Politenico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

Advanced Operating Systems: Project Report

Embedded Sound Classifier

Authors:
Marina Nikolic
Michele Scuttari

Supervisor:
Dr. Federico Terraneo

September 2019

POLITECNICO
MILANO 1863

Abstract

Implementation of a sound classifier on a STM32 board.

1 Introduction

1.1 The problem to solve

The goal of this project is the recording of audio samples through the usage of a STM32 board and their subsequent classification (into whistles and hands claps) by a neural network deployed on the aforementioned board.

1.2 Why neural networks?

For recognition and analysis of sound, AI techinques are used because of the complexity of the computations and the amount of noise present in the environment. Another important issue is that instances of the same sound have high variability due to different (yet homogeneous) sources. For example, think about words recognition: an effective application should recognise a word even if spoken by different people.

1.2.1 Which kind of neural network?

The neural network used in this project is a sequential feed-forward one. Since the aim is to distinguish two different sounds (actually three, considering also the silence as one), the neural network is a classifier which input is the FFT of a time window and the output has one-hot encoding.
This simple model is expected to work because of the simplicity of the problem and the characterization of the two sounds.

1.2.2 How to implement a neural network on a board?

The **X-CUBE-AI** package allows to convert a pre-trained neural network into a library to be called in the code.

2 Design and Implementation

2.1 Board Programming

2.1.1 Board description

The board in use is a **STM32F4 Discovery** (STM32F407VGT6) and is equipped with the open source operating system Miosix.
The peripherals used in this project are the microphone, the user button and the USART, although also the CRC module is enabled because needed by the ST's AI library.

2.1.2 Microphone

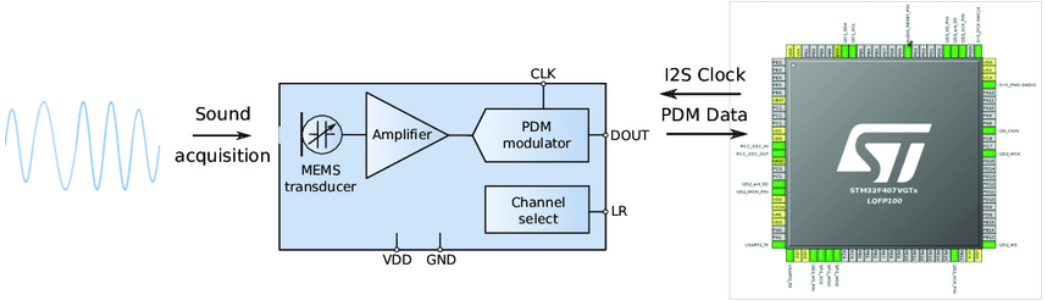
The board is equipped with a **MP45DT02** MEMS microphone, which produces a stream of 1-bit **Digital Pulse Modulation** (PDM) samples. The PDM samples are received using DMA and then converted to 16-bit signed **Pulse Code Modulation** (PCM) samples through the usage of a software low pass filter.
The PCM data is sampled at a frequency of 32 kHz, which is obtained by setting the *PLLI2SN*, *PLLI2SR* and *I2SDIV* registers to appropriate values (see chapters *7.3.23* and *28.4.4* of the datasheet for further details):

$$f_{\text{(VCO clock)}} = f_{\text{(PLLI2S clock input)}} * \frac{PLLI2SN}{PLLM} = 8 \text{ MHz } \frac{213}{8} = 213 \text{ MHz}$$

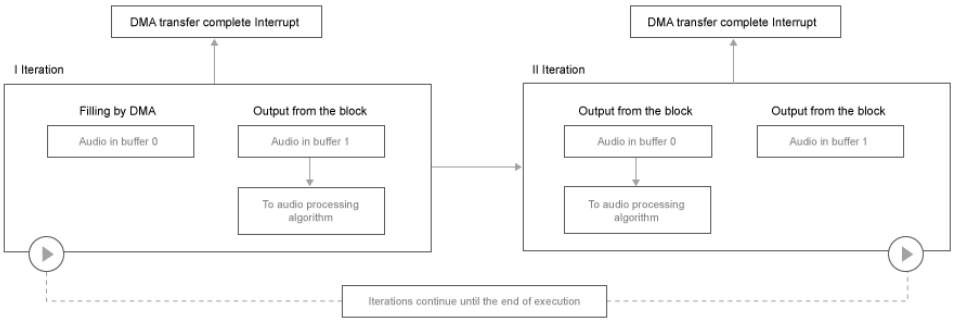
$$f_{\text{(PLL I2S clock output)}} = \frac{f_{\text{(VCO clock)}}}{PLLI2SR} = \frac{213 \text{ MHz}}{2} = 106,5 \text{ MHz}$$

$$f_{\text{(PDM)}} = \frac{f_{\text{(PLL I2S clock output)}}}{2 * I2SDIV * 8} = \frac{106,5 \text{ MHz}}{2 * 13 * 8} = 512 \text{ kHz}$$

$$f_{\text{(PCM)}} = \frac{f_{\text{(PDM)}}}{16} = \frac{512 \text{ kHz}}{16} = 32 \text{ kHz}$$



The driver also uses a double buffering strategy: one buffer (the "ready buffer") contains the stable data that are sent to the network, while the other one (the "processing buffer") is populated by the new incoming samples. When the processing buffer is full, they are swapped and thus the meanwhile collected data becomes ready.



2.1.3 FFT

The PCM samples are considered in blocks of 1024 values, called "windows". Before the values are processed, they are normalized and passed through a **Hann function**, which makes sure that the discontinuity at the windows borders are eliminated[1]. Then the FFT is computed and, being the output symmetric (because all the input has zero imaginary part), only half of the 1024 output bins are taken[2]. Finally, the magnitude is computed on each single value.
Being the Cortex M4 equipped with the **Digital Signal Processor** (DSP), the FFTs are determined using the dedicated hardware.

2.1.4 User button

The user button press is handled by enabling the **EXTI0** interrupt.
The calling thread is paused while waiting for the press event, in order to allow other tasks to be executed meanwhile.
Moreover, a software debouncing feature is implemented in order to avoid

[1]Shlomo Engelberg. *Digital signal processing: an experimental approach*. Springer Science & Business Media, 2008.

[2]Michael Parker. *Digital Signal Processing 101: Everything you need to know to get started*. Newnes, 2017.

multiple clicks due to the mechanical structure of the button. This is achieved by recognizing the first button press and ignoring further clicks for a predefined amount of time (200 ms), whose length is sufficiently long to discard false clicks and small enough to improbably ignore real clicks.

2.1.5 USART

The communication between the board and the client is done through USART connection ($PA2 = TX$, $PA3 = RX$), configured with a speed of 115200 bps. The communication is set in raw mode by disabling the canonical mode and not converting controls (i.e. $\hat{}C$, $\hat{}Z$) into signals.

2.1.6 Neural network library

The neural network library is generated by the X-CUBE-AI expansion pack for STM32CubeMX. This tool automatically converts the pretrained Keras model into a C library that can be called by the main program.
Instead, the core of the network runtime is provided by ST as a static closed source library (*neural_network.a*).

2.1.7 Issues

- The suggested IDE Netbeans gives a lot of troubles in the development, such as missing code completion and unrecognized definitions. So the Miosix build system has been converted to CMake, in order to allow the usage of modern IDEs such as CLion.

- The Miosix toolchain has a bug in the linking phase, resulting in the network runtime library to block in an infinite loop when its functions are called. To solve this issue, the linking process has been executed using the standard ARM linker.

2.2 Network Training

The neural network is described and trained with Python, using **Keras** (version 2.2.4) and **Tensorflow** (version 2.0.0-alpha0).
The data is retrieved from different people, asking them to clap or whistle in several ways. Then, we chose among the data the samples that seemed more peculiar and used them for the network.

The labeling was done by hand, using a numerical code for each class. Then data was shuffled to prevent dependant data to be close (which was the case, since data nearby did probably belong to the same audio record).
The target values (expected results) are extracted and transformed into one-hot code and then given as input to the network. The neural network is a sequential model (defined in tensorflow.python.keras.models) compiled with the *adam* optimizer.
The samples were (randomly) divided into two groups: train data and test data. The first were used for the training phase, while the latter were used for validation.
The final model is a network with one hidden level of ten neurons and an output level. The neurons of the first level use a **ReLu function** (the inputs are summed up with different weights and propagated next), while the output level uses **cross-entropy function**, that transforms the numbers in probabilities. The weights are initialized with a **uniform distribution**, so two different runs may produce slighty different results, but 50 epochs seem to be enough to make this difference really small.
In order to retrieve the aforementioned audio samples, the embedded software can be compiled in two different ways, by enabling or disabling the *TRAINING* variable definition. When the variable is defined, the neural network is not executed and the FFT samples are directly transferred to the computer. Here a specific Python script pass them to a decoder that takes care of converting the raw data into a CSV file for better visualization and usage. With this two-way configuration it is possible to obtain the data to separately train the network, so that, when the network is trained and uploaded, the variable definition can be removed and get back to the normal operating mode, in which the neural network runs and outputs the recognized sounds.

3 Experimental Results

After training the neural network, tests on local samples (different from the training ones) give an accuracy of 99.43%. Although it may be an overfitting warning[3], the runtime recognition is quite good: whistles and silence are perfectly identified, while claps are sometimes mistakenly recognized as

[3]Ben Coppin. *Artificial intelligence illuminated*. Jones & Bartlett Learning, 2004.

whistles, probably due to a low amount of claps training samples.

4 Conclusions

4.1 Future Improvements

The silence samples used to train the neural network only include basic ambient noise and doesn't take into account other consistent interferences such as human voice. Therefore, false results may be obtained if someone is speaking near the board. To fix this problem, it would be necessary to get more training samples covering also human voice and classify them as silence sounds.

4.2 Possible Use Cases

The project can be modified to recognize mostly any type of sound. For example, the first attempt was to recognize the keys pressed on a nearby keyboard[4]; anyway, this attempt failed because of the low SNR and the subsequent difficulty to even detect the key press peaks and their frequencies.

[4]Markus Jakobsson and Steven Myers. *Phishing and countermeasures: understanding the increasing problem of electronic identity theft.* John Wiley & Sons, 2006.