commercial closed-source software. Bat Detective beats all three by a wide margin in every metric and every dataset considered. The best performance from the commercial models was from Kaleidoscope on the Norfolk test set where it achieved an average precision of 0.553 while the CNN achieved at least 0.861 of average precision on the same test set. The dataset used was collected in the UK, Bulgaria and Romania. The dataset was split into three parts, one containing the data from Romania and Bulgaria, one containing the data from Norfolk (a county in the UK) and one containing the remaining data from the UK. The data from the UK, Bulgarian and Romanian sites were collected through the Indicator Bats Programme (iBats) [56] and the data from Norfolk were collected through the Norfolk Bat Survey [57]. The datasets have been collected and annotated with the help of thousands of volunteers.

Bat Detective used a CNN because other papers had shown its effectiveness in image classification tasks and, through the use of spectrograms, in audio classification tasks. They also compared the CNN with a segmentation method based on amplitude thresholding and with a random forest model using extracted call features as input. This comparison showed a clear advantage for the CNN model. The best average precision for the segmentation method was 0.506 on the Norfolk test, and the best case for the random forest was on the Romanian and Bulgarian test set, where it achieved an average precision of 0.674. The CNN based model outperformed both by a wide margin with an average precision of 0.861 and 0.863 respectively.

Two models have been created as part of Bat Detective, CNN_{FULL} and CNN_{FAST}. As its name suggests, CNN_{FAST} was intended to be a faster version of the model, running more than 5 times faster than CNN_{FULL} at the cost of a loss between 2.4 % and 9.8 % in average precision. CNN_{FULL} has three convolution layers of 32 filters of size 3x3, each followed by a 2x2 max-pooling layer and ending with a dense layer of 256 units. In contrast, CNN_{FAST} has only two convolution layers with 16 filters of size 3x3, still followed by a 2x2 max-pooling layer, and the dense layer has only 64 units.

Bat Detective led to several other works. For example, Batmen [24] by Franco and

Lipani used Bat Detective as a basis for species classification. This was extended with batML [26] by Beauvois and Dierckx for both the detection and classification at the same time. They compared several architectures, comparing the efficiency of separating or merging the detection and classification tasks. The best result for detection was obtained by a CNN doing detection only. Models that combined species classification and call detection showed inferior performance. Taking into account the multi-label classification of species, a better overall performance is obtained with a CNN model whose output was fed into an XGBoost, thus replacing the fully connected layer. Both models are trained to perform classification and detection simultaneously. XGBoost is a machine learning architecture based on ensemble learning from decisions trees whose generation is based on a gradient descent [58].

This last model served as the basis for the work of De Winter [27] who explored the possibility of binarizing the weights of the model. This allows faster execution and a smaller memory footprint at the cost of a loss in inference performance. He showed a reduction in size by a factor of 16 and a reduction in computation time of about 77 % with a loss of only 4 % in precision and 2 % in recall. The system was tested on two models of Raspberry Pi: the 3B and the 400. The Raspberry Pi 3B has a quad-core ARM Cortex-A53 running at 1.2 GHz[59] and the Raspberry Pi 400 has a quad-core ARM Cortex-A72 running at 1.8 GHz[60]. The system took 3.43 s to process a 3 s clip on the Raspberry Pi 3B and 1.2 s on the Raspberry Pi 400. Both De Winter and batML used the same dataset as Bat Detective, as it is one of the few freely available hand-annotated datasets of bat calls.

In 2022, a follow-up to Bat Detective was published by the same authors, called BatDetect2 [30]. They introduced the use of a self-attention layer, which adds memory to the model. This allows to take into account a longer time window at a reduced computational cost when compared with older techniques. The self-attention layer was integrated in a U-Net style architecture where an encoder first extracts features from the spectrogram which will then be decoded into the size and location of the call. This architecture showed significantly better performance than a baseline based on a random forest using extracted call features as input. It also had a globally better average precision than Bat Detective.

The dataset used for BatDetect2 is not the same as the one used for Bat Detective. The dataset introduced with BatDetect2 was collected in the UK, Yucatán (in Mexico), Australia and Brazil. Each location had its own split between training and test sets, so that the model was trained only on data from the same location as the data it was tested on. At the time of writing, this dataset is not yet publicly available, but is expected to be released in the coming months. When comparing the performance of the model with and without the self-attention layer, there was a negligible difference in the detection task: an average precision of 0.964 with the self-attention layer versus 0.962 without it. This suggests that a larger temporal memory is not an important aspect of detection. The classification task, on the other hand, improved significantly with the self-attention layer.

DeiT-tiny [28] by Bellafkir et al. is another example of the use of a self-attention layer, this time on the same dataset as Bat Detective with a clear performance gain when compared with Bat Detective. They used MSFBs and split the spectrograms into patches which were fed sequentially to the model.

In 2018, Prince et al. tackled the problem of bat detection using the low-cost device AudioMoth [15]. AudioMoth is an open source acoustic logger similar to the platform we are working on. It contains a single Arm Cortex-M4F processor running at 48 MHz, 256 kB of flash memory, 256 kB of RAM and the ability to write data to a microSD. Because AudioMoth has four times as many RAM and a Floating-Point Unit (FPU), it has more

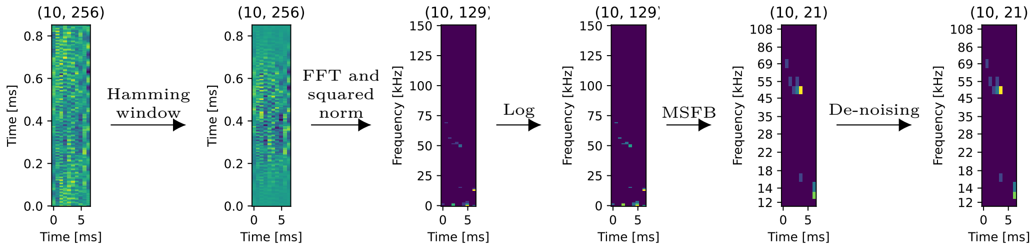


Figure 3.1: Full spectrogram generation pipeline.

computing power than our platform. They considered using Bat Detective, but did not because they thought it was "*not possible to produce an acoustic monitoring device capable of running a neural network such as this in real time without sacrificing both the low-cost and low-power requirement*" [15]. Instead, their solution was to use Goertzel filters to isolate a given frequency band on a window. The median of the result is then computed and compared a threshold to decide whether or not to record. The low computational cost of this solution allowed their system to work in real time. However, if there was a noise source other than a bat making sounds in the band the device is calibrated to, the system would still record those sounds because it did not detect bat calls specifically.

3.2 Chosen algorithm

3.2.1 Problem setting

The proposed algorithm is trained as a binary classification problem. Audio samples of a fixed length are classified as either containing a call or not.

The algorithm is divided into two stages. In the first phase, the audio samples are transformed into features, and in the second phase, these features are fed into a machine learning model. The output of the model then calculates a probability for the presence of a bat call in the input audio samples.

3.2.2 Feature extraction pipeline

The feature extraction pipeline will be mainly based on the Bat Detective pipeline. However, due to the limitations of the platform, modifications are made to maintain a real time execution. The features are computed vector by vector. The complete pipeline to generate a vector is shown in Figure 3.1.

The sample rate, the size of a window, the number of samples between two windows and the number of windows are all hyperparameters that are optimized in Section 3.4.

The first step in the computation is a multiplication by the shaping window. The purpose of this window is to reduce distortion. These distortions come from the truncation of the signal when the Fast Fourier Transform (FFT) is applied. Indeed, the FFT ideally takes one period of a periodic function. The truncation thus creates a discontinuity between the beginning and end of the signal, causing spectral leakage. The shaping window will flatten both ends of the window, making it closer to a periodic function.

Two windows, shown in Figure 3.2, were considered. The first is the Hann window, defined as

$$w[n] = 0.5 - 0.5\cos\frac{2\pi n}{N-1}, \tag{3.1}$$

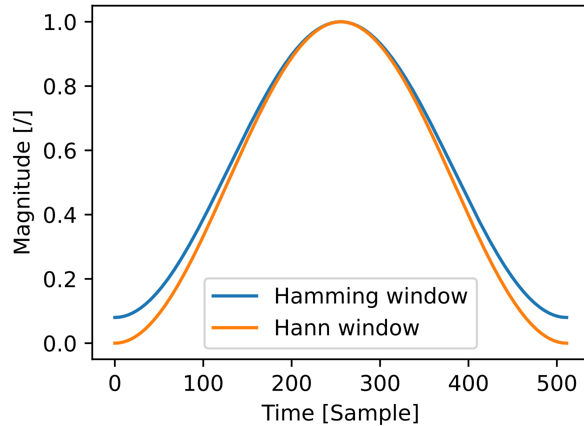
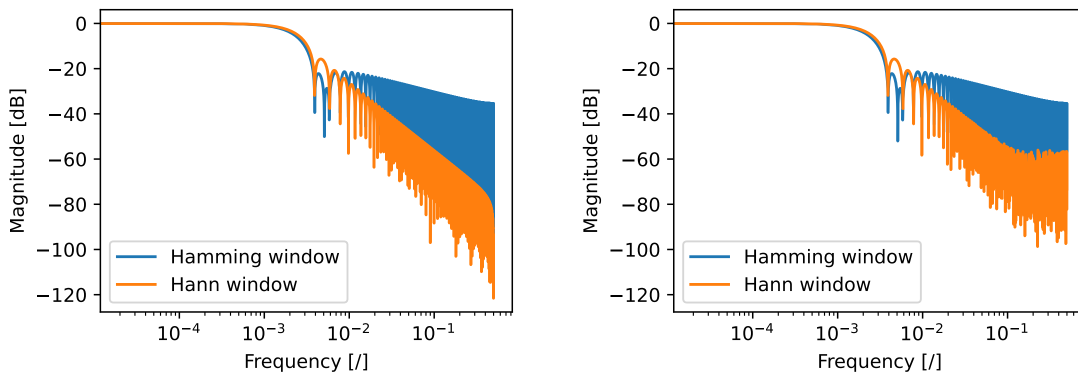


Figure 3.2: Hann and Hamming windows for a window of size 512.



(a) No quantization.

(b) With 16 bit fixed-point quantization of the taps.

Figure 3.3: Frequency response of Hann and Hamming windows.

with $0 \leq n < N$. It is used by several models such as Bat Detective or BatDetect2. The second shaping window considered is the Hamming window, defined as

$$w[n] = 0.54 - 0.46 \cos \frac{2\pi n}{N-1}, \tag{3.2}$$

with $0 \leq n < N$. The two shaping windows are relatively similar, as can be seen from their frequency responses in Figure 3.3a. The Hann window has a better roll-off for its side lobes, but the Hamming window has a lower first side lobe.

We need to consider the need to quantize the taps of the windows. Since the platform does not have a FPU, it would be very costly to use a floating point representation of the shaping window. Figure 3.3b shows the impulse response of the windows with quantized taps. The quantization has no visible effect on the Hamming window. The Hann window has a slight loss of performance at high frequencies. We also have to consider that the multiplication is quantized. As the Hann window is equal to 0 at each end of the window and is flat around 0, given the quantization required to fit into the 16 bit format, this means that many samples located at the end of the window will have no effect on the result, unless their amplitude is very large. Since the performance of the Hamming window

is satisfactory, and since it does not suffer from the same quantization problem due to its non-zero value at its extremities, the Hamming window was chosen.

The FFT is then applied. It takes the data from the time domain and translates it to the frequency domain. We then have complex values at the output. Since the input data is real, the output of the FFT is symmetric around 0 and only the values with positive indexes are computed. These complex values are then converted to the square of their magnitude, to which a logarithm is then applied. This logarithm was inspired by Bat Detective, and attempts to remove this step resulted in a large loss of performance in the machine learning model.

Given data in fixed point format, the formula used for the computation of the logarithm is

$$l(x) = \Big(15 + \log_2 \Big(2^{-15} + x\Big)\Big)\, 2^{-4}. \tag{3.3}$$

For all values between 0 and 1, it is less than 1, which is necessary to avoid overflow on a fixed point calculation. This logarithm gives more weight to the variation of smaller values.

Finally, the mel-scaled filters are applied to the data to obtain a vector of MSFBs. This is done by matrix multiplication. The matrix is generated by the function `compute_melmat` from the Python package `PyFilterbank` [61]. Using MSFBs has two advantages. First, they have been shown to increase model accuracy in state-of-the-art models [17]. Second, they reduce the number of dimensions of the data, which will lead to a great speed-up for most machine learning models (e.g. CNN). The number of MSFBs per vector is then another variable that allows a trade-off between model speed and inference performance. This is particularly important in this application if we want to achieve a real-time operation. In this step, only the band of interest is kept. The matrix multiplication simply contains a value of zero for samples outside the frequency band. The lower limit of the band is set at 10 kHz and the higher frequency is kept as a hyperparameter.

Once all the vectors have been generated, a de-noising process is applied, the mean value of each frequency band is computed and subtracted from said band. In the case of Bat Detective, they mention that this significantly improves performance [21]. It is worth noting that Bat Detect did not apply mel-spaced filtering, but when applied to the proposed design, similar improvements are observed.

3.2.3 Machine learning model architecture

Given the hardware constraints imposed by the platform used, the designed model must be much smaller than most of the models mentioned above. Models based on a self-attention layer, for example, are far too large to fit in memory and have no chance of running in real time on a single-core system.

The success of the CNN architecture in the detection task in both Bat Detective and BatML made the CNN seem like a relevant choice. Also, because of its speed, CNN_{FAST} is used as a starting point.

The proposed architecture uses a CNN with two convolutional layers, each followed by a max-pooling layer. The number of filters of both convolutional layers are kept as hyperparameters to be optimized. The kernel size is set to 3x3, as in Bat Detective. Two dense layers are then applied. The first with an output of size 28 and the second with an output of 1, which is the probability of detection. The size of 28 for the first dense layer is largely arbitrary. It did not seem to have much effect on the results and was not investigated further.

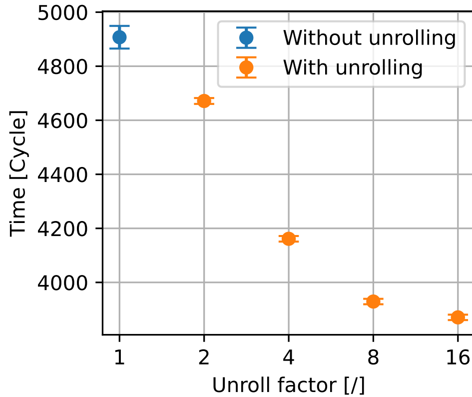


Figure 3.4: Time measurements of data formatting over unroll factor. Buffer of size 512.

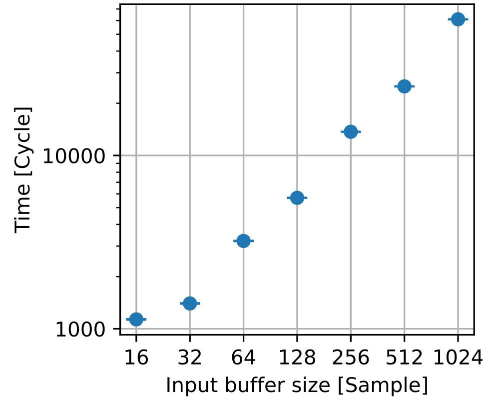


Figure 3.5: Time measurements of CMSIS `arm_rfft_q15` function over buffer size.

3.3 Implementation of a detection algorithm

3.3.1 Feature extraction pipeline

The pipeline works in 5 stages, separated into the formatting of the data, the FFT, the treatment of the complex samples, the mel-scaled filtering and the de-noising. The full code for the pipeline is given in Appendix B.

The formatting of the data consists of 3 operations. First, we remove the mean of the data. Since the samples of the ADC are in the range 0 to 2^{12}, it oscillates around 2^{11}, while our analogue input oscillates around $V_{dd}/2$. This subtraction is done with the `SSUB16` assembly instruction, which is a SIMD instruction that allows one to do two 16 bit subtractions in a single cycle. Then, we scale the data up with a bit shift. The value of this shift is calculated before the vector calculation so that the maximum of all samples used for the spectrogram is just below the overflow. Finally, the multiplication is performed using the Hamming window. This whole stage is done in a single *for loop* to minimize memory calls. The loop is repeated 8 times. Figure 3.4 shows the dependence of the unroll factor on the number of execution cycles. In all figures in this section, the buffer size refers to the number of samples in the input raw audio buffer. We can see that unrolling allows a gain of about 1037 cycles. The number of cycles taken by the program tends to a minimum as the unroll factor increases. Increasing the unroll factor beyond 16 gives negligible improvements. This stage takes about 3871 cycles to complete for a buffer size of 512. This stage scales linearly with the buffer size.

The FFT is done using the `arm_rfft_q15` function from the CMSIS library. It calculates the FFT of the signal, taking into account that the samples have no imaginary parts. To avoid overflows, the output of the functions is divided by the size of the input (so a bit shift to the right of 9 for an input of size 2^9) compared to the ideal FFT. Due to its recursive implementation, the function only accepts inputs whose size is a power of 2. This limits the choice of buffer size used to generate a vector. With 25 033 cycles for a buffer size of 512, this is the operation that will take the most time in the pipeline. Figure 3.5 shows the evolution of the computation time with the size of the buffer, which scales in $\mathcal{O}(n \log n)$.

The third stage, the treatment of complex samples, begins with the calculation of the norm. This is done with the `SMUAD` assembly instruction. It takes four 16 bit arguments,

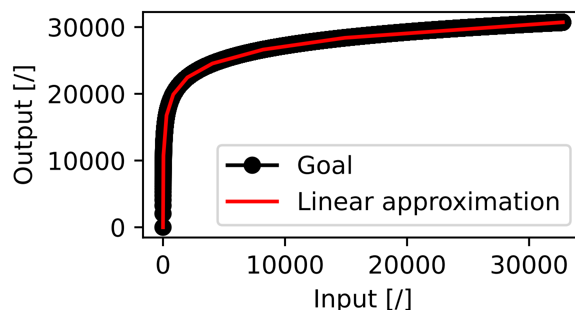


Figure 3.6: 8 segments piecewise linear approximation of the logarithm function.

Algorithm	Computation time [cycles]
CMSIS implementation	53 374
8-segment approximation	2276
4-segment approximation	1451
2-segment approximation	1014

Table 3.1: Comparison of computation times for logarithm computation for a buffer of size 512.

a_1, a_2, b_1 and b_2 as two 32 bit numbers and computes $a_1b_1 + a_2b_2$. By passing the same argument twice, with the first 16 bits as the real part of the sample and the last 16 bits as the imaginary part of the sample, this function will compute exactly the complex norm squared in a single cycle.

We then apply the logarithm. The CMSIS library logarithm, `arm_vlog_q15`, had good accuracy but proved too slow for the application. In our case, we do not need an exact calculation of the logarithm: we only need the non-linearity of the operation to help the model learn. We thus used an approximation. The approximation is a piecewise linear function with 8 segments. In software, the way this function is implemented by a binary search on which line equation to use. Table 3.1 shows the execution time of the CMSIS `arm_vlog_q15` function and our implementation for 4, 8 and 16 segments. We see an increase in computation time with the number of segments used to do the approximation.

The equations for the lines were found by applying three constraints. The first is a continuity constraint. The second consisted in having the same values as the function to be fitted at 0 and at 2^{15}. The last constraint applied forced the last three slopes that are less than one to be 0.5, 0.25, 0.125 so that they could be applied efficiently with bit-shift operations. The remaining free parameters were then modified by hand to fit the curve as well as possible. Figure 3.6 shows a comparison between the ideal logarithm and the chosen one.

The whole third stage (squared complex norm and logarithm) is once again done in a single *for loop* with loop-unrolling. However, as shown in Figure 3.7, unrolling using the pragma of GCC actually increased the number of cycles. The expected decrease in cycles was still observed when unrolling the loop explicitly. This method was therefore chosen.

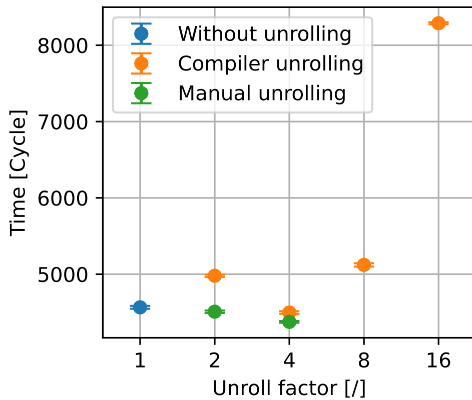


Figure 3.7: Time measurements of third stage over loop-unrolling factor. Buffer of size 512.

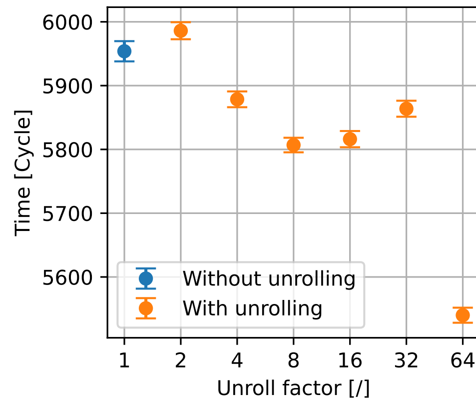


Figure 3.8: Time measurements for mel-scaled filtering over loop-unrolling factor. Buffer of size 512 and 64 MSFBs.

As mentioned above, mel-scaled filtering is implemented as a matrix multiplication. However, this matrix is actually sparse. A naive implementation of the function would therefore result in a lot of multiplication by 0 and waste of time. To take advantage of this sparseness, we store the matrix as a jagged array containing only the non-zero elements. Two arrays store the length and starting position of each row in the original matrix. We can then implement the matrix multiplication as a series of dot products computed using the `arm_dot_prod_q15`, each of which computes an element of the vector of MSFBs. This avoids unnecessary computations. Given the short length of an MSFB vector, the loop over the elements of the vector is fully unrolled. As can be seen in Figure 3.8, fully unrolling the loop results in a gain of about 414 cycles compared to the case without loop unrolling. This step takes 5540 cycles with a buffer size of 512 and 64 MSFBs. The number of cycles per computation should have a linear dependence on both the buffer size and the number of MSFBs.

All the operations stated before are performed on each vector. Then comes the denoising step. For this step we iterate over all the frequency bands, compute the average over all the vectors and subtract it from each vector. The loops have a relatively small number of iterations and are therefore fully unrolled. It takes 6246 cycles for 10 vectors with 64 MSFBs. It scales linearly with both number of vector and number of MSFBs.

3.3.2 Quantization of inference model

The platform does not have a floating-point unit. However, the model obtained after training is defined entirely with floating-point weights and operations. Running it would therefore require floating-point emulation, which is time-consuming. To run the model under the constraints of the platform, the weights and operations need to be quantized. Quantization is a process by which weights (and therefore operations) are converted into another data format. Quantization of weights usually reduces the number of bits on which they are stored (e.g. 32-bit floating-point to 8-bit fixed-point) but some quantizations only change the representation of data, which can still have advantages at the operation level (e.g. 32-bit floating-point to 32-bit fixed-point).

Quantization has two main effects. Firstly, the model will take up much less memory as the 32-bit floating point weights are converted to 8-bit integer weights. The model should

therefore take up about 4 times less memory. Secondly, the execution of the model will be sped up. This effect comes not only from the avoidance of floating-point emulation as mentioned above, but also from the use of the SIMD capabilities of the CPU.

The disadvantage of this technique is that quantizing the weights reduces the precision of the weight values, and quantizing the operations also reduces the precision of the intermediate values used in the computation. This loss of precision should lead to a loss of performance, which will be evaluated in Section 3.4.5.

Quantization is done after training using TensorFlow Lite's built-in converter [62]. Given a representative dataset, the converter is able to convert the model into a quantized model that takes 8 bit fixed-point data as input and output and uses only 8 bit operations. The representative dataset is a subset of the full dataset that is used by the converter to calibrate the model.

This calibration determines an optimal linear transformation to apply to the input data in order to use the quantized model. This adds an extra step between the features and the inference, which is efficiently done at the same time as the de-noising operation.

3.3.3 Embedded inference

To port the model to the platform, we use X-CUBE-AI, an STM32Cube expansion package [63]. This allows the automatic generation of code from the TensorFlow Lite model, taking into account its quantization and automatically optimizing it. The package also includes quality of life features such as model validation, as well as time and memory usage measurements.

3.4 Parameter selection

3.4.1 Validation metrics

To validate and compare model performance, we will introduce several metrics. These metrics will be computed on the probabilities returned by the model for a given test set. In fact, most models don't directly return a binary classification, but a value between 0 and 1, usually called a probability. The higher the probability, the more confident the model is in its prediction.

		Predicted condition	
		Predicted positive	Predicted negative
Actual condition	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

Table 3.2: Diagnostic testing diagram. [64]

As shown in Figure 3.2, there are four cases in which we can find ourselves when making a prediction: True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN). The aim, of course, is to have 0 FP and FN so that we always give the right answer. In practice, however, we rarely get a perfect model. To measure the efficiency of a model, several metrics have been defined. One is precision, defined as

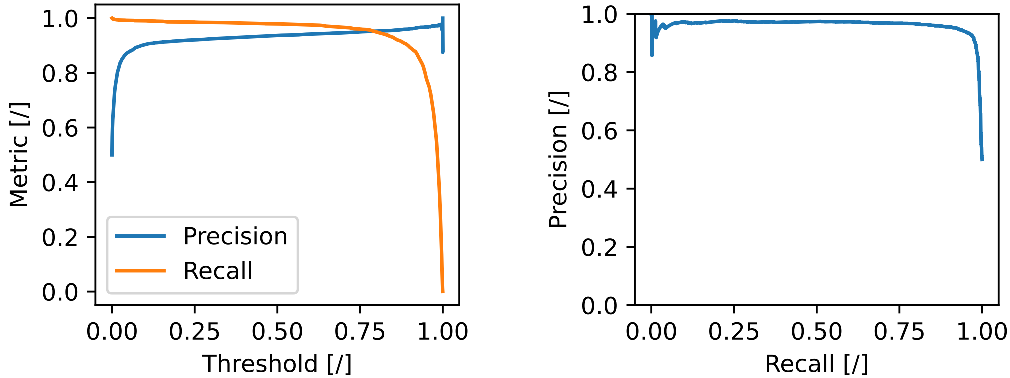
$$\text{Precision} = \frac{TP}{TP + FP}, \tag{3.4}$$

which, when measured on a sufficiently large representative dataset, can be thought of as the probability that an element predicted to be positive is actually positive. Another metric is recall, defined as

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{3.5}$$

which, under the same conditions as before, can be seen as the probability that a positive element is predicted to be positive by the model. Both of these metrics should tend to 1, as the evaluated model is better.

Precision and recall are actually antagonistic. Precision improves when the model rejects many inputs, even if they are very likely. Conversely, recall improves when the model accepts most of the inputs, even if they are very unlikely. It is therefore important to keep these two values in mind and not to focus on only one of them. As shown in Figure 3.9a, the trade-off between the two values can be directly controlled by the threshold we apply to the probabilities computed by the model. If we vary this threshold between 0 and 1 and plot the pairs (recall, precision), we get a precision-recall curve (e.g. Figure 3.9b). This precision-recall curve is a graphical representation of the aforementioned trade-off. The larger the area under the precision-recall curve, the more leeway we have in the trade-off. This area is also called the Average Precision (AP) because it is the average of the precision over all the recall values. This AP will be the first metric.



(a) Precision and Recall against threshold.

(b) Precision-Recall curve.

Figure 3.9: Example of precision and recall curves.

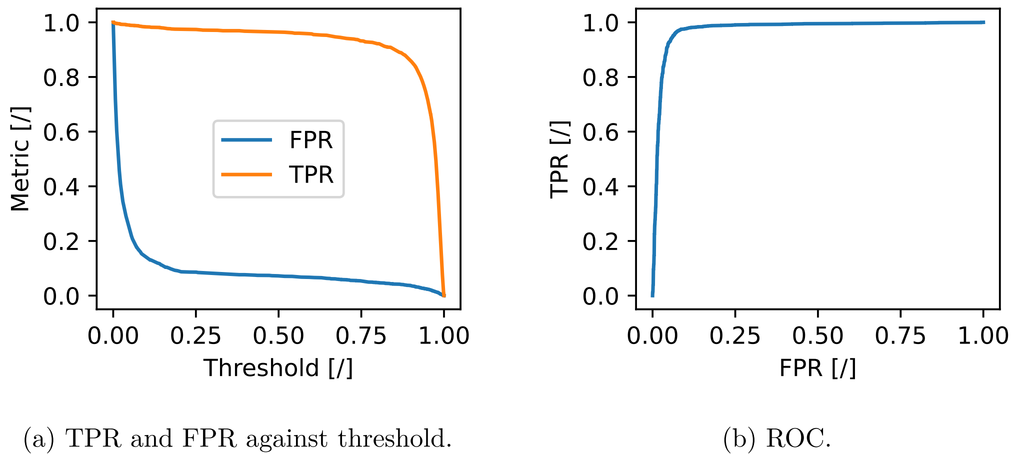
Our second metric will be the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC). The ROC is defined as the curve obtained by plotting the True Positive Rate (TP) against the False Positive Rate (FP) for all possible thresholds (e.g. Figure 3.10). The TPR is actually equal to the recall. The FPR is defined as

$$FPR = \frac{FP}{TN + FP}. \tag{3.6}$$

The FPR can be understood as the probability of false alarm, the probability that a given negative data will be classified as a positive.

Stage	Root mean squared error [/]
Post-formatting	0
Post-FFT	1.13
After complex norm and log	0
Output MSFBs	0

Table 3.3: Error done on computation of the features at different points of the pipeline.



(a) TPR and FPR against threshold.

(b) ROC.

Figure 3.10: Example of TPR and FPR curves.

Both AUC and AP will always be between 0 and 1. They should tend towards 1 as the model gets better. The AUC is one of the most commonly used metrics for binary classification because of its interpretability, ease of computation, and representativeness of model performance. However, it can be misleading in the case of unbalanced datasets [65]. Even if we make a balanced dataset, the task at hand is still highly unbalanced, with more negative cases than positive ones. We will keep track of both the AUC and the AP to be complete. This will also allow us to discuss possible differences in the behaviours of the two metrics.

3.4.2 Feature extraction emulation

To optimize the feature extraction pipeline, fast iteration over its components is required. There are two ways to go from raw sounds to spectrograms on which we can train the model. The first method is to transfer audio data to the board to compute the features with the implemented chain. However, the time taken to transfer the data over the entire dataset seems prohibitive and does not allow for fast iteration.

The second option is to emulate the C code in Python. This allows for efficient parameter and pipeline changes as all the work from audio clips to spectrograms is done on the computer. This comes at the cost of slight errors in the computed spectrogram compared to the embedded computation. These errors are due to approximation in the emulation for complicated functions (in our case mainly the FFT).

For a chosen test vector, Table 3.3 shows the error made. As we can see, there is no

error in the final result. The only error introduced is the FFT. However, the rounding that takes place in the following stages removes these errors. A validation with more diverse inputs should be done to really validate the accuracy of the method. Nevertheless, this example shows that the Python code behaves in the same way as its C counterpart. The Python code used is given in Appendix C.

3.4.3 Label generation

Our label generation pipeline takes as input a CSV file containing two pieces of data: the place where the audio clips are stored, and the name of a CSV file containing the information generated by the teacher model. Each audio clip in which a bat call has been detected is then processed.

There are four parameters which impact the result of the spectrogram computation:

- N_{vec}, the number of vectors of the spectrogram.
- N_{msfb}, the number of MSFBs.
- S_{hop}, the number of samples between two vectors.
- S_{fft}, the size of the sample buffer used for computation of a vector.

They have been chosen to allow a direct interpretation of their impact on the computational complexity. T_{spec} is particularly important as it defines the time over which the whole computation (feature generation + inference) must be done if we want continuous monitoring.

We then take steps of size S_{hop} and create a spectrogram at each step. The spectrograms are created by splitting the audio into N_{vec} blocks of size S_{buf} whose starts are separated by S_{hop} samples. They are then passed to the spectrogram generation algorithm described in Section 3.4.2.

The resulting spectrograms are then separated based on the presence or absence of a call in them. This is computed based on the annotation, which gives t_{start} and t_{end} for each spectrogram. For a given spectrogram starting at time t_{spec} there are two cases to consider. In the first case, the duration of the call is under the total spectrogram time. As can be seen in Figure 3.11a, it is considered to contain a given call if

$$t_{\text{end}} - T_{\text{spec}} < t_{\text{spec}} < t_{\text{start}}. \tag{3.7}$$

In the second case, the full spectrogram time is under the call duration. Then, as seen in Figure 3.11b, the spectrogram is considered to contain a bat call only if

$$t_{\text{start}} < t_{\text{spec}} < t_{\text{end}} - T_{\text{spec}}. \tag{3.8}$$

These two cases can be combined by the formula

$$\min\left(t_{\text{start}}, t_{\text{end}} - T_{\text{spec}}\right) < t_{\text{spec}} < \max\left(t_{\text{start}}, t_{\text{end}} - T_{\text{spec}}\right). \tag{3.9}$$

All spectrograms rejected by the previous condition for all calls could be taken as spectrograms without calls. However, to avoid adding false negatives to the ground truth, a safeguard was added. This spectrogram is defined as β in number of spectrograms. A spectrogram is thus considered containing no calls if, for all calls, it respects either

$$\max\left(t_{\text{start}}, t_{\text{end}} - T_{\text{spec}}\right) + \beta T_{\text{spec}} < t_{\text{spec}} \tag{3.10}$$

(a) Case where the call duration is shorter than the window.

(b) Case where the call duration is longer than the window.

Figure 3.11: The two cases to test for call in a window. Upper window is the point from which we begin to consider that it contains a call, lower window is where we stop.

Symbol	Meaning
$N_{filter,1}$	Number of filters in the first convolutional layer
$N_{filter,2}$	Number of filters in the second convolutional layer
f_s	Sampling frequency of the input audio stream
S_{hop}	Distance in samples between two vectors
N_{msfb}	Number of MSFBs per vector
N_{vec}	Number of MSFBs vectors
S_{fft}	Size of the FFT

Table 3.4: Variables used in the search for model optimum and their meaning.

or

$$t_{\text{spec}} < \min\left(t_{\text{start}}, t_{\text{end}} - T_{\text{spec}}\right) - \beta T_{\text{spec}}. \tag{3.11}$$

For each call, one spectrogram is then randomly selected from all the valid ones to be included in the dataset as a positive in the ground truth. An equal number of spectrograms containing no calls are also randomly selected to serve as negatives in the ground truth.

3.4.4 Methodology

The goal is to determine the best possible algorithm for a given computation time. To do this, we will evaluate the AUC, ROC and computation time. Table 3.4 summarizes all variables used in the exploration with their meaning. The exploration is done by hand, trying to find a local maximum.

TensorFlow is used for the framework. The optimizer used is the Adam algorithm. It is a first-order gradient based optimizer based on adaptive estimates of lower-order moments [66]. The loss used is the binary cross-entropy. It measures the difference between the distribution of the predictions of the model and the distribution of the ground truth [67]. The training rate is set to 10^{-3} and the model trains for 15 epochs. Those values were chosen such that no overfitting was observed, but the model still converged.

The model is evaluated using a KFold cross-validation with 20 splits. The dataset is thus divided in 20 folds and each fold is used to evaluate performance when all other folds are used for training. All probabilities computed are stored, and we use them to compute the metrics once all folds are treated. As only one spectrogram is generated for a call, there is no risk of a single call being found in both the training and test splits. We run this cross-validation 5 times. This is done in parallel to save time. We then take the average of

Symbol	Meaning
T_{comp}	Full computation time
$T_{\text{c,feat}}$	Computation time for the generation of the features
$T_{\text{c,inference}}$	Computation time for the inference based on the features
$T_{\max}$	Time taken by the search of the maximum value for normalization
S_{feat}	Number of samples used for the generation of all the features
N_{vec}	Number of MSFB vector in the features
T_{format}	Time taken by the formatting stage (normalization + windowing)
T_{fft}	Time taken by the FFT
$T_{\text{mag+log}}$	Time taken by the treatment of complex samples (magnitude + log)
T_{msfb}	Time taken by the filtering by the mel-scaled filters
N_{msfb}	Number of MSFBs per vector
$T_{\text{de-noise}}$	Time taken by the de-noising step

Table 3.5: List of symbols for computation time estimation with their meaning.

the 5 values. This repetition of the 5 cross-validations allows us to reduce the randomness coming from the split and from the initialization of the models.

The data set is computed from the raw audio based on the parameters each time. The whole evaluation is done with 5 different seeds, and we take the average of the metrics. This allows us to reduce the randomness involved in choosing which spectrogram to take from a given call. This means that for a single set of parameters, the model is trained 625 times: 25 times for the splits, 5 times for the repetition of the cross-validation and 5 times for the different seeds used to generate the data set. All these repetitions allow us to have a stable value that we can trust.

All audios with calls are used in the dataset, except the file recorded on the 11th April at 23:58, which is kept as a test set. It contains 3486 calls, which result in 27.77 % of the dataset used as test set. This test set will not be accessed during training and model selection. It will only be used to validate algorithm performance.

To evaluate the time, the number of cycles taken was measured for each stage of the pipeline. This was done with 64 MSFBs for a buffer of size 512 and with 21 MSFBs for a buffer of size 256 and both used 10 vectors. The computation time was then evaluated with the following formulas:

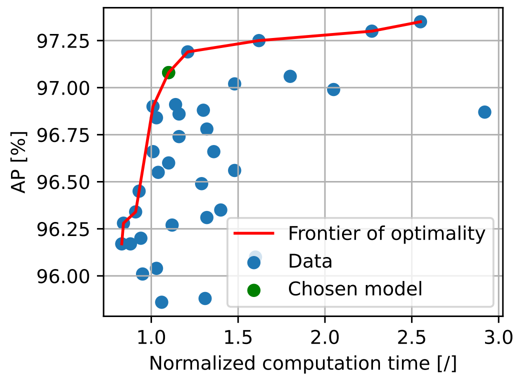
$$T_{\text{comp}} \approx T_{\text{c,feat}} + T_{\text{c,inference}} \tag{3.12}$$

where

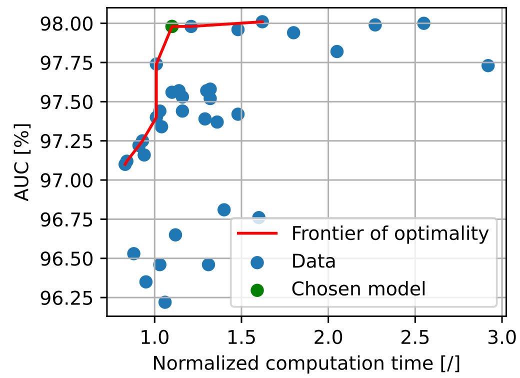
$$T_{\text{c,feat}} \approx T_{\max}\frac{S_{\text{feat}}}{S^*_{\text{feat}}} + N_{\text{vec}}\left(T_{\text{format}} + T_{\text{fft}} + T_{\text{mag+log}} + T_{\text{msfb}}\frac{N_{\text{msfb}}}{N^*_{\text{msfb}}}\right) + T_{\text{de-noise}}\frac{N_{\text{msfb}}}{N^*_{\text{msfb}}}\frac{N_{\text{vec}}}{N^*_{\text{vec}}}. \tag{3.13}$$

The variables followed by an asterisk indicate the values at which the reference time was computed. $T_{\text{c,inference}}$ is measured for each model using XCUBE-AI. The meaning of other variables is given in Table 3.5.

It should be noted that this test is carried out without approximation of the logarithm, whereas the times consider that the approximation is applied. On the other hand, the



(a) AP



(b) AUC

Figure 3.12: Distribution of models over metrics and normalized execution time with optimality boundary.

metric measured is done on the cross-validation without quantization, which could lead to a slight loss of performance when introducing it. This means that there could be a slight difference between the performance we claim is achievable for a given computation time and the actual performance achievable when approximations are taken into account. However, it is expected that all models will be equally affected by the loss of performance, so the trends observed should still hold.

3.4.5 Results

The full data from the optimization is given in Appendix A.

The results show a performance optimum when the algorithm processes between 7 ms and 8 ms of audio data to generate features. This is consistent with the observations of Section 2.5.2, in which it was observed that most calls in the dataset have a duration of around 7 ms. This shows that the model works best with a window that best fits the call. Note that our tests only generate features that contain a whole call if our window size is larger than the call, so this creates a bias. In fact, in real execution, only parts of calls may be recorded. This will be addressed in further characterization of the system.

Increasing the number of filters in both convolutional layers improves performance, but at a prohibitive cost. Reducing the sampling frequency to 200 kHz and the upper limit of the frequency band to 90 kHz results in a significant loss of performance. Reducing the sampling rate allows a longer time span to be considered at a constant computational cost. As shown in Figure 2.21b, calls can go up to 105 kHz. A part of some calls is thus out of the band with the reduced sampling rate. Another factor which may explain the performance loss is that mel-scaled filtering reduces the resolution of frequencies at the end of its band more. Since we are reducing the band bound, we may also be reducing the resolution of frequency bands that were of high importance for classification.

Figure 3.12a and 3.12b show the distribution of the AP and the AUC of the result of the tested parameters against the normalized computation time. The normalized computation time is calculated as the ratio between the computation time (time taken to compute the features and apply the inference) and the time taken to acquire the samples on which the computation is performed. Mathematically, the normalized computation time can be seen as the inverse of the sampling duty cycle. Indeed, if the normalized computation time is

Parameter	Value
# filter in first convolutional layer	4
# filter in second convolutional layer	4
N_{vec}	10
N_{mfsb}	20
S_{fft}	512 samples
S_{hop}	270 samples
f_s	300 kHz
f_H	120 kHz
S_{feat}	2942 samples
Result	**Value**
Flash memory requirement	26.69 kB
RAM memory requirement	3.89 kB
Computation time	11.47 ms
Normalized computation time	1.17
AUC	97.98 %
AP	97.08 %

Table 3.6: Chosen model parameters and results.

greater than one, the only solution for a real-time system will be to ignore some samples between two predictions in order to have enough time to perform the computations.

The optimality boundary is also shown in the Figures 3.12. If a model is to the right of the curve, it means that another model has better performance for less computation time. As can be seen, there is a sharp increase in the boundary around a normalized computation time of 1, and then the boundary stabilizes. The AUC remains approximately constant at around 98 %. The AP still increases with computation time, but with a smaller incremental gain. There is a clear linear trend between the AUC and the AP, showing that an improvement in one metric tends to lead to an improvement in the other.

The model chosen for the final system was taken at the corner of this transition between sharp increase and stabilization. It is shown in green in Figure 3.12. Its parameters and performance are given in the Table 3.6. The memory requirements are negligible, taking only 10.42 % of the flash memory and 4.05 % of the RAM. Most of the flash memory used is consumed by the library necessary to run the model; the model weights are negligible in comparison. The execution time is different from the data given in Figure 3.12. This is because it has been measured for this specific model to avoid the approximations made using Eq. (3.4.4). We can see that the change is negligible, which shows that our estimates are accurate. The chosen model shows a good performance on the test set for a sensing duty cycle just below one.

Figure 3.13 shows how the execution time is distributed. As can be seen, the FFT is the dominant operation, taking up 46.4 % of the total computation. It is therefore the operation that should be prioritized for acceleration. The inference takes only 25.18 % of the total computation time. Compared to other models tested, it seems that better quality features are more important than model complexity. Figure 3.14 shows the distribution of

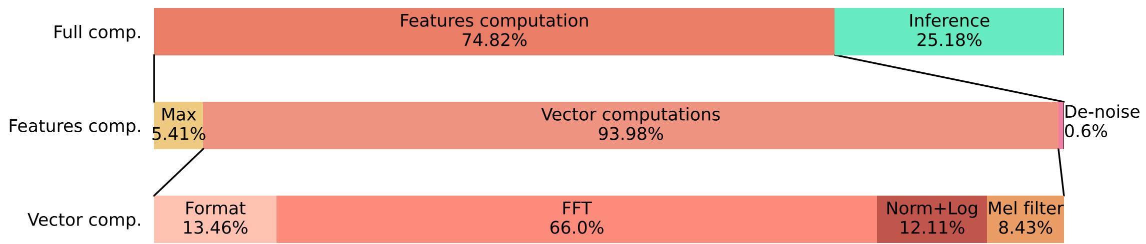


Figure 3.13: Distribution of time taken by each step of computation.

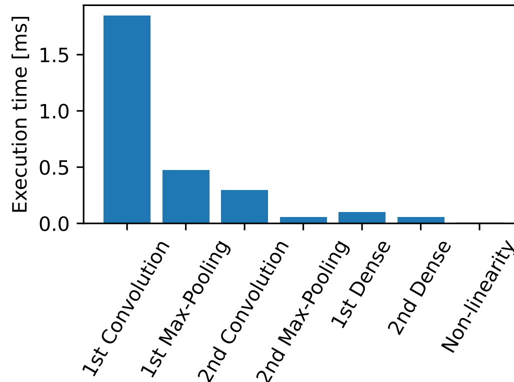


Figure 3.14: Execution time of the different layers of the chosen model.

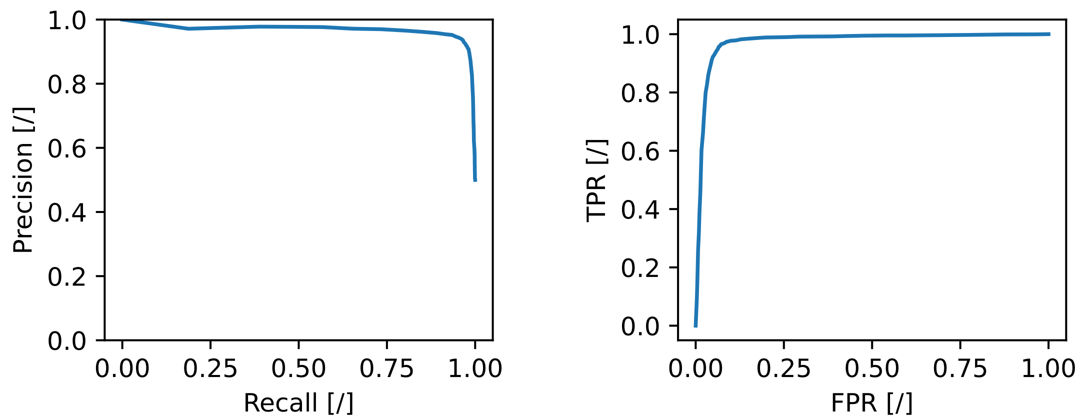
inference time across layers. As can be seen, the first convolutional layer dominates the execution time.

Given that we now have chosen our model, we can evaluate it on the test set. Table 3.7 gives the metrics obtained by the model. Looking at the case without approximation of the logarithm, we can see a slight loss of performance when used on the test set. This loss may be due to some overfitting, but it stays limited, being only a difference of 0.31 % of AUC and 0.6 % of AP. The quantization also leads to a negligible loss of performance. We see a loss associated with the inclusion of the logarithm approximation. This indicates that our approximation is not good enough to lead to a negligible difference in the features from the point of view of the model. This loss is a difference of 1.27 % of AUC and 0.35 % of AP when comparing the quantized models. The case with approximation of the logarithm shows no loss of performance when using the test set or when applying the quantization. This shows that the model is able to generalize to data it has never seen before.

Figure 3.15 shows the precision-recall curve and the ROC of the quantized model. We can observe that the system is not able to achieve a precision above about 98 % at any non-zero recall. Still, it is clear from those curves that the model is able to classify features accurately. From the precision-recall curve, we obtain that the recall at a precision of 95 % is 93.89 %.

	Train		Test		Post-quantization Test	
	AUC[%]	AP[%]	AUC[%]	AP[%]	AUC[%]	AP[%]
No log approximation	98.21	97.27	97.90	96.67	97.86	96.59
With log approximation	97.47	96.67	97.47	96.24	97.46	96.24

Table 3.7: Performance of the model in different situations.



(a) Precision-Recall curve

(b) ROC

Figure 3.15: Performance curves of the quantized chosen model.

Chapter 4

Characterization

4.1 Power consumption

4.1.1 Setup

A K2450 SourceMeter source measure unit from Keithley is used as a voltage generator and ammeter as shown in Figure 4.1. It supplies the system either at the input of the LDO or at its output. The only difference between the two nodes is the presence of the LDO, which stabilizes the voltage and consumes some power.

Three modes of the system are tested. The first is the Stop 1 mode and the second is the Standby mode. Both modes are described in Section 1.4. The last mode is the system when it is working. The MCU runs at 48 MHz, which is the state in which it consumes the most current. It should therefore be the main source of power consumption.

4.1.2 Results

Result of the measurements are reported in Table 4.1.

When attempting to measure the power consumption of writing to the microSD card, the system always crashed when either writing to the card or mounting the filesystem. This happened when the entire system was powered by the K2450, when only the microSD card was powered by the K2450, and when any ammeter was placed in the loop. The system seems to be too sensitive to work when a measurement system is placed in the supply chain.

4.1.3 Discussion

One of the objectives of this measure was to find the inconsistency in power consumption found during the initial characterization of the system [35]. Indeed, when using low power modes (stop modes and standby), the measured power consumption was higher than expected and no variation was observed with respect to the mode. This behaviour was observed when the debugger remained connected to the system.

When the debugger is removed, the measurements are as expected. In fact, if we compare the power consumption with the estimates made in Section 1.4, we see that the two are close. The system consumes slightly more power than typical consumption, but remains well below the maximum power consumption. From measurements in Stop 1 mode, we can see that the LDO consumes about 6 µA. The power consumption in standby measured at the input of the LDO should thus be about 115 µA

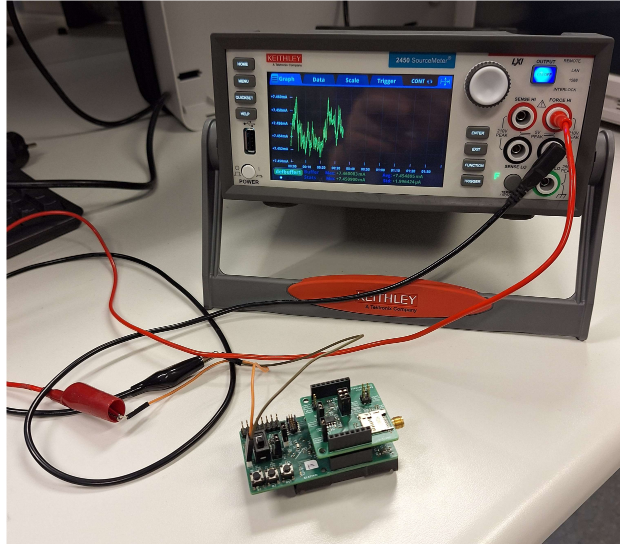


Figure 4.1: Measurement setup for the power consumption.

Measurement node	System mode	Voltage [V]	Current [mA]	Power [mW]
Input of LDO	Stop 1 mode with debugger connected	3.3	0.568	1.874
		4.5	0.573	2.579
Input of LDO	Stop 1 mode	3.3	0.118	0.389
		4.5	0.118	0.531
Output of LDO	Stop 1 mode	1.8	0.109	0.196
		3.3	0.112	0.370
Output of LDO	Standby	1.8	0.105	0.190
		3.3	0.109	0.360
Input of LDO	Computing	3.3	7.446	24.572
		4.5	7.453	33.538

Table 4.1: Result of power consumption measurement for different modes of the system.

With these results we can calculate the battery life. If we consider that we run the system for 8 hours a night, and that the system is on standby for the rest of the time, the energy consumed in a day is 61.46 mA h. This value is largely dominated by the 8 hours of execution, which account for 97 % of the consumption. Given that our system runs on 3 AAA batteries, and that they are alkaline, we have a total capacity between 3027 mA h and 3690 mA h[68]. This means that the system could run between 49.2 days and 60 days on the batteries.

If there was a need to increase this lifetime by any means other than increasing battery capacity, efforts would have to be made to reduce the MCU consumption during computing, as this dominates the overall power consumption. No effort has been made in this thesis to reduce power consumption.

If the device was powered at 4.5 V by an energy harvesting system (e.g. solar panels), taking into account sufficient storage to contain all the energy received during the day for use during the night, we would need to receive about 1 kJ of energy each day. It should be noted that the longer the night, the more difficult it is to meet the power consumption constraints, as there would be fewer hours in the day to receive energy and more hours to work at night.

4.2 Full system simulation

4.2.1 Setup

The simulation setup consists of recreating the data flow of the working system on a computer. The input audio stream used is the test file instead of the data sampled on the micro, and all operations are performed in Python. Feature generation is done in the same way as in training except for the time between features generated. Instead of being equal to the size of a single MSFB vector (S_{hop} in Table 3.4) as before, we now keep it as a variable called $S_{\text{spec,hop}}$. In practice, this variable would be set to the minimum value allowed by the computation time. In our case, this value is 3442, since the model takes about 11.47 ms for a full feature computation and inference. Henceforth, we will consider the value 3500 to add a small margin of safety.

The quantized model is run over all generated features. We then compute the TP, FP and FN from the data. For a given call detected by BatDetect2 with a probability greater than 0.6, if our model returns a positive result for a spectrogram whose start is less than 10 ms from the start or end of the call, we count it in the TP. If there is none, we count it in the FN. This value of 10 ms was the same as the one used in Bat Detective [21].

The FP was more complicated to calculate. A simple way was to say that if the model returned a positive on a given spectrogram and there was no call corresponding to it with the above 10 ms criterion, then it counted as a FP. However, on inspection of the data, some positives from the model did correspond to calls, but with a detection probability below 0.6 according to BatDetect2. There was a possibility that these were still calls that we had detected and that we were being pessimistic. To account for this, the following formula was used to calculate the value to add to FP:

$$s(\Pr(\text{detect})) = \begin{cases} 1 & \Pr(\text{detect}) \leq 0.4 \\ \frac{5}{2}(0.6 - \Pr(\text{detect})) & 0.4 \leq \Pr(\text{detect}) \leq 0.6 \\ 0 & 0.6 \leq \Pr(\text{detect}) \end{cases}, \tag{4.1}$$

where Pr(detect) is the maximum probability according to BatDetect2 of a call around the positive prediction. The formula adds 1 to FP if this probability is below 0.4, nothing if

it is above 0.6, and linearly interpolates between 0.5 and 0 if the probability is between 0.4 and 0.6. It is not continuous to ensure that it is a lower bound. The value of 0.5 was chosen by assuming that a call with a probability of 0.4 has a probability of 0.5 of being a real call. This is based on the fact that 0.4 is around the boundary between real calls and false positives of BatDetect2, as can be seen when comparing Figures 2.15 and 2.14.

This formula is arbitrary and aims to penalize the model less for returning a positive call when the probability of a call was between 0.4 and 0.6. However, it is designed to be particularly optimistic and should therefore underestimate the number of FP. The two methods were thus used and both reported as upper and lower bounds on the real performance.

The simulation is run with different values of the threshold to obtain precision recall curves, which are then used to compute the AP via an integration using a trapezoidal rule. Calculating the AUC would have required calculating the number of TN, which suffers from the same problems as counting the FP and would have introduced more uncertainty. Moreover, since we are in a highly unbalanced case, the AUC should be less informative about the true quality of our algorithm. The AUC was therefore not calculated.

4.2.2 Results

Figure 4.2 shows the precision-recall curve for all tested values of $S_{\text{spec,hop}}$. We can see that there is a loss of performance when increasing $S_{\text{spec,hop}}$. This is confirmed by Figure 4.3, which shows a clear maximum of AP at an $S_{\text{spec,hop}}$ of 875, followed by a decreasing trend. The fact that an $S_{\text{spec,hop}}$ of 270 is worse than an $S_{\text{spec,hop}}$ of 875 could be due to the increase in FP associated with an increased prediction rate. An $S_{\text{spec,hop}}$ of 850 means computing one prediction every 2.83 ms, which is obviously not possible with our system, which takes 11.47 ms per prediction.

The conclusion is that without any limit on the speed of prediction, the AP achievable by the system should be between 81.01 % and 91.22 %. However, taking into account the computational limitations of our system, this achievable AP falls between 60.29 % and 69.55 %.

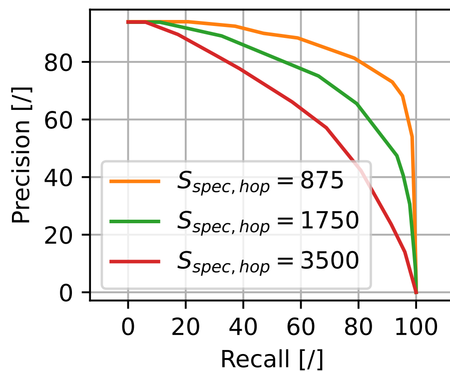


Figure 4.2: Precision-recall curves for different values of $S_{\text{spec,hop}}$. For clarity, average between higher and lower bound of the precision is used.

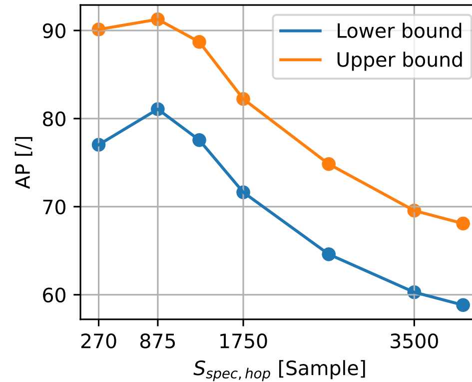


Figure 4.3: AP against $S_{\text{spec,hop}}$.

4.2.3 Discussion

This experiment shows that the rate at which prediction is crucial for call detection. In fact, even in the best case (using optimal $S_{\text{spec,hop}}$ and optimistic precision), we lose 5.02 % of AP when moving from the feature-based inference task to the call detection task. This loss can be explained by the strong imbalance between positives and negatives, which leads to more false positives.

Even under the time constraints imposed by the platform's processing power, the system showed clear bat detection capabilities.

We can also consider how our performance would improve with additional hardware. First, if CPUs were added to the platform, it would be possible to proportionally compute more predictions. With one additional CPU we would be able to achieve $S_{\text{spec,hop}} = 1750$ and with three additional CPUs we would be able to achieve $S_{\text{spec,hop}} = 850$, which is roughly the optimal value.

The STM32WLE5CCU MCU in the platform has a variant, the STM32WL55CC, with the same capabilities but with an additional Cortex-M0 core [69]. The Cortex-M0 core has slightly inferior performance in terms of computational speed, so it may not lead to an ideal parallelization of the algorithm. Considering the best case, where our speed is multiplied by 2, we would gain about 12 % of AP at little to no monetary cost, as the two MCU cost almost the same. There would still be an increase in power consumption due to the second CPU, which would reduce the amount of time the device could run without human intervention.

This gain of 12 % would also be approximately the gain observed when adding a hardware accelerator to compute the FFT, which was shown in Figure 3.13 to take almost half the time. If both the hardware accelerator for the FFT and the dual-core MCU were used, it would be possible to approach the ideal situation around $S_{\text{spec,hop}} = 850$.

4.3 Full system validation

4.3.1 Setup

Two platforms are used. The first is set up to continuously record the sound from the microphone to the microSD, with the external power supply, just like the dataset generation in Section 2.5.1.

The second platform continuously computes features and applies inference based on the data from the microphone. As it does not use the microSD card, there was no need to use an external power supply. When an inference returns a probability greater than 0.3, the system writes the current time to memory. 0.3 was chosen because it gave a good trade-off between recall and precision, while not predicting a positive result in the case where the features were only zeros, where a probability of around 0.26 is returned. The aim was to avoid false positives when the ambient sound was too weak.

The time is measured with the RTC of the MCU. The system is set up to print this list of times when a key is pressed, so we can save it. Time stamps are stored as 32-bit numbers, with the hour, minute, second and sub-second each taking up one byte. The sub-second is the unit of time based on the counter used by the RTC. In our case, the sub-second is encoded at 8 bits, giving a resolution of 3.9 ms.

$S_{\text{spec,hop}}$ is chosen to be 4000 samples, which gives a small margin to account for the timestamp saving mechanism mentioned above. This corresponds to one prediction every

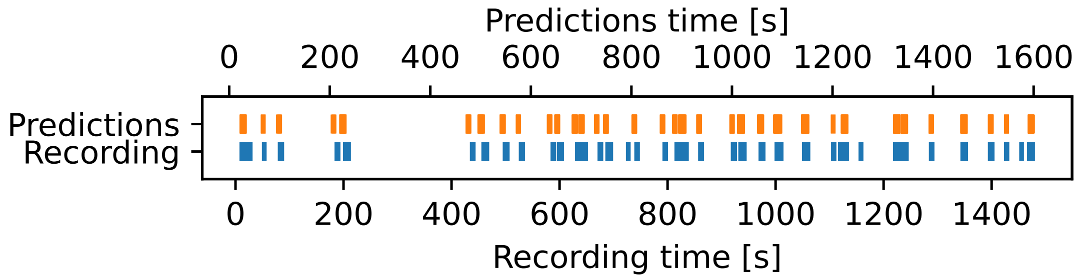


Figure 4.4: Timestamp of calls predicted in real time and calls detected with 0.6 probability in recording.

13.33 ms. The simulator described in section 4.2 predicts a precision between 41.14 % and 47.73 % and a recall of 77.78 %.

The site used is the one in Louvain-la-Neuve, where most of the training took place. The experiment was carried out on the 15th of May. The two platforms were placed side by side in the same setup as shown in Figure 2.12. As the two systems were side by side, we can assume that the data recorded on the microSD is approximately the data on which the other system based its inferences.

4.3.2 Results

Figure 4.4 shows the result of the experiment. The predictions are simply the list of timestamps printed by the system doing inferences in real time. The recording timestamps are obtained by running BatDetect2 on the recording with a threshold of 0.6 on the returned probability of call. Note that the axes are different. There are both a constant delay on the timestamps and a factor. The constant delay is introduced because the systems are not synchronized, i.e. they started operating at different times.

The reason for the factor is not exactly determined. On closer inspection of the data, however, it does not appear to be a simple factor, but rather a kind of jitter. Indeed, no choice of factor synchronizes all the calls simultaneously. The explanation found is the presence of leaps in the data written to the microSD card due to housekeeping, as mentioned in Section 2.5.2. This explains the observed jitter, as a delay is randomly introduced into the data, resulting in a shorter recording duration compared to reality. Based on the time difference between prediction and recording, approximately 6.72 % of the audio data is not recorded due to these data jumps. Unlike during the recording sessions, the microSD card used in this experiment was a 2 GB SD-C02G card. This is a low-cost card, and it therefore may have a higher latency due to housekeeping than the card used in the recording sessions. Using a higher-end card may eliminate this problem entirely, as the housekeeping may become negligible.

However, with the linear correction applied, we can clearly see that both system detect calls around the same time. This shows the functionality of the system. Some calls are visibly missed by the system, so it is clear that it has errors, which was expected according to the simulations. Due to the jitter, it was not possible to compute precision and recall, as it was not possible to accurately assign predictions to real calls without extensive manual correction of timestamps, rendering the metrics almost meaningless.

4.3.3 Discussion

The experiment was successful in validating the use of the system for bat detection. The system could therefore be used to monitor the bat population over time. Even if the system does not have perfect precision or recall, variations from year to year could provide information on the evolution of the bat population in the area. This could be achieved with the timestamp strategy used in this experiment. If timestamps are only stored in the RAM, this limits the number of calls recorded to around 18000 due to memory constraints. Another strategy, hereafter referred to as the wireless strategy, is to transmit detected bat calls via the LoRa radio. This reduces battery life and requires receivers to collect the data.

The choice of strategy can be based on the lifetime of the device without interaction. Given that 451 calls were collected in about half an hour during the experiment, and assuming that bat calls continue to occur at the same rate, we would fill the memory in about 20 hours using the timestamp strategy, which is equivalent to less than three full nights. Still considering the same number of calls per night, a spreading factor of 7, an output power of 14 dBm, and that the energy needed to send data scales linearly with the size of the data sent, the wireless strategy would consume 14.1 J per day. This is about 1 to 2 % of the total power consumption, which would result in a negligible loss of battery life.

Optimizations such as merging calls close to each other could improve the performance of the timestamp strategy, but are beyond the scope of this work. Writing timestamps to the microSD would make the memory constraint negligible against the performance constraint. The effect on battery life would also most likely be negligible, as it would consume 0.5 mJ per day to write the data to the microSD, assuming that there are 450 calls per half hour, a power consumption of 24 mW for the microSD card, as well as the throughput of 1.28 MB/s obtained in Section 2.2.1.

It should be noted that the wireless strategy may be less practical for small-scale applications than the timestamp strategy, which results in a stand-alone system. Indeed, it requires the installation of other equipment, either through a network of sensors or a base station to receive all the packets. However, it has the advantage of providing daily data without manual interaction with the system. With the addition of an energy harvesting system, the wireless solution could become completely unmanned except for maintenance.

There appears to be no positive prediction of calls when no bat is present. This means that a strategy based on recording a clip of one second when a call is detected would capture most calls while avoiding recording audio without useful data. This would be particularly useful for creating datasets, as it would minimize memory consumption, the current bottleneck, and guarantee that the 16 hours of sound the system can store are actually bat calls and not mostly empty data.

Conclusion

In this thesis, a bat detection algorithm is designed and implemented to run on a low-cost platform originally designed for fire detection. The main objective was to test the feasibility of the task and determine how hardware limitations translate into loss of performance. The final system shows promise for the use of such low-cost devices to aid biodiversity surveys.

A dataset was recorded at Louvain-la-Neuve using the device. Necessary modifications were made to the platform to make this dataset collection feasible. The first one is an optimization of the driver to store data on the microSD card, which increases the throughput by more than an order of magnitude. The second change is the addition of an external power supply to the microSD card. Its role is to eliminate the supply noise caused by writing to the microSD card, which caused noise spikes. These could be mistaken for bat calls, so it was necessary to eliminate them. The external supply solution proved successful in this respect and was therefore validated.

More than 10 000 calls were collected in just two nights. The annotation of the data is done by BatDetect2 [30], a machine learning algorithm developed by Mac Aodha et al. in 2022. The use of this different model allowed for easier processing of the data, which would normally have to be done by a long and tedious process of manual annotation. However, it introduces uncertainty into the evaluation of the model, as there are no hand annotated data available for our system.

The design of the algorithm is mainly inspired by Bat Detective [21], which led to the choice of a CNN for the machine learning model. However, several modifications are made to take into account the limitations of our hardware, the most important of which is the use of MSFBs to reduce the number of dimensions of the data. This allows a significant speed-up in the execution of the machine learning model.

The implementation of the pipeline to generate the features is done efficiently using SIMD and loop unrolling as much as possible. A piecewise linear approximation of the logarithm is proposed, with an execution more than 20 times faster than its equivalent from the CMSIS library. This approximation affects the result with a loss of 1.27 % of AUC and 0.35 % of AP, but this loss is not enough to offset the speed-up obtained. The machine learning model is quantized to 8-bit fixed-point format to speed up execution and reduce memory requirements.

After optimization of the hyperparameters, an optimized algorithm is obtained. The complete algorithm takes 11.47 ms to execute on the platform and takes 9.8 ms of audio data as input. The solution does not achieve a perfect sensing duty cycle. A simulation of the system in operation is performed, which shows that the AP of the algorithm for the bat call detection task is between 60.29 % and 69.55 %. The simulation shows that relaxing the execution speed constraints would allow a solution with an AP between 81.01 % and 91.22 %. Two hardware changes are proposed to approach this optimum performance. The first is to switch to a four-core MCU. The second is to use a two-core MCU and add a hardware accelerator to compute the FFT. Both of these suggestions would approach the

speed-up required for the optimal prediction rate.

The power consumption of the device has been measured and shows a consumption of approximately 1 kJ per night of operation. Using 3 AAA alkaline batteries, this gives a battery life between 49.2 and 60 days.

The system was validated in the field by running it while recording the ambient noise. This clearly demonstrated the system's ability to detect bat calls. However, precise performance metrics could not be obtained, most likely due to recording errors.

Two applications of the system are suggested. The first is to record a dataset of bat calls. Detecting bat calls before starting recording allows a net gain in recording time by avoiding storing audio data without a call. In this mode the system is able to record just under 16 hours of bat calls on a microSD card of 32 GB. The second task considered for the system is to monitor bat populations over time. Two modes of operation are possible, based on the storage or transmission of timestamps of detected calls. The first option results in a stand-alone device where data has to be collected manually. The second option requires investment in other equipment to make the connection, but allows automatic data collection. If energy harvesting is used, wireless operation could even lead to a completely unmanned system, except for maintenance.

The methodology used in this work could be applied to most other tasks where machine learning has been shown to produce good results. One direct extension that can be made to the proposed algorithm is to take bat species into account. The classical way to do this would be to try to classify bat calls over a large set of expected species as they are recorded, and add this information to the stored timestamp. Another way would be to classify only a precise species to filter the recording when creating a dataset. This would be particularly useful when creating a dataset of specific bat species to filter out common pipistrelle calls due to their high frequency in the recordings. In both cases, the algorithm would likely be more complex, both in terms of feature generation and the model used, to match the complexity of the task. Their implementation should therefore probably be accompanied by hardware improvements, possibly taking into account the design recommendations suggested here.

Bibliography

[1] DeepL. *Better writing with DeepL Write*. URL: https://www.deepl.com/write. (accessed: 28.05.2024).

[2] Edward Osborne Wilson. *Biodiversity*. National Acad. Press, 1988.

[3] "Post-2020 biodiversity targets need to embrace climate change". In: *Proceedings of the National Academy of Sciences of the United States of America* 117.49 (2020), pp. 30882–30891. DOI: 10.1073/PNAS.2009584117.

[4] IPBES. *Summary for policymakers of the global assessment report on biodiversity and ecosystem services*. Version summary for policy makers. Nov. 2019. DOI: 10.5281/zenodo.3553579. URL: https://doi.org/10.5281/zenodo.3553579.

[5] Katherine Richardson et al. "Earth beyond six of nine planetary boundaries". In: *Science Advances* 9.37 (2023), eadh2458. DOI: 10.1126/sciadv.adh2458. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.adh2458. URL: https://www.science.org/doi/abs/10.1126/sciadv.adh2458.

[6] IUCN Biodiversity Assessment & Knowledge Team Red List Unit. *The IUCN red list of threatened species*. URL: www.iucnredlist.org. (accessed: 12.03.2024).

[7] Danilo Russo et al. "Do We Need to Use Bats as Bioindicators?" In: *Biology* 10.8 (2021). ISSN: 2079-7737. DOI: 10.3390/biology10080693. URL: https://www.mdpi.com/2079-7737/10/8/693.

[8] ADAM S. LEWANDOWSKI, REED F. NOSS, and DAVID R. PARSONS. "The Effectiveness of Surrogate Taxa for the Representation of Biodiversity". In: *Conservation Biology* 24.5 (2010), pp. 1367–1377. DOI: https://doi.org/10.1111/j.1523-1739.2010.01513.x. eprint: https://conbio.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1523-1739.2010.01513.x. URL: https://conbio.onlinelibrary.wiley.com/doi/abs/10.1111/j.1523-1739.2010.01513.x.

[9] Carles Flaquer, Ignacio Torre, and Antoni Arrizabalaga. "Comparison of Sampling Methods for Inventory of Bat Communities". In: *Journal of Mammalogy* 88.2 (Apr. 2007), pp. 526–533. ISSN: 0022-2372. DOI: 10.1644/06-MAMM-A-135R1.1. eprint: https://academic.oup.com/jmammal/article-pdf/88/2/526/2500764/88-2-526.pdf. URL: https://doi.org/10.1644/06-MAMM-A-135R1.1.

[10] INGEMAR Ahlén. "Heterodyne and time-expansion methods for identification of bats in the field and through sound analysis". In: *Bat Echolocation Research: tools, techniques and analysis* (2004), p. 72.

[11] SonoBat. *Zero-crossing*. URL: https://sonobat.com/zero-crossing/. (accessed: 23.03.2024).

[12] Titley scientific. *Current product range comparison*. 2023. URL: https://www.titley-scientific.com/wp-content/uploads/2023/11/Titley-Scientific-Current-Product-Range-Comparison-2023.pdf. (accessed: 24.03.2024).

[13] SonoBat. *SonoBat Classification. Tracking, recording, learning, training & innovating since 1991.* URL: https://sonobat.com/sonobat_classification. (accessed: 23.03.2024).

[14] Adrian T. Ruiz et al. "Detection of Bat Acoustics Signals Using Voice Activity Detection Techniques with Random Forests Classification". In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications.* Ed. by Ruben Vera-Rodriguez, Julian Fierrez, and Aythami Morales. Cham: Springer International Publishing, 2019, pp. 253–261. ISBN: 978-3-030-13469-3.

[15] Peter Prince et al. "Deploying Acoustic Detection Algorithms on Low-Cost, Open-Source Acoustic Sensors for Environmental Monitoring". In: *Sensors* 19.3 (2019). ISSN: 1424-8220. DOI: 10.3390/s19030553. URL: https://www.mdpi.com/1424-8220/19/3/553.

[16] D. O'Shaughnessy. *Speech Communication: Human and Machine.* Addison-Wesley series in electrical engineering. Addison-Wesley Publishing Company, 1987. ISBN: 9780201165203.

[17] Imran Zualkernan et al. "A Tiny CNN Architecture for Identifying Bat Species from Echolocation Calls". In: *2020 IEEE / ITU International Conference on Artificial Intelligence for Good (AI4G).* 2020, pp. 81–86. DOI: 10.1109/AI4G50087.2020.9311084.

[18] Barataud M. 2015. *Écologie acoustique des Chiroptères d'Europe: Identification des espèces, études de leurs habitats et comportements de chasse.* 3e éd. Biotope, Mèze;Muséum national d'Histoire naturelle, Paris (collection Inventaires et biodiversité), p. 344.

[19] Raghuram H., Manjari Jain, and R Balakrishnan. "Species and acoustic diversity of bats in a palaeotropical wet evergreen forest in southern India". In: *Current science* 107 (Aug. 2014), pp. 631–641.

[20] Mark Skowronski and John Harris. "Acoustic detection and classification of microchiroptera using machine learning: Lessons learned from automatic speech recognition". In: *The Journal of the Acoustical Society of America* 119 (Apr. 2006), pp. 1817–33. DOI: 10.1121/1.2166948.

[21] Oisin Mac Aodha et al. "Bat Detective - Deep Learning Tools for Bat Acoustic Signal Detection". In: 2018.

[22] Keigo Kobayashi et al. "Development of a species identification system of Japanese bats from echolocation calls using convolutional neural networks". In: *Ecological Informatics* 62 (2021), p. 101253. ISSN: 1574-9541. DOI: https://doi.org/10.1016/j.ecoinf.2021.101253. URL: https://www.sciencedirect.com/science/article/pii/S1574954121000443.

[23] Imran Zualkernan et al. "An AIoT System for Bat Species Classification". In: *2020 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS).* 2021, pp. 155–160. DOI: 10.1109/IoTaIS50849.2021.9359704.

[24] Maxime Franco and Corrado Lipani. "Automated monitoring of bat species in Belgium". Université catholique de Louvain, 2020. Prom. : Bonaventure, Olivier ; Nijssen, Siegfried.

[25] Michael A. Tabak et al. "Automated classification of bat echolocation call recordings with artificial intelligence". In: *Ecological Informatics* 68 (2022), p. 101526. ISSN: 1574-9541. DOI: https://doi.org/10.1016/j.ecoinf.2021.101526. URL: https://www.sciencedirect.com/science/article/pii/S1574954121003174.

[26] Mélanie Beauvois and Lucile Dierckx. "Automated detection of bat species in Belgium". Université catholique de Louvain, 2021. Prom. : Bonaventure, Olivier ; Nijssen, Siegfried.

[27] Nathan De Winter. "Binarized neural networks: bat call classification". Université catholique de Louvain, 2022. Prom. : Nijssen, Siegfried.

[28] Hicham Bellafkir et al. "Bat Echolocation Call Detection and Species Recognition by Transformers with Self-attention". In: *Intelligent Systems and Pattern Recognition.* Ed. by Akram Bennour et al. Cham: Springer International Publishing, 2022, pp. 189–203. ISBN: 978-3-031-08277-1.

[29] E Schwab et al. "Automated Bat Call Classification using Deep Convolutional Neural Networks". In: (Apr. 2021).

[30] Oisin Mac Aodha et al. "Towards a General Approach for Bat Echolocation Detection and Classification". In: *bioRxiv* (2022).

[31] Google. *Machine Learning Glossary. convolutional neural network.* URL: https://developers.google.com/machine-learning/glossary#convolutional_neural_network. (accessed: 08.04.2024).

[32] Raghubir Singh and Sukhpal Singh Gill. "Edge AI: A survey". In: *Internet of Things and Cyber-Physical Systems* 3 (2023), pp. 71–92. ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.02.004. URL: https://www.sciencedirect.com/science/article/pii/S2667345223000196.

[33] Youssef Abadade et al. "A Comprehensive Survey on TinyML". In: *IEEE Access* 11 (2023), pp. 96892–96922. DOI: 10.1109/ACCESS.2023.3294111.

[34] Google. *Edge TPU.* URL: https://cloud.google.com/edge-tpu. (accessed: 17.05.2024).

[35] Nicolas Brusselmans. "Design and characterisation of a wireless sensing node for natural-environment monitoring". Université catholique de Louvain, 2023. Prom. : Bol, David ; Louveaux, Jérôme.

[36] *Multiprotocol LPWAN 32-bit Arm® Cortex®-M4 MCUs, LoRa®, (G)FSK, (G)MSK, BPSK, up to 256KB flash, 64KB SRAM.* DS13105. Rev. 12. STMicroelectronics. Dec. 2022.

[37] Thomas Lorenser. "The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors. DSP feature set and benchmarks". In: (Nov. 2016), p. 7.

[38] CMSIS. *CMSIS Version 6.1.0.* URL: https://arm-software.github.io/CMSIS_6/v6.1.0/General/index.html. (accessed: 28.05.2024).

[39] *TPS7A03 Nanopower IQ, 200-nA, 200-mA, Low-Dropout Voltage Regulator With Fast Transient Response.* SBVS375C. Rev. September 2022. Texas Instruments. July 2019.

[40] *BGS12SN6. Wideband RF SPDT Switch in small package with 0.77mm2 footprint.* Rev. 2.3. Infineon. Sept. 2016.

[41] *BME680 - Datasheet. Low power gas, pressure, temperature & humidity sensor.* Rev. 1.9. Bosch. Feb. 2024.

[42] *SPU0410LR5H-QB. Zero-Height SiSonic Microphone*. Rev. H. Knowles Electronics. Mar. 2013.

[43] ChaN. *How to Use MMC/SDC*. URL: http://elm-chan.org/docs/mmc/mmc_e.html. (accessed: 28.04.2024).

[44] kiwih. *Tutorial: An SD card over SPI using STM32CubeIDE and FatFS*. Aug. 2020. URL: https://01001000.xyz/2020-08-09-Tutorial-STM32CubeIDE-SD-card/. (accessed: 19.03.2024).

[45] Dr. Gough Lui. *Experiment: microSD Card Power Consumption & SPI Performance*. Feb. 2021. URL: https://goughlui.com/2021/02/27/experiment-microsd-card-power-consumption-spi-performance/. (accessed: 18.05.2024).

[46] *TLVx316. 10-MHz, Rail-to-Rail Input/Output, Low-Voltage, 1.8-V CMOS Operational Amplifiers*. Rev. SEPTEMBER 2016. Texas Instruments. Feb. 2016.

[47] Nick Patavalis. *picocom*. URL: https://github.com/npat-efault/picocom. (accessed: 27.05.2024).

[48] Oliver Laumann et al. *screen(1) - Linux man page*. URL: https://linux.die.net/man/1/screen. (accessed: 27.05.2024).

[49] Miquel van Smoorenburg and Jukka Lahtinen. *minicom(1) - Linux man page*. URL: https://linux.die.net/man/1/screen. (accessed: 27.05.2024).

[50] *STLINK-V3MINIE debugger/programmer tiny probe for STM32 microcontrollers*. UM2910. Rev. 3. STMicroelectronics. Mar. 2024.

[51] Tom Hoag. *nms documentation*. 2018. URL: https://nms.readthedocs.io/en/latest/index.html. (accessed: 04.04.2024).

[52] Service Public de Wallonie. *Les chauves-souris de Belgique*. URL: http://biodiversite.wallonie.be/fr/nos-especes.html?IDC=5579. (accessed: 11.05.2024).

[53] Paul Rubin, David MacKenzie, and Stuart Kemp. *dd(1) - Linux man page*. URL: https://linux.die.net/man/1/dd. (accessed: 28.05.2024).

[54] Binary Acoustic Technology. *SCANR Software. Product details*. URL: https://binaryacoustictech.com/batpages_files/scanr.htm. (accessed: 17.05.2024).

[55] Wildlife Acoustics. *Kaleidoscope Pro Analysis Software*. URL: https://www.wildlifeacoustics.com/products/kaleidoscope-pro. (accessed: 17.05.2024).

[56] Kate E. Jones et al. "Indicator Bats Program: A System for the Global Acoustic Monitoring of Bats". In: *Biodiversity Monitoring and Conservation*. John Wiley & Sons, Ltd, 2013. Chap. 10, pp. 211–247. ISBN: 9781118490747. DOI: https://doi.org/10.1002/9781118490747.ch10. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118490747.ch10. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118490747.ch10.

[57] Stuart E. Newson, Hazel E. Evans, and Simon Gillings. "A novel citizen science approach for large-scale standardised monitoring of bat activity and distribution, evaluated in eastern England". In: *Biological Conservation* 191 (2015), pp. 38–49. ISSN: 0006-3207. DOI: https://doi.org/10.1016/j.biocon.2015.06.009. URL: https://www.sciencedirect.com/science/article/pii/S0006320715002323.

[58] NVidia. *XGBoost*. URL: https://www.nvidia.com/en-us/glossary/xgboost/. (accessed: 23.05.2024).

[59] Raspberry Pi. *Raspberry Pi 3 Model B*. URL: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/. (accessed: 17.05.2024).

[60] Raspberry Pi. *Raspberry Pi 400 unit.* URL: https://www.raspberrypi.com/products/raspberry-pi-400-unit/. (accessed: 17.05.2024).

[61] Siegfried Gündert. *PyFilterbank documentation. Mel Filter Bank.* 2014. URL: https://siggigue.github.io/pyfilterbank/melbank.html. (accessed: 05.05.2024).

[62] TensorFlow. *Model conversion overview.* URL: https://www.tensorflow.org/lite/models/convert/. (accessed: 04.05.2024).

[63] *X-CUBE-AI Data brief. Artificial intelligence (AI) software expansion for STM32Cube.* DB3788. Rev. 10. STMicroelectronics. Feb. 2023.

[64] Wikipedia. *Template:Diagnostic testing diagram — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Template%3ADiagnostic%20testing%20diagram&oldid=1213926654. [Online; accessed 09-April-2024]. 2024.

[65] Takaya Saito and Marc Rehmsmeier. "The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets". In: *PloS one* 10 (Mar. 2015), e0118432. DOI: 10.1371/journal.pone.0118432.

[66] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG].

[67] Usha Ruby and Vamsidhar Yendapalli. "Binary cross entropy with deep learning technique for image classification". In: *Int. J. Adv. Trends Comput. Sci. Eng* 9.10 (2020).

[68] *Data Sheet High Energy 4903.* VARTA Consumer Batteries GmbH & Co. KGaA. June 2005.

[69] *Multiprotocol LPWAN dual core 32-bit Arm® Cortex®-M4/M0+ LoRa®, (G)FSK, (G)MSK, BPSK, up to 256KB flash, 64KB SRAM.* DS13293. Rev. 5. STMicroelectronics. Dec. 2022.

Appendix A

Data of the algorithm optimization

Table A.1: Full data from the model optimization. Chosen model in green.

# filt 1	# filt 2	N_{vec} [/]	N_{msfb} [/]	S_{fft} [S]	S_{hop} [S]	f_s [kHz]	f_H [kHz]	S_{feat} [S]	T_{comp} [ms]	$\frac{T_{\text{comp}}}{S_{\text{feat}}}$ [/]	AUC [%]	AP [%]
4	4	22	28	256	82	300	120	1974	19.2	2.92	97.73	96.87
4	4	22	28	256	95	300	120	2261	19.2	2.55	98.00	97.35
4	4	22	28	256	109	300	120	2547	19.3	2.27	97.99	97.30
4	4	22	28	256	123	300	120	2833	19.3	2.05	97.82	96.99
8	8	12	20	512	175	300	120	2437	14.6	1.80	97.94	97.06
8	8	12	20	512	200	300	120	2712	14.7	1.62	98.01	97.25
4	4	15	15	256	120	300	120	1936	10.3	1.60	96.76	96.10
4	4	10	28	256	180	300	120	1876	9.2	1.48	97.42	96.56
8	8	12	20	512	225	300	120	2987	14.7	1.48	97.96	97.02
4	4	15	15	256	140	300	120	2216	10.3	1.40	96.81	96.35
8	8	12	20	512	250	300	120	3262	14.8	1.36	97.37	96.66
4	4	10	22	256	180	300	120	1876	8.2	1.32	97.52	96.78
8	4	12	20	512	250	300	120	3262	14.3	1.32	97.58	96.31
4	4	12	15	256	150	300	120	1906	8.4	1.31	96.46	95.88
4	4	10	28	256	210	300	120	2146	9.3	1.30	97.57	96.88
4	4	10	21	256	180	300	120	1876	8.1	1.29	97.39	96.49
4	4	10	20	512	240	300	120	2672	10.7	1.21	97.98	97.19
4	4	10	28	256	240	300	120	2416	9.3	1.16	97.44	96.74
4	4	10	22	256	210	300	120	2146	8.3	1.16	97.53	96.86
4	4	10	21	256	210	300	120	2146	8.1	1.14	97.57	96.91
4	4	10	20	256	210	300	120	2146	8.0	1.12	96.65	96.27
4	4	10	20	512	160	200	90	2928	10.8	1.10	97.56	96.60
4	4	10	20	512	270	300	120	2942	10.8	1.10	97.98	97.08

Continued on next page

Table A.1: Full data from the model optimization. Chosen model in green. (Continued)

4	4	10	20	256	140	200	90	2274	8.0	1.06	96.22	95.86
4	4	10	28	256	270	300	120	2700	9.4	1.04	97.34	96.55
4	4	10	22	256	240	300	120	2416	8.3	1.03	97.44	96.84
4	4	12	15	256	175	300	120	2181	7.5	1.03	96.46	96.04
4	4	10	21	256	240	300	120	2416	8.2	1.01	97.40	96.90
4	4	10	20	512	300	300	120	3212	10.8	1.01	97.74	96.66
4	4	10	20	256	160	200	90	2544	8.1	0.95	96.35	96.01
4	4	10	28	256	300	300	120	3000	9.4	0.94	97.16	96.20
4	4	10	22	256	270	300	120	2700	8.4	0.93	97.25	96.45
4	4	10	21	256	270	300	120	2700	8.2	0.91	97.22	96.34
4	4	10	10	256	210	300	120	2146	6.3	0.88	96.53	96.17
4	4	10	22	256	300	300	120	3000	8.4	0.84	97.12	96.28
4	4	10	21	256	300	300	120	3000	8.3	0.83	97.10	96.17

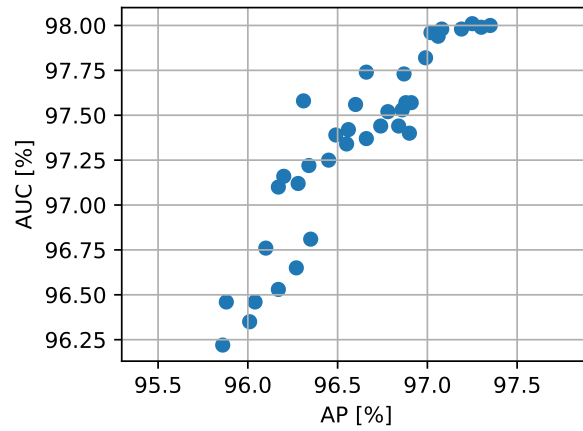


Figure A.1: AUC against AP on measured models.

# Filter 1	# Filter2	Width	Height	Time [ms]							
				Conv2D 1	Pool	Conv2D 2	Pool	Dense 1	Dense 2	NL	Total
4	4	22	28	6,325	1,654	1,462	0,269	0,207	0,054	0,006	9,976
4	4	10	10	0,837	0,218	0,128	0,027	0,081	0,056	0,006	1,352
4	4	10	20	1,847	0,472	0,295	0,054	0,099	0,056	0,006	2,835
4	4	10	28	2,591	0,676	0,427	0,081	0,115	0,054	0,006	3,949
4	4	10	22	2,007	0,523	0,328	0,067	0,099	0,054	0,006	3,083
4	4	10	21	1,951	0,473	0,294	0,054	0,099	0,054	0,006	2,937
4	4	12	21	2,363	0,585	0,41	0,054	0,099	0,052	0,006	3,569
4	4	12	15	1,634	0,395	0,258	0,041	0,091	0,054	0,006	2,477
4	4	15	15	2,109	0,47	0,321	0,066	0,097	0,056	0,006	3,124
4	4	12	20	2,241	0,585	0,41	0,054	0,099	0,056	0,006	3,451
8	4	12	20	3,112	1,106	0,407	0,054	0,099	0,056	0,006	4,838
8	8	12	20	3,113	1,107	0,788	0,089	0,119	0,054	0,006	5,274

Table A.2: Tested models execution time.

Appendix B

Code of the pipeline for generation of the features

```
q7_t melspec [N_MEL_BIN*N_MELVEC];
q7_t* melvec[N_MELVEC] = {melspec, melspec + N_MEL_BIN, melspec + 2 * N_MEL_BIN,
                          melspec + 3 * N_MEL_BIN, melspec + 4 * N_MEL_BIN,
                          melspec + 5 * N_MEL_BIN, melspec + 6 * N_MEL_BIN,
                          melspec + 7 * N_MEL_BIN, melspec + 8 * N_MEL_BIN,
                          melspec + 9 * N_MEL_BIN};

void treat_spec(q15_t* audio_data){

    //Find max shift without overflow
    q15_t max_value;
    uint32_t trash;
    arm_max_q15(audio_data, (HOP_LENGTH*(N_MELVEC-1) + SAMPLE_PER_MELVEC),
                &max_value, &trash);
    max_value = 16-log2_q15(max_value);

    //Compute each vector of MSFB
#pragma GCC unroll 10
    for (int i = 0 ; i < N_MELVEC ; i++){
        vec_computation(audio_data+HOP_LENGTH*i, melvec[i],max_value);
    }

    //De-noising + linear calibration for quantized model
#pragma GCC unroll 20
    for (int i = 0 ; i < N_MEL_BIN ; i++){
        int16_t mean = 0;
#pragma GCC unroll 10
        for (int j = 0 ; j < N_MELVEC ; j++) mean += melvec[j][i];
        mean /= N_MELVEC;
#pragma GCC unroll 10
        for (int j = 0 ; j < N_MELVEC ; j++)
            melvec[j][i] = (int8_t)(((int16_t)melvec[j][i]-mean)*2-99);
    }
```

```
    //Compute inference based on features
    MX_X_CUBE_AI_Process();

}


static const uint8_t log2_q15(q15_t x){
    //Return ceiling of log_2 of x
    //Based on
    //https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogDeBruijn

    static const uint8_t MultiplyDeBruijnBitPosition[32] =
    {
        1, 10, 2, 11, 14, 22, 3, 30, 12, 15, 17, 19, 23, 26, 4, 31,
        9, 13, 21, 29, 16, 18, 25, 8, 20, 28, 24, 7, 27, 6, 5, 32
    };
    x |= x >> 1; // first round down to one less than a power of 2
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;

    return MultiplyDeBruijnBitPosition[(uint32_t)(x * 0x07C4ACDDU) >> 27];
}


static const  q15_t bin_search_log(q15_t x) {
    //Approximation of 15*2^11+np.log2((1+x)/2^15) * 2^11
    //by 8 segment piecewise linear function
    if(x<2000){ if(x<290){ if(x<37){
                return x*312;
            }else{
                return x*24+10656;
        }}else{ if(x<840){
                return x*5+16166;
            }else{
                return x*2+18686;
    }}}else{if(x<8200){if(x<4050){
                return x+20686;
            }else{
                return (x>>1)+22711;
            }}else{if(x<14904){
                return (x>>2)+24761;
            }else{
                return (x>>3)+26624;
}}}}
```

```c
void vec_computation(q15_t* in, q7_t* out, uint8_t shift_val) {

    q15_t buf [ SAMPLE_PER_MELVEC ];

    uint32_t* word_in  = (uint32_t*) in;
    uint32_t* word_ham = (uint32_t*) hamming_window;
    uint32_t* word_buf = (uint32_t*) buf;

    //Center data, scale data and multiply by hamming window
#pragma GCC unroll 8
    for(int i = 0 ; i < SAMPLE_PER_MELVEC/2 ; i++){
        int32_t normalizeds = (__SSUB16(word_in[i], 0x08000800)<<shift_val)&0xFFF0FFF0;
        q31_t facs = word_ham[i];
        q15_t term1 = (((normalizeds >> 16) *  (facs >> 16)))>>15;
        q15_t term2 = (((q15_t) normalizeds * (q15_t) facs))>>15;
        word_buf[i] = __PKHBT(term2, term1 , 16);
    }

    q15_t fft_buf [ 2 * SAMPLE_PER_MELVEC ];
    uint32_t* word_fft_buf = (uint32_t*) fft_buf;

    //Compute FFT of the window
    arm_rfft_instance_q15 rfft_inst;
    arm_rfft_init_q15(&rfft_inst, SAMPLE_PER_MELVEC, 0, 1);
    arm_rfft_q15( &rfft_inst, buf, fft_buf);

    //Complex norm and application of logarithm
    for(int i = MIN_INDEX_MATMUL ; i < MAX_INDEX_MATMUL ; i+=4){
        uint32_t shifted = (word_fft_buf[i]<<1)&0xFFFEFFFE;
        buf[i]   = bin_search_log(__SMUAD(shifted, shifted)>>16);
        shifted  = (word_fft_buf[i+1]<<1)&0xFFFEFFFE;
        buf[i+1] = bin_search_log(__SMUAD(shifted, shifted)>>16);
        shifted  = (word_fft_buf[i+2]<<1)&0xFFFEFFFE;
        buf[i+2] = bin_search_log(__SMUAD(shifted, shifted)>>16);
        shifted  = (word_fft_buf[i+3]<<1)&0xFFFEFFFE;
        buf[i+3] = bin_search_log(__SMUAD(shifted, shifted)>>16);
    }

    //Mel-Scaled filtering
    q63_t temp_val = 0;
#pragma GCC unroll 20
    for(int i = 0 ; i < N_MEL_BIN ; i++){
        arm_dot_prod_q15(buf+s_pos[i],ls[i],l_lens[i], &temp_val);
        out[i] = (q7_t)(temp_val>>26);
    }
}
```

Code B.1: Embedded implementation of the pipeline.

Appendix C

Emulation of the pipeline for generation of the features

```
import numpy as np
import pyfilterbank as pf

def true_approx_log(x):
    if(x<2000):
        if(x<290):
            if(x<37):
                return x*312
            else:
                return x*24+10656
        else:
            if(x<840):
                return x*5+16166
            else:
                return x*2+18686
    else:
        if(x<8200):
            if(x<4050):
                return x+20686
            else:
                return (x>>1)+22711
        else:
            if(x<14904):
                return (x>>2)+24761
            else:
                return (x>>3)+26624

def vec_approx_log(vec, model):
    return np.vectorize(lambda x: true_approx_log(x))(vec)

def treat_spec(audio_vecs, fs, n_mel_bin):
    """
    audio_vecs is considered already formated
    (remove mean + scaling)
```

```
"""

audio_vecs = audio_vecs.astype(np.int16)

#multiply by Hamming window
quant_hamm = (np.hamming(len(audio_vecs[0]))*2**15).astype(np.int16)
audio_vecs = audio_vecs.astype(np.int32)*quant_hamm.astype(np.int32)
audio_vecs //= 2**15
audio_vecs = audio_vecs.astype(np.int16)

#fft
vecs = np.fft.rfft(audio_vecs, axis = 1)
vecs /= 2**9  #cmsis overflow avoidance

revecs = vecs.real.astype(np.int16)
imvecs = vecs.imag.astype(np.int16)
vecs = (revecs + 1j*imvecs)

vecs *= 2   #we can use some more bits
vecs = np.abs(vecs)**2
vecs /= 2**15   #q15_t re-normalisation
vecs /= 2  #cmsis overflow avoidance

vecs = vecs.astype(np.int16)

vecs = vec_approx_log(vecs, model).astype(np.int16)

band = [10e3, 120e3]

mmat, tup = pf.melbank.compute_melmat(num_mel_bands=n_mel_bin,
                freq_min=band[0], freq_max=band[1],
                num_fft_bands=len(vecs[0]), sample_rate=fs)

mmat = (mmat*2**15).astype(np.int16)
spec = np.dot(mmat.astype(float), vecs.T.astype(float)).T

spec //= 2**26
spec = spec.astype(np.int8)

#denoize like it would be done on the device
spec -= np.mean(spec, axis = 0, dtype=np.int8)[None, ...]

#put the features in the right format
spec = np.transpose(spec, (1, 0))[:, ::-1]
return spec
```

Code C.1: Python emulation of the pipeline.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**