POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Cascading on-device keyword spotting and speaker verification in TinyML

Tesi di Laurea Magistrale in
Ingegneria Informatica

Author: **Gioele Mombelli**

Student ID: 953234
Advisor: Prof. Manuel Roveri
Co-advisors: Ing. Massimo Pavan, Ing. Marco Cova
Academic Year: 2021-2022

Abstract

In recent years, interest in artificial intelligence has grown rapidly thanks to the countless possibilities it opens up. Although the most common direction is towards the development of extremely complex and increasingly demanding systems from an energy and computational point of view, a new branch of this discipline has begun to move towards the execution of such algorithms on integrated systems extremely limited both from the point of view of energy and computing power. This new approach to artificial intelligence has been called Tiny Machine Learning (TinyML), and it has a multitude of advantages that are not limited to reduced power consumption; it also provides greater attention to user privacy and a decrease in response times thanks to the processing of data directly on devices and platforms that are needed to collect them. TinyML could also guarantee a capillary diffusion of intelligent systems thanks to the extremely low cost of the platforms to which they are oriented. This Master's Thesis focuses on two fundamental Tiny Machine Learning tasks: keyword spotting, i.e. the reaction to voice commands pronounced by human beings, and speaker verification, which consists in ability to distinguish people from their own unique voice. The latter problem is tackled organically in the TinyML framework for the first time in this work, framing it in a new one-class few shot classification context. This work demonstrates the possibility of creating a system that combines both functions with a request for resources compatible with the capabilities offered by modern microcontrollers, proposing a comparison between different approaches: some of them were never been used before in a TinyML context, some have been identified and tested for the first time in this work.

Keywords: TinyML, on-device learning, keyword spotting, speaker verification, one-class few shot

Abstract in lingua italiana

Negli ultimi anni l'interesse per l'intelligenza artificiale (IA) è cresciuto rapidamente grazie alle innumerevoli possibilità che essa apre. Nonostante la direzione più comune dello sviluppo di sistemi di intelligenza artificiale sia orientata verso algoritmi estremamente complessi, e sempre più esigenti dal punto di vista energetico e computazionale, una nuova branca di tale disciplina ha iniziato ad esplorare la possibilità di eseguire algoritmi di IA su sistemi integrati caratterizzati da memoria e potenza di calcolo estremamente ridotte. Questo nuovo approccio all'intelligenza artificiale è stato denominato *Tiny Machine Learning* (TinyML), e presenta una moltitudine di vantaggi. Il più rilevante è l'irrisorio consumo energetico di tali sistemi intelligenti, ma vi sono anche una maggior attenzione alla privacy degli utenti e una diminuzione dei tempi di risposta dei sistemi grazie all'elaborazione dei dati direttamente sui dispositivi che si occupano della loro raccolta. Questo approccio all'IA potrebbe anche garantire una diffusione capillare di sistemi intelligenti grazie al costo estremamente ridotto delle piattaforme a cui sono orientati. Questa Tesi di Laurea Magistrale inquadra all'interno del *Tiny Machine Learning* due compiti fondamentali: *keyword spotting*, ovvero la reazione a comandi vocali pronunciati da esseri umani, e *speaker verification*, che consiste nella capacità di distinguere le persone ascoltandone la voce. Quest'ultimo problema viene affrontato in maniera organica in ambito TinyML per la prima volta in questo lavoro, inquadrandolo in un nuovo contesto di classificazione *one-class few shot*. Viene dimostrata la possibilità di realizzare un sistema che combini entrambe le funzionalità a fronte di una richiesta di risorse compatibile con quelle offerte dai moderni microcontrollori, proponendo un confronto tra approcci differenti: alcuni già presenti in letteratura, ma mai usati in questo contesto, mentre altri identificati e testati per la prima volta in questo lavoro.

Parole chiave: TinyML, apprendimento automatico, apprendimento sul dispositivo, riconoscimento vocale, verifica del parlante

Contents

1 | Introduction

In recent years, there has been an increasing demand for human-machine interfaces that operate through voice. These interfaces are becoming necessary in a wide range of systems and are increasingly required in resource-limited devices. Along with common tasks such as the ability to recognize commands, the verification of a speaker's identity through voice has become a necessary feature for many human-machine interfaces. This Master's Thesis aims at investigating techniques for integrating speech processing capabilities on ultra-low power microcontrollers, with a particular focus on the tasks of keyword spotting (KWS) and speaker verification (SV). This research is essential as memory, computational power, and energy consumption pose significant challenges to the complexity of algorithms that can be executed on such systems.

Keyword spotting involves the system's ability to continuously listen for one or more commands, referred to as "keywords," and respond accordingly upon detection. Speaker verification pertains to the system's ability to recognize an individual's voice, and respond accordingly to the speaker's identity. In particular, the implementation of a SV system on a microcontroller-powered device presents significant advantages, including the provision of personalized and privacy-focused vocal interaction solutions that do not require an internet connection or a substantial amount of power. This has a lot of potential practical applications, including smart locks that can recognize their owners, objects that offer varying behaviors based on the user interacting with them, or in general avoiding to actively performing authentication. A solution showing the described behavior should be able to perform keyword spotting and speaker verification both as separate tasks, for example by firstly asking for a voice sample to recognize the speaker and then unlocking voice commands functionalities, and as cascaded tasks, for example by performing speaker verification on the same audio utterance that contained a recognized keyword. Investigating techniques and possibility for implementing this last system on a microcontroller-powered device is the goal of this Master Thesis.

Understanding speech and recognizing different voices are tasks that are performed almost unconsciously by human beings, thanks to biological specialized structures such as the

auditory apparatus and particular portions of the brain. This makes really hard to identify precise rules to be applied to an audio signal to extract information like pronounced words or speakers' identity. This characteristic makes the problem particularly suited to be tackled with modern Artificial Intelligence techniques, mainly the ones based on Deep Learning, a subfield of Machine Learning, that try to mimic the functionalities of biological brains. The complexity of such algorithms, both regarding memory and processing power [52], poses unfortunately a serious limitation in their deployment in portable and embedded devices, where this systems could have great impact.

Recent research however managed to port some machine learning algorithms on ultra-low power devices, such as microcontrollers and embedded computers, giving birth to the emerging field called Tiny Machine Learning (TinyML). One of the first applications targeted by this research field has been indeed keyword spotting, laying the basis for methodologies and techniques to port AI systems in resource constrained computers. However, TinyML is still in its infancy and many tasks still require further investigation using this approach. Speaker verification is a promising new proposal in this regard.

The TinyML approach has a lot of advantages, ranging from the reduced environmental footprint of data processing, the increase in privacy and security given by the absence of data transfer, and the great flexibility given by the variety of embedded systems available. Moreover, the low cost of TinyML systems makes them affordable and suitable to be adapted to countless number of situations.

However, TinyML by its nature presents also several challenges: apart from the complexity given by device heterogeneity and MCU constraints, the vast majority of tiny machine learning applications performs only inference on device, relying on external machines to provide the underlying trained models. This makes TinyML systems harder to adapt to new environments or situations on the field, a great obstacle in the development of truly intelligent low-power systems. For this reason, research is rapidly progressing and frameworks for training models directly on-device are being investigated and developed [15]. Another great obstacle to learning on-device is the absence of labels attached to the collected data. The majority of ML models are trained with a supervised learning approach, which requires a great quantity of labeled data. It is difficult to obtain such data directly on-device without relying on human intervention or already deployed external ML models for labeling. Even if some solutions are being developed, this is still an open challenge [61].

To overcome the problem of data scarcity and absence of labels, this thesis work addresses the speaker verification problem by framing it on a few-shot one-class classification ap-

proach. Few-shot means that only a limited number of voice samples must be needed to setup the system, and one-class means that such samples must come only from the authenticated person, i.e. the person whose voice must be recognized. This approach makes the system interesting also from a purely industrial and commercial point of view, because users would be required to provide just a small set of voice samples to let the system learn their own vocal fingerprint, leading to a product with high usability. An organic approach to the speaker verification task in a TinyML context was still missing in literature and is proposed here for the first time, providing also a comparison of different methodologies including a new custom SV algorithm that obtained promising results.

The keyword spotting task is instead tackled by enriching state-of-the-art procedures, proposing an architecture developed specifically for a dedicated hardware which was never used before for the task.

Among all the systems identified in this work for performing KWS and SV, best-performing ones have been chosen to build a final on-device implementation of a cascaded KWS-SV system able to continuously listen for keywords and, upon a detection, provide a personalized answer if the utterance has been pronounced by the authorized speaker.

1.1. Master Thesis Structure

The second chapter of the Master Thesis provides the background needed to understand TinyML state-of-art regarding audio and speech processing techniques and methodologies. Third chapter details the solution of the Keyword Spotting task. The fourth chapter similarly details the approach to the Speaker Verification task. The fifth chapter reports testing results, and the sixth chapter describes the implementation of our solution on-device. The last chapter lays the basis for future works.

2 | Background Research

This chapter will provide an analysis of background literature related to the thesis. After a brief introduction to artificial intelligence and machine learning, there will be an analysis of audio and speech processing in such contexts, followed by an introduction to Tiny Machine Learning. Two subsections will also be dedicated to insights about common audio preprocessing and machine learning models for TinyML oriented speech processing systems, with a particular focus on the tasks addressed by this work: keyword spotting and speaker verification.

2.1. Artificial intelligence and machine learning

Artificial Intelligence is a broad term that identifies different techniques, algorithms and subfields of Computer Science with the common goal of modeling rational behaviors typical of biological beings. Machine Learning is a subfield of Artificial Intelligence that allows developing programs that autonomously improve by learning from their own experience, thanks to collected data. Artificial intelligence algorithms based on machine learning models have become more and more affirmed in a society that produces more data than ever. Such algorithms leverage huge processing power usually available only on high-end machines, and they are often distributed as a service relying on cloud computing. A specific term for this distribution of AI systems has been defined, naming it AIaaS (Artificial Intelligence as a Service) [23]. This approach requires the collected data to be transmitted using network connections, providing astonishing performances but also posing serious questions about energy consumption, privacy of transmitted data and network usage.

The majority of Machine Learning algorithms is based on an inductive approach to the problems to be solved: starting from a collection of facts, usually collected data, they try to extract a general rule to act on such information to describe it, classify it or predict other information. The general rule identified by a Machine Learning algorithm is indeed a program able to solve the specific task. Such programs are referred to as "models". It is worth noting that a Machine Learning model learns to extract information, and does not

produce information from scratch [40].

Machine Learning problems can be divided into three main categories: supervised learning, unsupervised learning and reinforcement learning [21]. The speech processing problems addressed in this work can be respectively classified into two of these categories.

Keyword Spotting is a supervised learning task, where the algorithm has to learn to map speech inputs to specific known categories corresponding to the words to be recognized.

Speaker Verification is an unsupervised learning task, where the algorithm must learn an efficient representation of the input speech signal to extract speaker-dependent information and recognize the speaker's identity.

Machine learning is a broad term that comprehends a variety of different models, each one more suitable to be applied to certain tasks than others. Regarding KWS and SV, great results have been achieved by algorithms belonging to the subfield of machine learning called deep learning [22]. Deep learning identifies all the machine learning algorithms in the shape of deep neural networks, models composed by layers of artificial neurons stacked to resemble the structures of biological brains. Such algorithms are particularly prone to learn rules to solve problems that have no clear algorithmic solution, but are intuitively solved by biological beings. Under this category fall problems related to vision, audio processing and, in general, pattern recognition in data.

2.2. Audio and speech processing in AI/ML contexts

Voice is the primary means of communication for human beings. Our species' remarkable ability of articulating complex sounds is at the basis of speech, the way we communicate each other thoughts and ideas.

It is no surprise that designing a machine able to understand language as a human would do is a problem that has intrigued scientists for centuries. Speech processing, the field that aims at developing such machines, presents signs of life since early years of computer science. [20]. The speech processing field has a vast range of applications, such as human-computer interaction, computer linguistics, speech-based psycholinguistics and of course language processing [7].

Although humans can intuitively recognize speech and voices, the underlying reasoning process is not easily explainable, and therefore, defining a precise algorithm to replicate these tasks on a machine is not trivial, mainly because it is not possible to explicitly identify rules and operations to be applied on speech signals.

Such characteristics make this problem well-suited to be solved by applying modern Artificial Intelligence (AI) techniques, particularly those in the subfield of Machine Learning (ML) called Deep Learning [22]. Strictly regarding audio and speech processing, ML techniques obtained promising results and affirmed as the state-of-the-art approach in such tasks [7]. However, the complexity and computational requirements of deep learning algorithms are well-known and represent one of the major issues of the current AI technology domain [52]. Providing a detailed analysis of all speech processing applications and paradigms is outside the scope of this thesis, so this section will focus on machine learning approaches to the speech processing tasks addressed in this work.

2.2.1. Audio preprocessing

Humans perceive sounds thanks to specialized cells sensitive to variations in air pressure. The frequency and the amplitude of such variations influence pitch and loudness of perceived sounds. Common digital audio representations encode the sound waves directly by using continuous sequences of numbers, called samples, that can be processed by computers and eventually converted again into sounds via loudspeakers. While this encoding is useful for human beings, because it is directly linked to our biological mechanisms, it may not be as effective for artificial intelligence algorithms that have to extract information from it; main issues with such encoding are related to the high number of samples needed to represent audio, and the lack of immediately distinguishable information from their observation.

Although the literature presents examples of speech processing systems that process raw audio waveforms [18] [28], the state-of-the-art approach requires extracting features from audio signals which highlight relevant information, and the most common feature extracted are in the form of Mel-Frequency Cepstral Coefficients (MFCCs) [24].

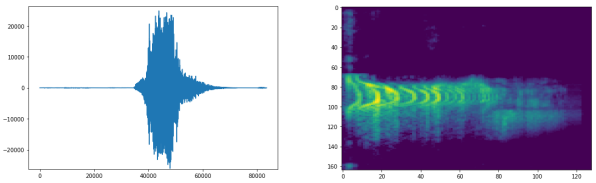


Figure 2.1: A time-domain audio file (left) and the related MFCC plot (right).

MFCCs are coefficients obtained from a raw audio signal that represent its spectrogram, which is the distribution of energy for the various frequencies in a signal, in a way that considers non-linearities of the human ear perception. Differently from traditional spectrograms, MFCCs present an emphasis in frequencies below 1000 Hz, where the human

ear is most sensitive due to evolutionary reasons: the majority of human voices have a fundamental frequency that ranges from 85 to 255 Hz [41].

The fundamental steps to obtain MFCCs from a waveform audio signal are the following [43]:

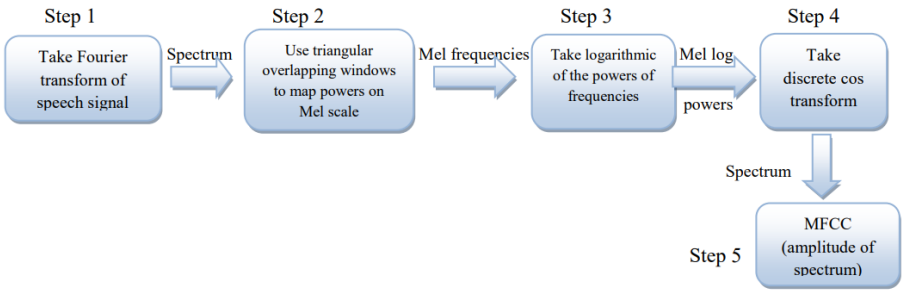


Figure 2.2: Steps for extracting MFCCs from raw-waveform audio files.

The most important advantage of using Mel-Frequency Cepstral Coefficients as features in speech processing is that they capture and highlight characteristics of speech that are normally hidden in raw waveform signals, as it can be seen at a glance from the following image:

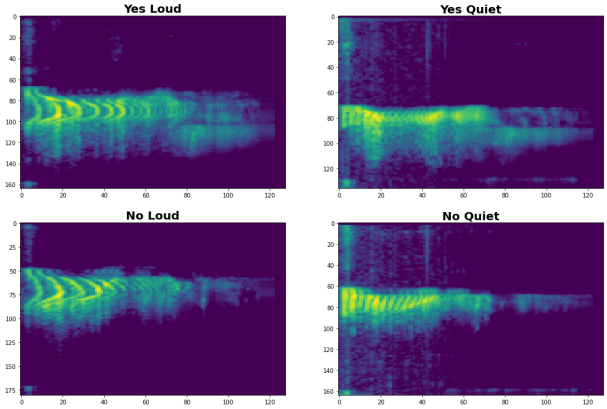


Figure 2.3: We can notice that MFCCs for "yes" and "no" are different just at a glance.

Moreover, they reduce the complexity of the processed data by switching from a time-domain signal to a frequency-domain signal with the consequence of reducing the dimensionality of data needed to be stored and processed.

They however present some disadvantages: they tend to be a sub-optimal choice in noisy environments [33], and they require some expertise in setting parameters such as the

filters' bandwidth or the number of filters themselves [32].

MFCCs remain anyway the most common features when developing speech processing system, mainly because of their low dimensionality and great capability of extracting meaningful information from audio signals containing human voices.

More details on MFCC preprocessing adopted for this thesis work can be found in section 3.4

2.3. Keyword Spotting

Keyword Spotting (KWS) can be identified as the task of detecting the presence of a predefined set of words, called keywords, inside an audio stream containing human speech. There are no limitations on the number of keywords that can be recognized by a system in order to classify it as a KWS system: they can range from one single word, usually called "wake-word", to more complex sets of commands. Moreover, also locutions composed by multiple words can be treated as keywords, such as the common "Hey Siri" or "Ok Google". However, recognize complex phrases and long commands is not the ultimate goal of KWS, where the short duration of the speech being processed is the common factor to all the applications.

2.3.1. Historical approaches

Historically, several different approaches to keyword spotting have been developed: ranging from LVCSR systems [26, 63] to the Hidden Markov Model-based approaches [42, 64].

The advent of Deep Neural Networks (DNNs) in 2014 brought a sensible improvement in performances and keyword spotting tasks, in particular for a Tiny Machine Learning perspective. Deep neural networks are a relatively new kind of algorithm which gave birth to the subfield of machine learning called deep learning. Deep learning has shown promising performances in a great variety of tasks, including speech processing problems. Major differences with the previous approaches are that DNNs are able to provide results that can be directly interpreted as the presence of keywords, without relying on complicated sequence-search algorithms (i.e. Viterbi decoding) [24]; moreover, their complexity can be adjusted to fit the computational resource constraints. This combination of simplicity and effectiveness made Neural Networks the state-of-the-art approach KWS, given the fact that target devices for this particular task often pose constraints in computational power and complexity [24]. The general scheme for a neural network based keyword spotting system is the following:

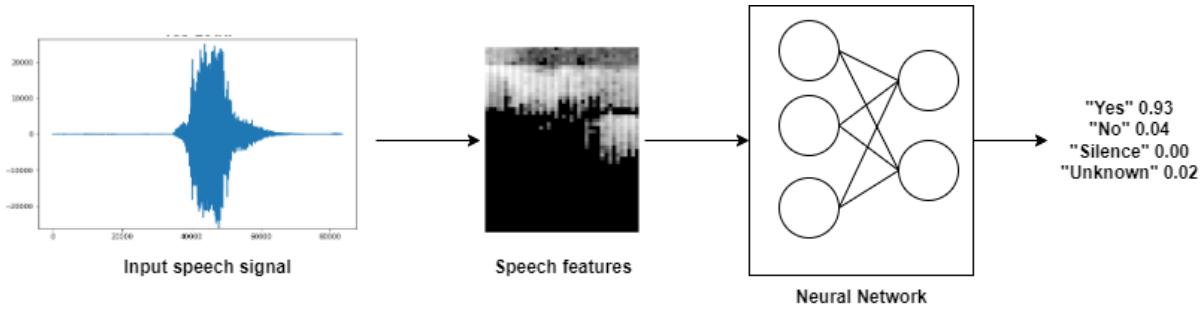


Figure 2.4: Steps of a KWS system based on deep learning.

Neural networks require to be trained on large quantities of data related to the task to be solved. For KWS application, such data must be in the form of audio recordings of the desired keywords, possibly collected by a large number of different subjects for capturing differences in pronunciation, accent and voice tone.

2.3.2. Google Speech Commands Dataset

Google Speech Commands [60] is a dataset specifically designed for building and testing on-device keyword spotting models, collected in 2017 by asking volunteers to produce audio samples, with the ultimate goal of providing a common ground for benchmarking KWS systems and speeding up progress and collaboration. This dataset was deemed particularly interesting also because it has been collected by taking into account the needs of hardware manufacturers; the author wanted that chip vendors could use this dataset to demonstrate accuracy and energy usage of their products in a way that is easily comparable for potential purchasers [60]. This translated into data collected not by relying on studio microphones and treated environments, but using phone or laptop microphones in normal rooms and situations. However, for privacy reasons, each audio sample was collected by asking volunteers to speak in situations where there were no background conversations. The dataset is composed by 24 common words chosen from the English dictionary used as core, with the intent to reflect some common useful commands for IoT or robotics applications: the digits from “Zero” to “Nine”, plus “Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", “Backward”, “Forward”, “Follow”, and “Learn”. Moreover, since KWS models have to deal also with rejecting background speech and non-keywords commands, ten more words were added with the intent of testing this capability: "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree", and "Wow". The final list of words contains 34 elements, and the number of available samples is reported in table A.1.

Each audio sample in the dataset has been collected with a sampling frequency of 16 kHz, and each word is already cropped to be contained in a 1-second long audio file.

2.3.3. Multilingual Spoken Words Corpus Dataset

The whole Multilingual Spoken Word Corpus is a large and constantly growing audio dataset of spoken words in 50 languages, collectively spoken by over 5 billion of people, for academic research and commercial applications in keyword spotting and spoken term search [25]. The dataset contains over 340000 keywords, totaling 23.4 million 1-second spoken examples. Differently from the Google Speech Commands, keywords in this dataset have been automatically harvested by applying algorithms for extracting 1-second long single word utterances from longer speech audio segments. This makes samples in this dataset less clean and defined with respect to ones contained in Speech Commands. Each file is stored in .opus file format, and is encoded with a 48 kHz sample rate.

2.4. Speaker Verification

Lots of subtasks of speech processing have the ultimate goal of extracting information about the identity of a person from samples of speech. Speaker verification refers to the task of identifying a precise speaker, called "enrolled" or "authenticated" speaker, from all the other possible speakers. The output of a speaker verification system is a confidence value on the correspondence between the voice contained in the input audio sample and the identity of the enrolled speaker. According to the confidence value provided by the system, a decision can be taken.

2.4.1. Historical approaches

Speaker verification field had a similar evolution to the KWS field: traditional methods for performing speaker verification included Gaussian Mixture- Model-Universal Background Models (GMM-UBM), Gaussian Mixture-Model Support Vector Machines (GMM-SVM), Joint Factor Analysis (JFA) and i-vectors [13, 65]. All of the aforementioned methods required heavy statistical computations and modeling.

With the advent of deep learning and its strong representation and classification abilities, the research took two directions: one included DNNs in the traditional frameworks, like the DNN/i-vector approach [65], the other is completely based on DNNs to try to extract a representation of speakers' voice characteristics in a low-dimensional space called "embedding", on which classification and comparison algorithms can be run [65]. Neural

networks took the place of the statistical models that used to be built to represent voice features. Several neural network architectures have been tested for this scope since, ranging from classical Feed-Forward Neural Networks [56], Convolutional Neural Networks [44] and Recurrent Neural Networks [29], but there are also promising techniques that involve deep belief networks [6] and variational autoencoders [57]. Purely deep learning based methods for performing speaker verification are d-vectors [56], x-vectors [49] and j-vectors [50].

Independently from the chosen approach, however, speaker verification process is composed by three major phases:

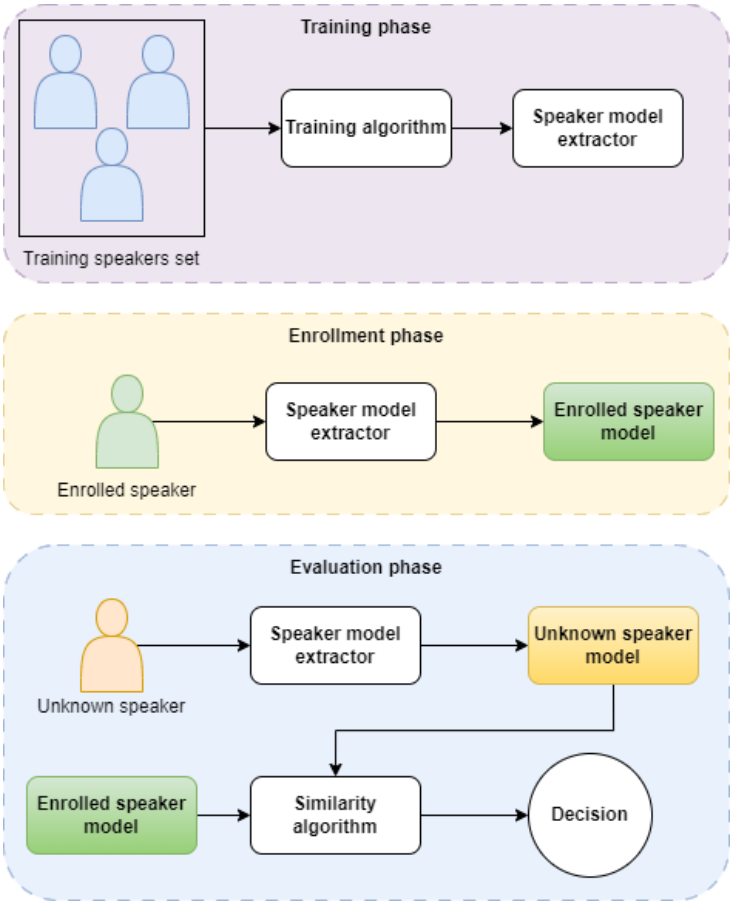


Figure 2.5: Phases of speaker verification.

Training phase: this step consists in building an algorithm able to extract the unique characteristics of a speaker's voice, called speaker model extractor. If a machine learning approach is chosen, this step usually involves training a model for recognizing different speakers. In deep learning contexts, it is simple a classification system against a set of N speakers, where the goal of the model is to obtain the best performance possible. This phases must be executed only once during the design of the

system. Only the speaker model extractor will used in the next phases.

Enrollment phase: the speaker verification system must now be calibrated to recognize a new speaker, called "enrolled" speaker. This step will require the user to provide some voice samples, from which the system will extract the enrolled speaker model, i.e. a unique representation of the speaker's voice. This model is saved for future comparisons.

Evaluation phase: this is the inference phase, where new audio utterances from unknown users are submitted to the system. For each utterance, a model for its author's voice is produced and confronted against the stored model of the enrolled user. A decision on the identity of the speaker is taken according to the similarity measured via a similarity algorithm.

Different speaker verification systems can use different speaker model extractors or different recognition algorithms, but the underlying backbone is the same.

For the same reasons highlighted in Keyword Spotting tasks, some approaches purely based on neural networks were deemed particularly prone to be explored also in resource constrained computers. This is the case of the D-Vector approach, and the next section will provide more details about.

2.4.2. D-Vector approach

This approach has been extensively described for the first time in [56]. It is based completely on deep neural networks for all the phases of the speaker verification task. The core of the approach relies on using deep neural networks as speaker model extractors, to leverage their capability of learning abstract and more compact representations of input data.

Referring to figure 2.5, the step of building the speaker model extractor is represented by the "training phase". In this approach, to obtain the speaker model extractor we have to build a deep neural network to classify the speakers in the training speakers set. Such network, in the proposed architecture, is composed by several hidden layers and one last output layer with one neuron for each speaker in the training set.

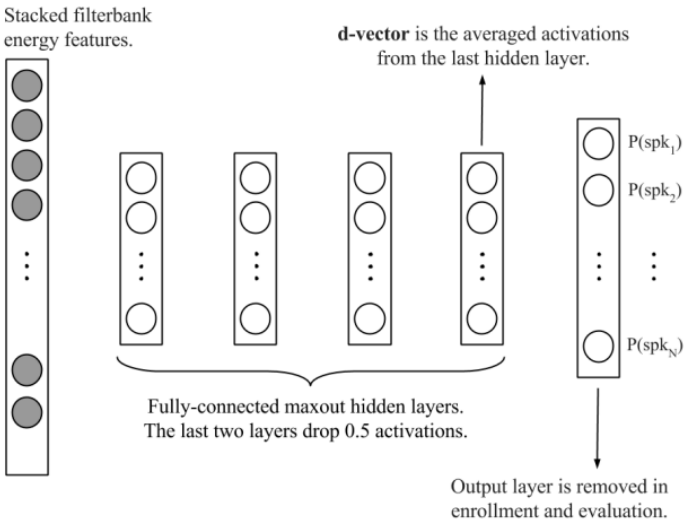


Figure 2.6: The speakers classifier DNN [56].

Once the neural network is trained on the set of speakers and able to recognize their voices, the last classifier layer of the network is removed, leaving exposed the outputs of the neurons of the last hidden layer. This vector is referred as "deep-vector" or d-vector.

The underlying assumption is that each speaker produces a unique d-vector, and the last classifier layer simply learns to associate it to one of the speakers in the training set. However, and here lies the strength of the method, also speakers that are not part of the training set should be able to produce a unique d-vector.

With the removal of the last layer we have obtained the speaker model extractor, that produces models in the shape of d-vectors.

The enrollment phase, on the yellow box of figure 2.5, happens by submitting to the neural network a certain number of voice samples from the enrolled speaker, obtaining an equal number of d-vectors. Such vectors are stored, and their element-wise average is taken as the enrolled speaker model. Such model is stored for future comparison.

The evaluation phase, as described in figure 2.5, happens when an unknown voice sample is submitted to the system. The neural network extracts the D-Vector from the utterance, and by using a similarity measure such as cosine similarity, performs the confrontation of the new D-Vector with the stored enrolled speaker's average D-Vector. More in detail, cosine similarity between two vectors x and y is defined as:

$$cossim(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{xy}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{x}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{y}_i)^2}} \tag{2.1}$$

If the similarity is above a certain threshold, determined by the designer in an empirical way according to sensitivity needs, the utterance is evaluated as belonging to the enrolled speaker. If the similarity is below the threshold, the identity of the speaker is deemed different from the enrolled speaker.

The whole process can be summarized with the following scheme:

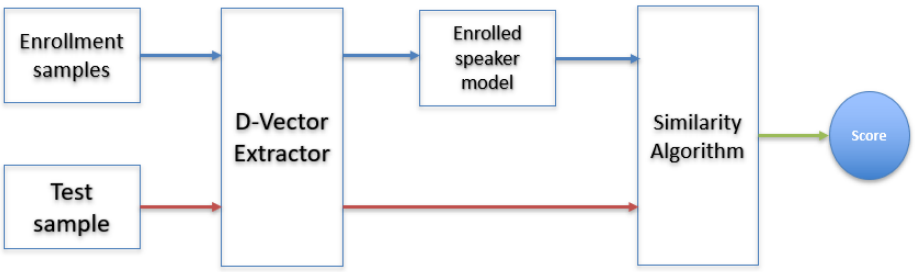


Figure 2.7: D-Vector speaker verification approach.

The enrolled speaker model must be obtained only once, and during the evaluation phase is retrieved from memory and used as a confrontation.

As it can be noticed by the general mechanisms of this system, there are a lot of moving parts that could benefit from improvement. For example, literature has proposed to take d-vectors as activations of layers different from the last hidden one, by cutting the networks higher in their structure [48]; other works proposed Convolutional Neural Networks or Locally-Connected Networks instead of Feed-Forward Neural Networks to be used as D-Vector extractors, as in [10]. The final design of a d-vector based speaker verification system is driven by the needs of the application and the resources available to the engineer, so it is particularly suited to be ported also in low-power computers and embedded devices.

2.4.3. Text-Dependent and Text-Independent speaker verification

Speaker verification can be performed in two different ways, namely "text-dependent" and "text-independent". The first one refers to the capability of the system of recognizing the voice of the enrolled speaker when a specific word or phrase is pronounced; the second one defines a system able to recognize the speaker independently from the pronounced words [55].

Text-independent speaker verification is generally deemed more challenging to solve; however, both the problems present pros and cons. In a text-dependent context, the constraint

on lexical content translates into a low degree of phonetic variability, which eases the work of the recognition algorithm. In principle, text-dependent SV could operate on extremely short utterances like passphrases or even single words, leading to systems with a quick response time. This is not true for text-independent contexts, where to attenuate the problem of phonetic variability usually engineers rely on processing longer speech utterances [55]. However, it is easier to find data related to text-independent speaker verification, while text-dependent contexts could require collecting in-domain data for training machine learning models, which increases development costs and time.

2.4.4. Librispeech Dataset

LibriSpeech is described in [36] and is a corpus of read English speech, suitable for training and evaluating speech processing systems, and in particular for text-independent speaker verification systems. It contains 1000 hours of speech sampled at 16 kHz from thousands of different readers. The dataset contains training, validation and testing partition, without overlapping of speakers between the three sets. The dataset presents gender balance at the speaker level and in terms of the amount of data available for each gender. Moreover, audio from each speaker has been limited to 25 minutes, to avoid unbalance between speakers. The training portion of the corpus is split into three subsets, with approximate size 100, 360 and 500 hours respectively. This has been done because the sheer size of the dataset made a single distribution impractical.

The following table describes the structure of the dataset as in [36]:

subset	hours	per-spk minutes	female spkrs	male spkrs	total spkrs
dev-clean	5.4	8	20	20	40
test-clean	5.4	8	20	20	40
dev-other	5.3	10	16	17	33
test-other	5.1	10	17	16	33
train-clean-100	100.6	25	125	126	251
train-clean-360	363.6	25	439	482	921
train-other-500	496,7	30	564	602	1166

Table 2.1: LibriSpeech dataset subsets details.

2.4.5. Evaluation metrics

The most common metrics for evaluating threshold-based biometric identification systems are the *Receiving Operating Characteristic Curve* (ROC curve), the *Area Under The Curve*

(AUC) metric and the *Equal Error Rate* (EER). The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings:

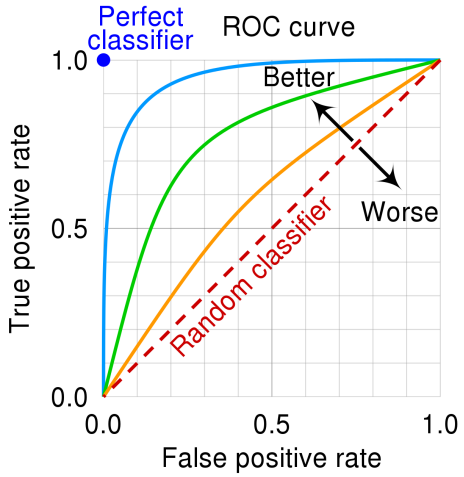


Figure 2.8: ROC curves for different classificators.

The AUC metric is a measurement of the capability of a binary classifier to distinguish between classes, and is used as a summary of the ROC curve. The higher is the AUC metric, the better is the model in discriminating between the two classes. AUC is equal to 1 for a perfect classifier, and equal to 0.5 for a random classifier.

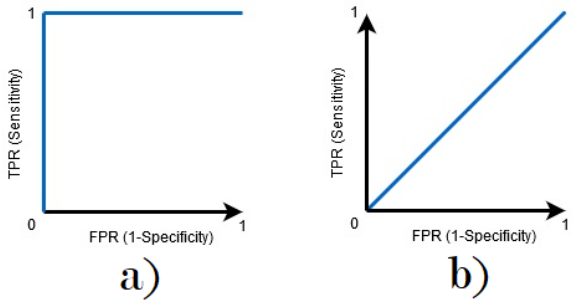


Figure 2.9: a) ROC of perfect classifier with AUC = 1. b) ROC of random classifier with AUC = 0.5

The EER is computed by finding the location on a ROC curve, corresponding to a threshold value, where the false acceptance rate and false rejection rate are equal. Such rate is called EER. In general, the lower the equal error rate value, the higher the accuracy of the biometric authentication system.

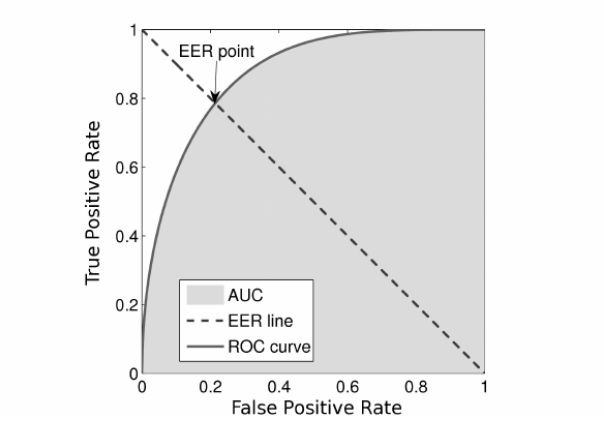


Figure 2.10: Finding the EER by intersecting the EER line with the ROC curve [54].

The aforementioned evaluation metrics are useful because they remove the dependency on the threshold when evaluating the system, giving insight on its behavior at each possible value of the threshold. However, they can not be applied to speaker verification approaches that provide a discrete classification value such as authenticated-not authenticated, because there is no threshold to compute ROC.

An evaluation metric that could be used for every speaker verification system is *accuracy*, defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

Where TP are true positives, TN are true negatives, FP are false positives and FN are false negatives. Informally, the accuracy of a classifier algorithm on a dataset represents the percentage of predictions that the model got right. While it may seem the best evaluation metric to compare the proposed solutions, accuracy alone is not enough when dealing with a class-imbalanced dataset, where there is a significant disparity between the number of positive and negative labels.

To overcome this problem, two more suitable evaluation metrics are available, namely *precision* and *recall*. They are defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.3}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.4}$$

Both the precision and the recall are focused on the positive class (the minority class) and are unconcerned with the true negatives (majority class). They are useful metrics to assess performance of a classifier on a minority class in an unbalanced dataset. To fully evaluate the effectiveness of a model, there is the need of examine both precision and recall. Unfortunately, they often are in a trade-off: improving precision typically reduces recall and vice versa.

A useful way to evaluate at the same time precision and recall is by using a metric that combines the two values. *F1-score* is defined as the harmonic mean of precision and recall, computed as:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{2.5}$$

The highest possible F1-score obtainable is 1.0, meaning that precision and recall are both 1.0, and the worst possible value is 0.0, indicating that one between precision and recall is 0.

2.5. Tiny Machine Learning

In later years, our society showed an exponential diffusion of Internet of Things and embedded devices, integrating processing units in more and more battery-powered everyday objects. It has been estimated that by 2025 more than 75 billion embedded devices will be connected to the internet [14]. Such items have direct access to data thanks to sensors that perceive the world around them, but do not possess the processing capabilities of larger systems, posing limitations to what they can accomplish.

Tiny ML is the subfield that tries to merge two realms of the current technological domain: the high performing and data-hungry artificial intelligence algorithms and the flexible and low-power embedded devices; its ultimate goal is to move the AI closer to the data sources, potentially directly on the devices that have the burden to collect data. With this idea in mind, a definition of Tiny ML was proposed as follows: "machine learning architectures, techniques, tools and approaches capable of performing on-device analytics for a variety of sensing modalities (vision, audio, motion, chemical, etc.) at "mW" (or below) power range targeting predominately battery-operated devices.". [17]

TinyML approach to artificial intelligence can provide a lot of advantages. First of all, on a society more and more concerned about power consumption and energy saving, ultra-low power computers equipped with AI systems could operate for long periods of time relying

only on batteries or even in-place green energy harvesting, reducing the environmental footprint of complex computations that nowadays require much more power. TinyML also allows for affordable and flexible devices, thanks to the low deployment costs and the extreme heterogeneity of embedded systems. Moreover, on-device processing removes the need of transferring data from the source (sensors) to the processing server, reducing not only bandwidth consumption, but also risks of cyberattacks, preserving users' privacy [45]. On the other hand, TinyML is not free from challenges and problems. It is difficult to define general frameworks and toolchains due to the heterogeneity of MCU existing; this leads to inconsistency in approaches and methodologies and makes benchmarking complex [5]. Moreover, TinyML systems could suffer from the lack of adaptability in presence of different situations and concept drift. This happens because the vast majority of tiny machine learning applications performs inference on-device relying on pre-trained models, produced by external powerful machines: a change in environmental or operation context would require an update of the device with a new model adapt to the evolved situation.

More in detail, TinyML identifies two main approaches to the porting of machine learning models on embedded devices with serious resource constraints:

On-Device inference only: approach based on training models on high-end machines leveraging affirmed ML libraries, such as TensorFlow [3], and then converting them in a format suitable for the low-power system to be used in an inference-only fashion. Several open source frameworks such as TensorFlow Lite have been developed to perform the conversion and on-device execution of ML models [3], but also closed-source internal solutions have been developed by companies, like the Infineon ModusToolbox ML-Tools for MCUs [19].

On-Device training and inference: this second approach relies on integrating also the training procedure on embedded devices, allowing systems not only to perform inference but also to generate new models from data retrieved directly on-device, without the need of relying on external machines [45]. This is often referred to as "training on-device". Being this a more complex task both in term of required processing power and memory needed, it is more common to find Tiny ML implementations that rely on the on-device inference only, even though research is always improving and trying to investigate ways to optimize the training procedure to be executed directly on-device [2]. Moreover, this approach suffers also from other limitations of TinyML, one of which being the lack of labeled data during training. Most common machine learning models are trained in a supervised way, where the algorithm needs to process a large quantity of data associated to labels in order to extract useful

patterns and rules. There is no easy way to obtain labeled data directly on-device, without the intervention of a human or an already trained classifier [61]. This is why the inference-only approach is the most common.

The vast majority of Tiny ML frameworks focus on a single type of machine learning algorithm, neural networks. However, some libraries like sklearn-porter and m2cgen also allow the porting of algorithms like k-NN, SVMs and Random Forests [1, 30]. For the scope of this thesis, literature research showed that neural networks are the most effective method for solving the specific problems addressed [24], even though some other methods were also investigated. We adopted Tensorflow and Keras [11] as frameworks for developing deep learning models, and we relied on the company developed Infineon Inference Engine [19] for deploying the algorithms on-device. All our solutions, however, can be ported also to open-source frameworks such as TensorFlow Lite Micro [12].

It is worth noting that the experience of the designer plays a fundamental role even in the early stages of design of the ML model for the intended application. For example, even though several techniques for reducing the size of the models have been developed, such as quantization and pruning [45], there are limitations to the effectiveness of the reduction applied. Models which are too large won't be able to fit in the embedded system, even after quantization and pruning. Moreover, if a model requires too much calculations, inference times could grow in such a way that a real-time execution, if needed, might be infeasible.

A Tiny Machine Learning designer must also keep in mind that every pre-processing step applied to data during the training of the applications, when this step does not happen on-device, must be exactly reproducible and feasible to be computed also at inference time on the low-power computer. This poses serious constraints on the design of the algorithms from early stages of the development.

2.6. Speech processing in TinyML contexts

Among the vast possibilities regarding Tiny ML applications, audio and speech processing tasks were deemed particularly interesting. Applications and systems that perform speech processing are already available in our homes and even in our pockets, but they often rely on sending audio stream data to large servers to be processed by complex machine learning models. Continuously sending audio data on the network poses serious problems, not only regarding the power consumption, but also from a privacy perspective.

Tiny Machine Learning initially allowed the development of always-on ultra-low power

systems with the purpose of listening for specific words ("Hey Siri" or "Okay Google" are two well-known examples [4]) and performing the detection locally, completely on-device, sending the audio stream to be analyzed on servers only after such detection happened.

Later, with advance in research and development, complete speech commands applications became feasible to be run in embedded systems, removing the need of communicating with external servers even for more complex interactions.

Lot of other interesting speech processing applications could benefit from a TinyML implementation, such as emotion, health, language, accent, age, or gender recognition.

Among others, keyword spotting is an application that perfectly showcases the advantages of Tiny ML, being privacy oriented and providing an energy efficient, fast and offline inference. This is why it is often considered as the first application to be ported on an embedded system targeted at speech processing [62]. Moreover, it allows designers to acquire confidence with the state-of-the-art preprocessing required for speech-based applications.

3 | Keyword Spotting

This chapter will report the approach to Keyword Spotting adopted in the Master Thesis. After a general definition of the problem, an insight about datasets used, feature engineering and neural network architectures will be provided. The chapter ends with a performance report and few notes regarding on-device implementation. More details about on-device implementation can be found in chapter 6.

3.1. Problem definition

This thesis work addresses the keyword spotting task from both a high-level perspective, due to its didactical significance, and a practical application-oriented perspective, with the aim of creating a versatile system that could be adapted to support various real-world use-cases. The ultimate goal is to provide the company with a KWS system capable of supporting different demonstrations, such as simple wake-word features, vocal controls for a mobile robot or generic voice-controlled portable devices. The primary source of information for the implementation of the KWS system has been the work of Peter Warden and Daniel Situnayake, as described in [62]. The proposed approach rely on training a machine learning model in a supervised way; input data must be composed by recordings of the chosen keyword(s), recordings of speech corresponding to "unknown" words and noise samples corresponding to the "silence" condition. The model must be able to listen and provide a real-time classification of the input audio in one of the aforementioned categories. Our first attempts focused on making the system recognize between a single keyword, silence and unknown speech, but later on we moved onto more complex recognizers with more than one command detectable.

3.2. Application architecture

Being this work application-oriented, it is useful to provide insight on the structure of the system from a high-level perspective by identifying the various components of the KWS application. However, details related to the on-device implementation will be clarified

later on, with this chapter mainly focusing on machine learning related components. Let's start by providing the general steps that a KWS machine learning application should perform:

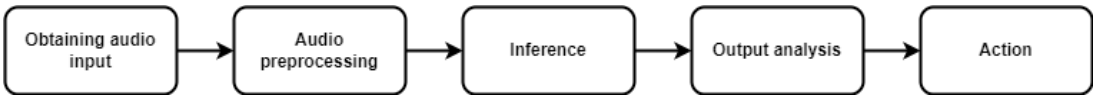


Figure 3.1: Steps of a KWS system.

Obtaining audio input: Audio data must be collected from a source like a digital microphone.

Audio preprocessing: Audio requires heavy preprocessing before it can be fed into the model.

Inference: A machine learning model processes the input data and provides a confidence value on the presence of keywords.

Output analysis: Output of the machine learning model is analyzed.

Action: An action is taken depending on the presence or absence of keywords.

The core of the system is composed by a machine learning model that has to process audio data and provides a confidence value on the presence of the chosen keywords. The model adopted is a Convolutional Neural Network that does not process raw audio data, but relies on inputs in the shape of MFCC features. One of the key features of the system is that it must be able to run in real-time, processing audio data as soon as they are available from an acquisition device, and providing an acknowledgment of a detection in a reasonable amount of time. All of these characteristics will guide the development of the machine learning model.

3.3. Datasets

For the scope of this thesis, and in agreement with the company, we chose to rely only on publicly available datasets for keyword spotting. This limited the possible keywords to the ones included in such datasets, but on the other hand allowed us to provide a confrontation of our implementation performance on standard datasets. Two principal keywords datasets were used in this work: Google Speech Commands Dataset [60] and Multilingual Spoken Words corpus [25]. Such dataset presented a selection of keywords already classified, and with enough samples for training deep learning models with satisfying results. It is important to notice that the samples were already split in audio

segments of 1 second, each one containing one single keyword. Moreover, for data augmentation purposes, 6000 seconds of background noise samples in different environments were collected, both by directly recording environmental noise and from publicly available background noise videos.

3.3.1. Google Speech Commands Dataset

After an analysis of the available words, we decided that a system able to recognize the keyword "Sheila" would have been our first goal. Even though such keyword is part of the dataset, it is not one of the words intended to be used as a command, but as a test for the background speech rejection. However, we believed that the number of "Sheila" samples available was enough to train a model and obtain satisfiable results. During this master thesis, other commands from the dataset were used to build several keyword spotting systems, such as "Yes", "No" and some digits. Unused commands were exploited to make our models more robust to background speech and noise.

3.3.2. Multilingual Spoken Words Dataset

Given the size of the dataset and the variety of words proposed, we decided to extract a subset of such dataset to be used in development of our KWS systems. Only the English language words have been extracted, and among all of them we kept only the ones that had more than 800 samples available. This search identified 1239 different words suitable to be used. Due to the inconsistence in quality and number of samples of this dataset, it has not been our primary benchmark during training and testing phases. Instead, we relied on samples from this dataset only after having designed the KWS system, for the purpose of adding more commands to our demos to showcase complex vocal interactions.

3.3.3. Additional Background Noise

6115 additional 1-second long samples have been collected by recording or extracting background and environmental noises. The additional samples belong to the following environmental recordings:

Environment	Time length
Living room (quiet)	2m:02s
Living room (rainy day)	10m:03s
Library	10m:00s
Office	20m:00s
Restaurant	10m:03s

Table 3.1: Environmental recordings for background noise.

These recordings were taken with a digital microphone at a sample rate of 16kHz. Samples have been extracted from these recordings by taking 1-second long windows with an overlapping of 0.5 seconds. This data will be used to make the model more robust to environmental and background noise, both for identifying keywords even if the environment presents some noise, but also for detecting the absence of speech and keywords.

3.3.4. Data conversion and augmentation

The target hardware where the keyword spotting system will be executed can provide audio samples with a sample rate of 16 kHz via a digital microphone. For training purposes, each audio sample must be converted in a .wav file with a sample rate of 16 kHz. This conversion has been applied to all the samples in the Multilingual Spoken Words dataset by relying on ffmpeg library [53], while the Speech Commands samples were already in the correct format. We decided also to perform data augmentation on the available samples to increase their number, but also for allowing our models to distinguish keywords also in noisy environments. Formally, let $\boldsymbol{x}$ be an input audio sample in the shape of a vector of $16,000$ elements. Data augmentation is the function $a(\boldsymbol{x})$ that produces two outputs $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$, also representing audio samples in the shape of vectors with $16,000$ elements:

$$\begin{pmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{pmatrix} = a(\boldsymbol{x}) \tag{3.1}$$

By writing equation 3.1 highlighting the intermediate steps, we obtain the following equation:

$$a(\boldsymbol{x}) = \begin{pmatrix} no(gs(ts(\boldsymbol{x}), \boldsymbol{n}_1), \boldsymbol{n}_1) \\ no(gs(\boldsymbol{x}, \boldsymbol{n}_2), \boldsymbol{n}_2) \end{pmatrix} \tag{3.2}$$

Where $ts(\boldsymbol{x})$ is a function that takes as input an array $\boldsymbol{x}$ and returns an array of the same size but with the elements shifted forwards or backwards by a step of size $s \in [-4800, +4800]$. $gs(\boldsymbol{x}, \boldsymbol{n})$ is a function that takes as input an array $\boldsymbol{x}$ and a noise sample array $\boldsymbol{n}$ of size $16,000$ and performs a gain augmentation or reduction on $\boldsymbol{x}$ according to the difference in amplitude between $\boldsymbol{x}$ and $\boldsymbol{n}$, returning an array of $16,000$ samples. Finally, $no(\boldsymbol{x}, \boldsymbol{n})$ performs the overlapping between the noise sample $\boldsymbol{n}$ and the audio sample $\boldsymbol{x}$ by performing element-wise sum on the two arrays.

Equation 3.2 has been implemented with the following pipeline:

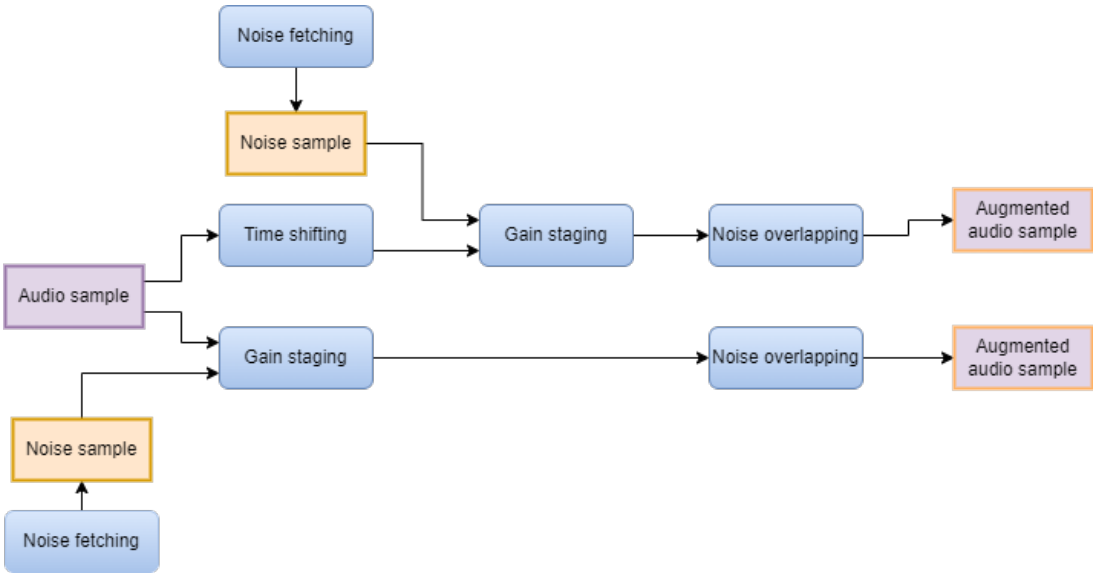


Figure 3.2: Data augmentation pipeline

Noise fetching block a random noise sample is picked from the available ones.

Time shifting block Implementation of function $ts(\boldsymbol{x})$. The audio sample fetched is shifted by +/- 300ms at most (4800 samples), in order to have the same word represented in different positions inside the 1-second window. This will help the recognizer to discriminate better even on early or later windows. The shifted sample is padded with 0s.

Gain staging block Implementation of function $gs(\boldsymbol{x}, \boldsymbol{n})$. This step performs an amplitude difference measurement between the audio sample and the noise sample: if the signal to noise ratio is too low, the word audio sample is amplified to have a clearer utterance.

Noise overlapping block Implementation of function $no(\boldsymbol{x}, \boldsymbol{n})$. Performs the overlapping between the audio sample and the noise sample.

As it can be noticed, the data augmentation has been designed to produce two different samples consuming one. The first one is obtained by time shifting the original one and overlapping some noise, while the second one is obtained by overlapping alone. Noise samples are different for the two cases to increase the variety of noise. However, the code also allows for keeping only one of the two audio samples produced by data augmentation, if needed. A code snippet of the data augmentation pipeline implemented in Python is the following:

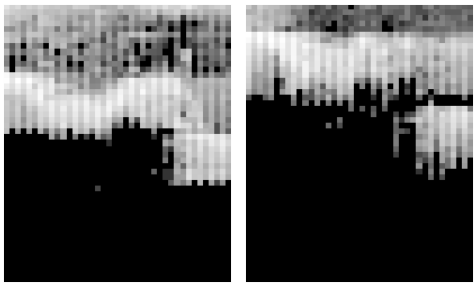
```
def runAugmentation(input_sample, sample_path, noise_sample, destfolder,
                                        overwrite=True):

  print("Processing " + sample_path)

  augmented_sample_file = os.path.basename(sample_path)

  if(not overwrite): # Performs data augmentation pipeline branch with
                                          time shifting
    augmented_sample_file = "aug_" + os.path.basename(sample_path) #Name
                                             of new sample
    augmented_sample = timeShift(input_sample) #Shift in time

  augmented_sample = gainAdjust(augmented_sample, noise_sample) #Gain
                                          correction
  noiseOverlap(augmented_sample, noise_sample, os.path.join(destfolder,
                                          augmented_sample_file)) #
                                          Overlapping of noise and audio +
                                          saving
```

This process of data augmentation is applied statically once before each training session, and only to training data. Testing and validation data are left without augmentation in order to test the model performances on the clean dataset for benchmarking purposes.

3.4. Feature Extraction

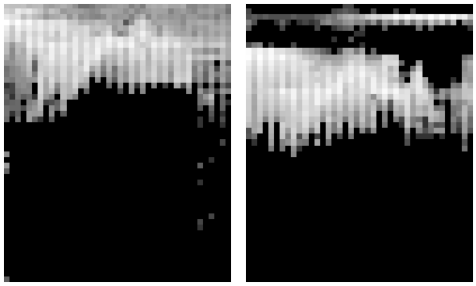
Feature extraction is a fundamental step of a lot of machine learning applications; it is a process that changes the shape of data to highlight relevant information to facilitate the work of the machine learning model [62]. The advantages given by an intelligent feature extraction are not only related to the overall performance of the model; even the size of the data itself can benefit from the process, and this is of particular interest for our application, given the TinyML context we are operating in. The feature extraction system as described in [62] provides both of aforementioned advantages, relying on the extraction

of MFCC coefficients (see 2.2.1) with a total of 1960 values, obtaining a reduction of 87% in data size and highlighting differences between MFCCs of different words. Let's take as an example four MFCC spectrograms obtained with this system, two from "yes" utterances and two from "no" utterances:



(a) Utterance 1. (b) Utterance 2.

Figure 3.3: MFCC spectrograms of "Yes" words.



(a) Utterance 1. (b) Utterance 2.

Figure 3.4: MFCC spectrograms of "No" words.

As it can be noticed, each word is characterized by its own unique spectrogram shape, making easy to distinguish them even at a glance. This is much harder to do by looking only at raw audio waveforms. This is why we adopted the same preprocessing pipeline reported in [62], which has been built by Google engineers for their production releases but has never been described in literature, with just a few modifications. Formally, let $\boldsymbol{x}$ be an input audio sample in the shape of a vector of $16,000$ elements. Feature extraction can be modeled as the function $fe(\boldsymbol{x})$ that produces the output matrix $\boldsymbol{S}$, representing the MFCC coefficients obtained from the input audio.

$$\boldsymbol{S} = fe(\boldsymbol{x}) \tag{3.3}$$

The output matrix S has the following shape:

$$\boldsymbol{S} = \begin{bmatrix} s_{11} & s_{12} & s_{13} & \dots & s_{1n} \\ s_{21} & s_{22} & s_{23} & \dots & s_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & s_{m3} & \dots & s_{mn} \end{bmatrix} \tag{3.4}$$

Where each element s_{ij} represents the i-th coefficient of the j-th time interval considered in the computation of the MFCC spectrogram. m and n are determined by the parameters of the MFCC extraction algorithm: n represents the number of frequency buckets identified and m the number of time intervals considered in the computation. In our application, the MFCC extraction function requires processing a one-second long audio file with a 30ms wide sliding window, shifted at each step of 20ms. From each 30ms window, one row of $\boldsymbol{S}$ is produced by applying the following pipeline:

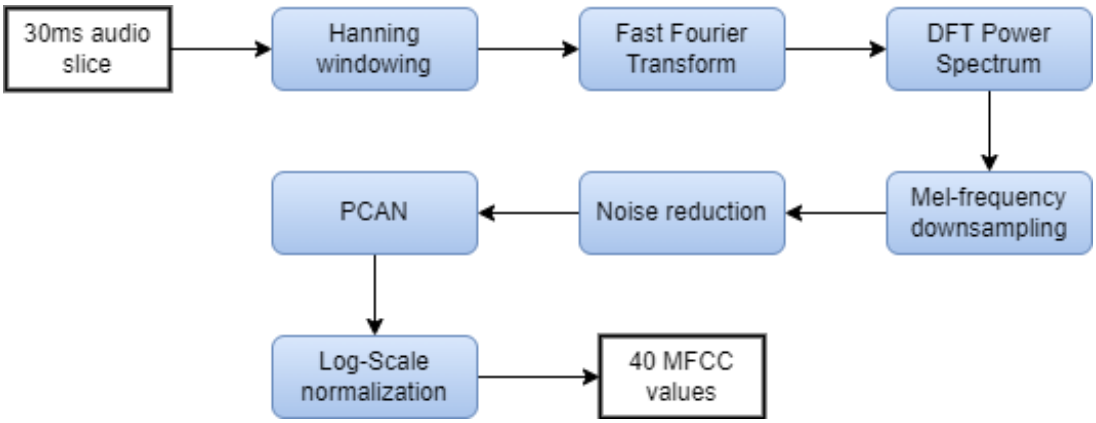


Figure 3.5: MFCC extraction pipeline

Hanning windowing block: The 30ms audio portion is processed through a Hanning window to reduce spectral leakage at the edge of the signal, thanks to its ability of reducing discontinuities:

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) \tag{3.5}$$

In (3.5), N is the length of the window;

Fast Fourier Transform block: This block implements the computation of the Fast Fourier Transform of the signal (FFT). FFT is a way to obtain the Discrete Fourier Transform, which is a mathematical operation that transforms a time-domain signal

into its frequency-domain representation:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-2\pi j \cdot \frac{kn}{N}} \tag{3.6}$$

where $X[k]$ is the k-th frequency component of the signal in the frequency domain, $x[n]$ is the n-th sample of the signal in time domain and k is an integer from 0 to $N-1$, representing the frequency bin index. This operation takes $N = 480$ samples in time domain as input, which is the number of samples contained in 30ms of audio a with a sampling rate of 16kHz. A 0-padding is added to reach 512 samples, since the FFT algorithm requires a power of two as number of samples in input. The result provides $(N/2) = 256$ different frequencies.

DFT Power Spectrum block: This block computes the power spectrum from the DFT values. Since the DFT outputs complex numbers, with a real and imaginary part, we can obtain their magnitude via the relation:

$$|z| = \sqrt{\mathrm{Re}(z)^2 + \mathrm{Im}(z)^2} \tag{3.7}$$

Where $|z|$ is the magnitude for each frequency;

Mel-frequency downsampling block: The output of the DFT is composed by 256 frequency buckets. This number is quite high; moreover, such frequencies are evenly spaced. To model the characteristics of the human ear, non-linear downsampling is applied, mapping the frequencies to the Mel-scale leveraging triangular filters. The mapping of frequencies happens following this scheme [47]:

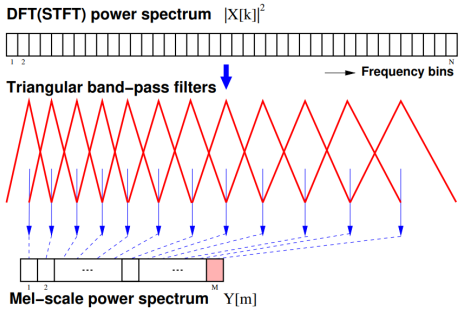


Figure 3.6: Conversion to mel-frequencies

Triangular filters act by drawing and aggregating information from different frequencies, with a higher resolution in lower frequencies and a lower resolution towards higher ones. This is modeled by the distance and the wideness of the triangular fil-

ters. These two parameters are precisely identified by the mel-scale, and the relation between Hertz and mels is the following [34]:

$$m = 1127 \ln\left(1 + \frac{f}{700}\right) \tag{3.8}$$

This preprocessing pipeline implements 40 mel-scale filters, that will give us 40 frequency values, a lot less than the starting 256 frequency values.

Noise reduction block: A unusual aspect of this feature extractor pipeline is the presence of a noise reduction block. A running average of the value of each frequency bin is kept in memory during the execution, and this average value is subtracted by the instantaneous value at each computation. The rationale behind this operation is that background noise is deemed to be fairly constant over time, and present only in particular frequencies. By subtracting the running average, we have better chances to remove constant noise in certain parts of the spectrum. Moreover, this computation requires different coefficients for the even and the odd frequency buckets, explaining the comb tooth shape of the spectrograms. This has been deliberately introduced because empirical evidence showed an improvement in performance. More details on this can be found in [59].

PCAN block: This block implements the Per-Channel Amplitude Normalization autogain to boost the signal based on the running average noise. This step is designed to work in conjunction with the noise reduction block: it works by dividing the amplitude of each frequency bucket by a normalization factor that is derived from the average noise level in that channel. This has the effect of boosting the signal-to-noise ratio of the speech signal in each frequency, while preserving the relative power distribution across the spectrum;

Log-Scale normalization block: At the end of the process, this block performs a normalization with a logarithmic scale of the values in the frequency buckets. This is needed to avoid having relatively loud frequencies that drown quieter ones, facilitating the recognition system that will have to work on these samples.

As it can be noticed by the scheme, this pipeline has been carefully designed to extract all the relevant information from the audio signal, including also a noise reduction part embedded in the pipeline itself. The slice produced contains $n = 40$ frequency values, and by repeating the process for the whole 1-second audio segment we can obtain $m = 49$ slices. At the end, each spectrogram is composed by $40 \times 49 = 1960$ values, and can be interpreted as a 2D black and white image. We have also adopted a further normalization

step at the end of this pipeline. The original transformations produced MFCC coefficients in the range between 0.0 and 26.0. We decided to normalize each value between -1.0 and 1.0 to ease the training process. It is worth mentioning that this preprocessing pipeline must be implemented in the same exact way both for the training, testing and validation steps during the model development, but also in the microcontroller unit during inference time. This is a general rule that holds in every TinyML application that relies on executing inference on-device and training on dedicated machines: the preprocessing pipeline must be suitable to be executed also in the MCU, and must be designed with this goal clear in mind to avoid introducing a bottleneck in the system. Our implementation relies on the already optimized code by [62].

3.5. Neural Network Architecture

A fundamental part of the development of a machine learning algorithm is the choice of the model. As anticipated in section 2.2, deep learning models are particularly suited for solving speech processing tasks. Given the preprocessing steps applied to the audio, which turned audio signals into 2D images representing MFCC spectrograms, the problem of identifying different keywords can be thought as a pattern recognition problem in 2D images. The goal of our model is to learn the shape of the spectrograms of the desired keywords, and return high confidence values when one of the desired patterns is observed in input. Among all the possible deep learning models, Convolutional Neural Networks are particularly suited for solving patter recognition tasks in images, making them the state-of-the-art approach to image classification [39]. A CNN is a deep neural network where specialized layers, called convolutional layers, have the purpose of identifying patterns inside images by using sliding filters. Fully connected layers can be used to interpret such similarities and provide a confidence value on the presence of certain elements in the input image. Among all the possible CNN architectures, the choice must be guided with the final goal of developing a model small enough to fit into microcontrollers. This means that there is a limitation on deepness and number of parameters of the models, such as filters' size and neurons per layer. For the purpose of developing the KWS system we tried two different architectures, varying only in the size of the convolutional filters. While such architectures are largely based on the one proposed by [62], we adopted two convolutional layers instead of only one, and we relied on this structure for all the experiments. This led to smaller networks that however scored satisfying accuracy values. To clarify the structure of our models, here is a scheme representing the networks:

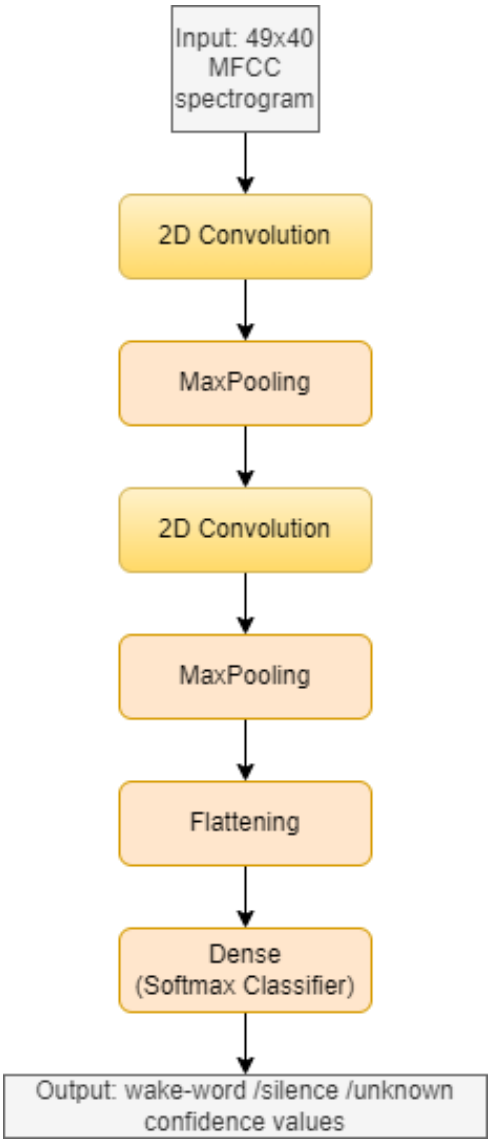


Figure 3.7: KWS model architecture

The number of parameters in this architecture changes according to the number of classes, i.e. the number of keywords that a single network must be able to recognize, and according the number and size of filters of convolutional layers. While the number of classes depends on the desired behavior of the application, size and number of filters can be tweaked by the architect to trade-off between power and size of the model. By using this architecture as common ground, we tested two different models varying only in the size of the convolutional filters. The example brought by [62] involved one single convolution with kernel size (8×10); we tested one model with a kernel size of (8×20) for the first convolution and (4×10) for the second convolution, and another model with smaller kernels of size respectively (4×4) and (3×3). As it can be seen from the tables below,

the size of the filters plays a fundamental role in the total number of parameters of the model, which in turn determines the final size of the neural network:

Model: smallconv-kws-nn			
Layer type	**Output shape**	**Settings**	**Parameters**
Input	(49, 40, 1)	None	0
2D Convolution	(25, 20, 16)	Kernel size = (4, 4) Filters = 16 Stride = (2, 2)	272
2D MaxPooling	(12, 10, 16)	Pool size = (2, 2)	0
2D Convolution	(12, 10, 32)	Kernel size = (3, 3) Filters = 32 Stride = (1, 1)	4640
2D MaxPooling	(6, 5, 32)	Pool size = (2, 2)	0
Flattening	(960)	None	0

Dense	(#output classes)	Units = #output classes
		Activation = softmax
			960 * (#output classes)
			+ (#output classes)

Table 3.2: Structure of small CNN for keyword spotting.

Model: conv-kws-nn			
Layer type	**Output shape**	**Settings**	**Parameters**
Input	(49, 40, 1)	None	0
2D Convolution	(25, 20, 16)	Kernel size = (8, 20) Filters = 16 Stride = (2, 2)	2576
2D MaxPooling	(12, 10, 16)	Pool size = (2, 2)	0
2D Convolution	(12, 10, 32)	Kernel size = (4, 10) Filters = 32 Stride = (1, 1)	20512
2D MaxPooling	(6, 5, 32)	Pool size = (2, 2)	0
Flattening	(960)	None	0

Dense	(#output classes)	Units = #output classes
		Activation = softmax
			960 * (#output classes)
			+ (#output classes)

Table 3.3: Structure of large CNN for keyword spotting.

In the scope of our KWS application, we investigated models able to distinguish between one, two or three different keywords at the same time, in addition to silence and unknown speech (i.e. all the speech that is not part of the chosen keywords set). The simple case with one keyword can be good for developing simple wake-word applications that rely on activating more complex speech processing pipelines, both on-device or relying on cloud computing, while networks able to distinguish between different commands can be useful to develop complete on-device voice-controlled applications. As mentioned before, the number of keywords has an influence on the number of parameters of the models, and of course on their final accuracy. Without changes in the architecture, the more words a model is required to recognize the lower its accuracy will be.

Each one of the models reported here has been trained and tested on data taken from Google Speech Commands dataset for comparison purposes. Models that have been found particularly interesting from an application perspective have been also trained and tested on keywords taken from the Multilingual Spoken Words corpus. However, this dataset has been chosen with the precise goal of developing demos and use-cases thanks to its large number of keywords, without the intent of using it as an organic benchmark for our models. The next section will report details on training and testing results for each model on keywords taken from the Speech Commands dataset.

3.6. Training and testing

We identified six different neural networks, depending on the two different architectures and the three different number of possible keywords to be recognized. Each model has been trained on the same subset of keywords, all taken from the Google Speech Commands dataset. Chosen words are the following, according to the number of terms to be recognized:

Chosen keywords	
Number of keywords	**Keyword(s)**
1	"Sheila"
2	"Yes", "No"
3	"One", "Two", "Three"

In addition to the chosen keywords, KWS models should also be able to recognize between silence and unknown words. Silence samples have been taken from the background noise recorded as explained in section 3.3.3, while the set of unknown speech has been built by drawing randomly words from the Speech Commands dataset not belonging to the

desired keywords' classes. More in details, this is the dataset preparation pipeline for the training phase:

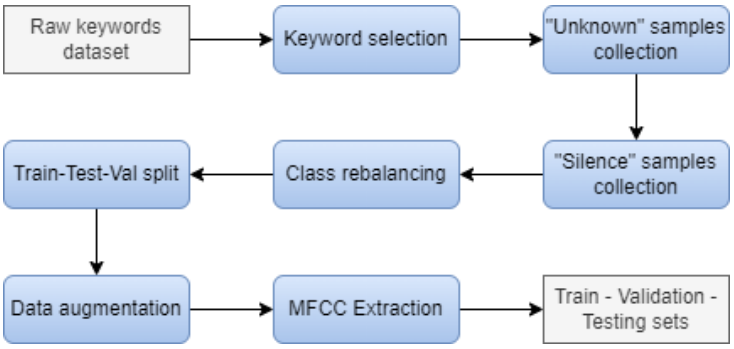


Figure 3.8: Steps to obtain training, testing and validation sets for KWS models training.

Keyword selection: Among all the possible keywords contained in the dataset, a subset is chosen as targets for the current model. All other words are suitable to be used for training the "unknown" samples.

"Unknown" samples collection: All words that do not belong to the desired keywords are merged together in a "Unknown" class.

"Silence" samples collection: This step collects the "silence" samples by importing the background noise samples recorded as explained in section 3.3.3.

Class rebalancing: Since "unknown" and "silence" classes may contain a lot of more samples than the desired keywords classes, this steps performs a rebalancing where the number of samples gets balanced. Parameters of this step can be manually tuned by the designer: the choice we made was to remove randomly some samples from "silence" and "unknown" classes until their size matched the size of the largest keyword class. If the dataset is unbalanced even in the number of samples for keyword classes, this step can act also on such classes to let them have the same number of samples.

Train-Test-Val split: Available samples are divided into training set, validation set and testing set. Chosen proportions for such sets are respectively $80\%, 10\%, 10\%$.

Data augmentation: Data are augmented according to the pipeline explained in section 3.3.4. This process gets applied only to the training set, leaving testing set and validation set without augmentation to act as a benchmark on the raw dataset.

MFCC extraction: At the end of the process, each sample in training, testing and validation set is processed according to the pipeline explained in section 3.4 to

extract the related MFCC spectrum to be fed into the neural networks.

This process of data preparation is performed statically before training. The neural networks have been trained for 200 epochs with a batch size of 8, using the Adam optimizer and a learning rate of 0.0001. Early stopping has been used to control overfitting with a patience of 30 epochs on the validation loss. Testing results produced the following accuracy values:

Model: conv-kws-nn			
Number of keywords	**Testing accuracy**	**Testing loss**	**Parameters**
1	0.9450	0.1743	25,971
2	0.9269	0.2319	26,932
3	0.8960	0.2900	27,893

Table 3.4: Accuracy results for large CNN.

Model: smallconv-kws-nn			
Number of keywords	**Testing accuracy**	**Testing loss**	**Parameters**
1	0.9350	0.2045	7,795
2	0.9275	0.2301	8,756
3	0.8729	0.3644	9,717

Table 3.5: Accuracy results for small CNN.

As it can be noticed by the testing results, both the models show a discrete capability in distinguishing keywords, with the larger network performing slightly better than the smaller one. The large CNN provides a mean accuracy of 0.9226 and a mean loss value of 0.2320, while the smaller one provides a mean accuracy of 0.9118 and a mean loss value of 0.2663. Preferring the smaller network to the larger one gives a decrease in mean accuracy of 1%, but allows reducing almost three times the size of the models, which in a TinyML context is a fundamental feature. Thanks to optimization techniques such as model quantization and pruning [45], models can be compressed to fit into the small space available in microcontrollers. Both the models tested in this section are suitable for the execution on real hardware even in the larger version. The following section will provide estimations on memory consumption and inference time of the aforementioned models, and more details on the on-device implementation will be provided in chapter 6.

3.7. Memory and latency estimation

Memory and latency estimations for models have been obtained via the ModusToolbox Machine Learning Configurator provided by Infineon Technologies. The software performs the conversion of neural networks in a format suitable for the execution on microcontrollers, thanks to a custom ML inference engine for MCUs called IFX Inference Engine, and provides measurements on memory and latency estimations. The tool is also able to perform network quantization, which is a technique for reducing size of a neural network that involves quantizing its weights and activations to different bit depth values, such as 16 bits or 8 bits, reducing its memory footprint. Estimated memory consumption is provided as flash memory estimation (i.e. the space needed for storing layers and weights of the models) and buffer size memory estimation (i.e. the space needed for storing the activations at runtime). Remember that during execution, the MCU RAM needs to have enough space for storing both weights and activations. Estimated inference time is computed by measuring the cycles needed by the MCU to run a single inference of the model, which is the forward propagation from input to output.

Various quantization levels are applied to the models by the tool to provide a comparison. In table 3.6 and 3.7 results are reported, respectively for the large version CNN with 3 keywords as target and for the small version CNN, again with 3 keywords as target.

Model: conv-kws-nn			
Quantization	**Flash memory**	**Buffer size**	**Inference time**
float	112.5 kB	79.3 kB	0.336 s
int16x16	56.4 kB	45.6 kB	0.268 s
int16x8	28.3 kB	45.6 kB	0.302 s
int8x8	28.3 kB	25.8 kB	0.309 s

Table 3.6: Memory and latency estimations for large KWS CNN - 3 keywords version.

Model: smallconv-kws-nn			
Quantization	**Flash memory**	**Buffer size**	**Inference time**
float	39.2 kB	39.6 kB	0.046 s
int16x16	19.7 kB	21.8 kB	0.037 s
int16x8	9.9 kB	21.8 kB	0.042 s
int8x8	9.9 kB	11.9 kB	0.043 s

Table 3.7: Memory and latency estimations for small KWS CNN - 3 keywords version.

In this tables, various quantization levels are reported. The `int16x8` quantization means that the activations have been quantized to 16 bits, and the weights have been quantized at 8 bits. This is why there is a reduction in flash memory but not in buffer size with respect to the `int16x16` quantization level.

Inference time values are computed considering as a target MCU the 150-MHz Arm® Cortex®-M4 system-on-chip. This is a quite powerful microcontroller, but it has been chosen because it will be the target hardware for the implementation of our system. Refer to chapter 6 for more details. It is also worth noticing that such target is optimized for executing 16 bits operations: this is why the lower inference time for each model is obtained by the `int16x16` quantization level.

It is useful to notice that small convolutional neural networks are much faster in inference time and require a lot less memory to execute: this makes them suitable to be executed even on small MCUs. The reason lies on the different kernel size, which reduces the number of operations needed at each layer. The memory/time saving is so noticeable that makes the loss in accuracy reported in section 3.6 acceptable also for real use-case developments.

Remember that such estimations come from conversion and quantization algorithms applied by the chosen tool: using different frameworks to convert the models in a MCU suitable format could lead to slightly different results, according to optimizations implemented.

4 | Speaker verification

This chapter will report the approach to speaker verification adopted in this work. After a general description of the problem and the goals of the development, a note on custom datasets used during this work will be provided. Sections related to the algorithm and methodologies will follow, along with a description of the training and testing processes. Testing results will be commented on the next chapter, while on-device implementation will be explained in chapter 6.

4.1. Problem definition

The speaker verification task in its general characteristics has been described in section 2.4. While a lot of the approaches reported have shown promising results, very few information is available about architectures specifically designed for the execution on ultra low-power MCUs in a TinyML context. The vast majority of architectures proposed in section 2.4 involve models too big to fit into small MCUs, or require computations which would be too complex to be carried on by such processors. Before analyzing techniques to tackle speaker verification in a TinyML context, a formal definition of the task is provided.

Let $\boldsymbol{x_u}$ be an array of integer values of size $16,000$ representing one second of audio recorded at $16kHz$, containing speech signal from a speaker s_u. Let also be $\boldsymbol{m_e}$ the so-called enrolled speaker model: a representation of the unique vocal characteristics of a chosen speaker s_e. The speaker verification system can be modeled as the function v:

$$v(\boldsymbol{x_u}, \boldsymbol{m_e}) \mapsto [0, 1] \tag{4.1}$$

This function returns 1 if $s_u = s_e$, i.e. if the speaker that produced the utterance x_u is the same speaker that provided the speaker model m_e. As it can be seen from the following scheme, the function produces an authentication value $auth$ that can be used to affirm that the unknown speaker s_u is indeed speaker s_e, also called "enrolled speaker" or "authenticated speaker".

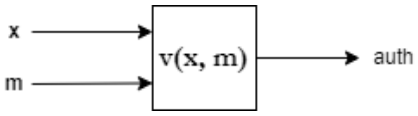


Figure 4.1: Modeling of a general speaker verification system.

4.2. Goal of the experiments

The goal of the experiments conducted is to test and compare carefully designed machine learning algorithms for speaker verification, with the final target of deploying on-device a demonstration of such a system. Apart from all the constraints given by the limitations of the target hardware, there are other conditions that emerged from an analysis of the context we are operating in, taking into account both the needs of a hypothetical final user but also the work done on keyword spotting. Such conditions are one-class classification, few-shot classification, on-device adaptation to new speakers and synergy with the keyword spotting system already developed. The following sections will provide more details about all of the mentioned goals that the system should achieve.

4.2.1. One-Class Classification

The speaker verification task can be solved as a particular classification task, namely one-class classification.

Classification is one of the most common problems solved by machine learning applications. The usual approach is to train a machine learning model able to distinguish between two or more classes with a training set containing objects from all the classes. In one-class classification (OCC) [31], instead, the ML algorithm has to learn to distinguish objects of a specific target class amongst all objects belonging to all possible existing classes, by primarily learning from a training set containing only the objects of the target class. This is much more difficult than the previous case, and in one-class classification can also fall problems related to outlier detection, anomaly detection and novelty detection [38].

Figure 4.2: Different classification types [38].

Speaker verification can be thought as a one-class classification problem because the ultimate goal of the system is to learn to distinguish one single class (the enrolled speaker voice samples) to all the possible existing samples belonging to all other classes (voice samples from all other speakers in the world).During the enrollment of the speaker, no samples from other speakers are available to the system to perform a confrontation, and this fits the task in the OCC category.

This feature guided us in the development of the system, by experimenting with algorithms generally suited for one-class classification. More details on the chosen approach are given in the next sections.

4.2.2. Few-Shot Classification

In addition to the one-class condition, the problem of speaker verification can also be considered among Few-Shot Classification problems. Few-shot classification aims to learn a classifier to recognize unseen classes during training with limited labeled examples [9]. In the speaker verification context we are considering, which is strongly tied to MCU-based systems with end-user interactions, it is reasonable to believe that users that want to enroll will be required to provide only a few samples of their voice: ultra-low power microcontroller are often very limited regarding memory and storage space available, and a user must be able to perform the enrollment without having the burden of collecting large quantities of samples.

Few-Shots classification in the context of speaker verification means that the system must be able to learn the enrolled speaker's model by listening only to a very limited number of voice samples from such speaker. During testing, we tried to measure performances of our models taking into account the few-shots condition by using small sets of enrollment samples. More details can be found in section 5.1.

In Few-Shot classification, a common approach is to train and develop models not able to recognize specific sets of classes, but to recognize if two objects are sufficiently similar. Later, by using few labeled samples from the target class as support, the system should in principle be able to associate to such class new samples which are deemed sufficiently similar [51]. As it can be noticed, this approach reflects the general speaker verification framework adopted in this work, and this is why our problem can also be considered a few-shot classification one.

4.2.3. On-device adaptation to new speakers

Another fundamental characteristic of the developed system must be the ability of enrolling the speaker directly on-device. No interaction with an external machine, both via a physical connection or network communication, must be needed. There are two main reasons behind this need: one, of course, is the industrial need of developing products that are easy to use by customers, and removing the need of additional hardware for the set-up of the system would be extremely valuable. The other reason is that this makes the system interesting also from a TinyML point of view, where the intelligent adaptability of an embedded system is a feature that is highly sought in this field. The development of truly intelligent TinyML models requires having the possibility of adapting their behavior directly on the field.

This will constraint our choice in the algorithms: we have to make sure that the enrollment phase can be performed by the low-power microcontrollers in all of its step, and in a reasonable amount of time. This will also guide our choice of final models for on-device deployment.

4.2.4. Cascading with Keyword Spotting

The keyword spotting system described in chapter 3 should still be part of the system. This means that the microcontroller should show capabilites of recognizing keywords and eventually of performing checks on the speaker's identity. While developing a fully functional keyword spotting-speaker verification cascaded system is not the ultimate goal of this work, design choices must be taken to allow future implementations of such complex interactions.

A SV system that works cascaded with a keyword spotting system could provide personalized behavior if the word spoken is deemed to come from the enrolled speaker, as the following scheme represents:

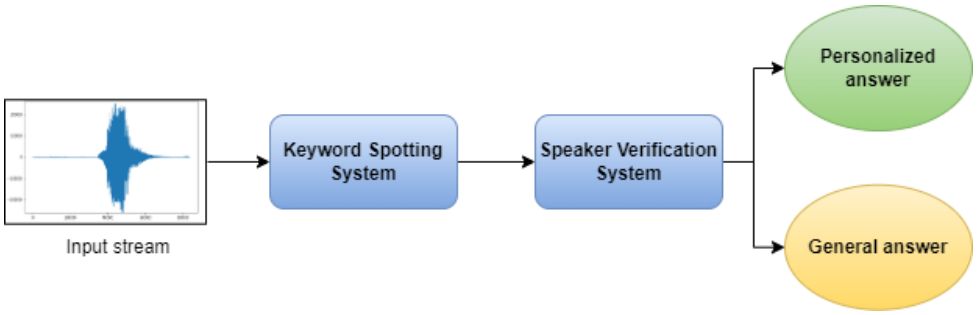


Figure 4.3: A possible KWS-SV cascade application.

This means that in a single-core setting the two systems must be small enough to be executed on the same microcontroller unit. Several optimizations and design choices must be done to achieve this result. For example, this is why we decided to share the same feature extraction backbone for both systems. More details are given in section 4.3.

4.3. Approach to Speaker Verification

A choice on the approach to the speaker verification problem must be taken considering all the constraints of the specific context we are operating in. Among all the methods proposed in section 2.4, the ones completely based on neural networks seem the more suitable to be ported on a microcontroller, thanks to the existing consolidated TinyML methods for optimizing such models. The most promising approach seems to be also the simple d-vector based one, extensively described in section 2.4.2. However, neural networks involved in examples reported in literature often presents architectures with too many layers or parameters, even in range of millions. Such complex systems are not feasible to be executed on a microcontroller, and a careful design is needed to achieve results in constrained systems. To support the neural network in solving the problem, different similarity algorithms have been tested to perform the comparison between d-vectors, but also cascading of other more powerful classification algorithms have been tried.

Unfortunately, neural network size is not the only concern. The presence of the keyword spotting system that already takes space and resources is also something to be considered, and the more application parts can be shared, the better it is for the memory and computational footprint of the system.

4.3.1. Feature extraction backbone

To maintain the compatibility with the keyword spotting system designed, and also to save memory in the final on-device implementation, we decided to maintain the same input data preprocessing pipeline of the keyword spotting system, extensively described in section 3.4. This means that all the moving parts in the speaker verification system take as input MFCC spectrograms of audio data. In addition to the efficiency gained, all the advantages related to data complexity that MFCC preprocessing brings to the system are still valid. However, we can't affirm the same regarding the optimality of such pipeline also for SV tasks. While MFCC spectrograms are also widely used in speaker verification systems, the pipeline adopted has been built specifically for the keyword spotting task, highlighting differences between spectrograms of different spoken words. There are no guarantees that such way of extracting MFCCs is optimal to perform also speaker verification.

4.3.2. Text-Dependent and Text-independent

As explained in 4.3.2, the choice of the text-independent or the text-dependent speaker verification task determines not only the availability of data, but also the general complexity of the task, with the text-independent condition being slightly more difficult to be solved. However, choosing one of them does not pose constraints on the design that exclude the other option from being implemented: a text-independent speaker verification system should in principle be able to perform also in a text-dependent way.

All the proposed solutions have been tested both on text-independent and on text-dependent datasets. An important consideration must be done: literature research showed that text-independent systems tend to perform better when longer audio utterances are submitted. In our case, we are limited by the length of the keyword spotting utterances that can be recognized, fixed at 1-second long audio samples due to memory constraints and compatibility reasons. This is an element that causes additional difficulties to the text-independent speaker verification, and must be considered when commenting final performances.

4.3.3. Final system structure

All the systems designed follow the general structure identified in section 2.4.2, and can be represented by the following scheme:

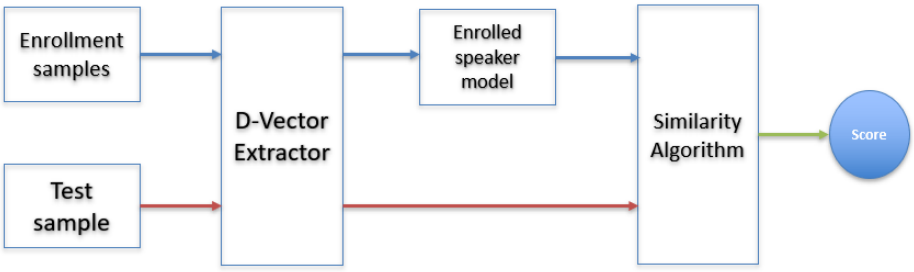


Figure 4.4: D-Vector speaker verification approach.

Enrollment samples: set of MFCC spectrograms coming from the enrolled speaker. Several sizes of the set have been tested to measure the impact of the number of samples on the final model accuracy.

Test sample: MFCC spectrogram of an audio sample coming from an unknown speaker. During testing, such samples may come both from the enrolled speaker and from another unknown speaker.

D-Vector Extractor: last hidden layer activation of a neural network trained as a classifier on a finite set of speakers. Its purpose is to obtain a unique d-vector for each speaker, with a certain consistency across different utterances.

Enrolled speaker model: representation of the speaker identity obtained by processing the d-vectors produced by the enrollment samples. It can be obtained as the element-wise average of the enrollment d-vectors or it can simply be the full set of d-vectors.

Similarity algorithm: algorithm that performs the confrontation between the speaker model and the d-vector extracted from an utterance of an unknown speaker. This confrontation can happen using various techniques and methods.

Score: output value that establish the belonging of the test sample to the enrolled speaker.

However, even if the scheme identifies a similarity algorithm in the block responsible for obtaining the score, also more complex classification algorithms that operates on d-vectors can be adopted. Such block is to be intended as the algorithm responsible for verifying the identity of the speaker, regardless of the method adopted. More details will be clarified in section 4.5.

4.4. Building the D-vector extractor

The first step in developing the speaker verification system, regardless of the similarity approach adopted, is building the neural network responsible for extracting d-vectors from input audio samples. It is the block called "D-Vector extractor" in figure 4.4. The most common approach is to train a neural network on a sufficiently large speakers dataset to perform classification among them. The proposed architecture is a convolutional neural network.

4.4.1. Convolutional Neural Network

The neural network adopted is deeper than the ones used in keyword spotting task. As it can be noticed from the scheme below, a batch notmalization layer has been introduced right after the input layer. The network presents the classical convolution-maxpooling blocks, ending with two convolutions followed by a dense layer. Dropout is added to control overfitting, and the last output layer is a softmax classifier with as many nodes as the number of speakers in the training dataset, which is 92 in the chosen implementation (refer to section 4.4.2 for details).

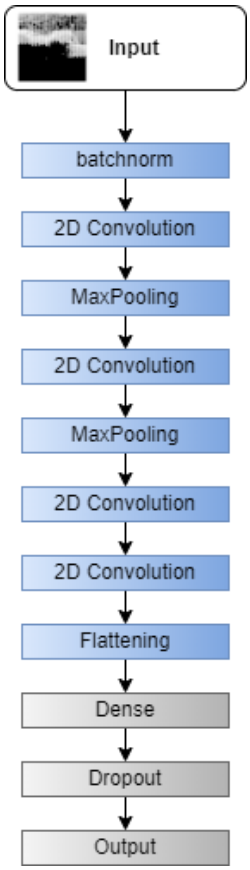


Figure 4.5: D-vector extractor CNN structure.

Details of the various NN layers are the following:

Model: speakers-classificator-cnn			
Layer type	**Output shape**	**Settings**	**Parameters**
Batch Normalization	(49, 40, 1)	None	4
2D Convolution	(49, 40, 8)	Filters = 8, Kernel = (3, 3) Strides = (1, 1)	80
Maxpool layer	(16, 13, 8)	Pool size = (3, 3) Padding = "valid"	0
2D Convolution	(16, 13, 16)	Filters = 16, Kernel = (3, 3) Strides = (1, 1)	1168
Maxpool layer	(8, 6, 16)	Pool size = (2, 2) Padding = "valid"	0
2D Convolution	(4, 3, 32)	Filters = 32, Kernel = (3, 3) Strides = (1, 1)	4640
2D Convolution	(2, 2, 64)	Filters = 64, Kernel = (3, 3) Strides = (2, 2)	18496
Flattening	(256)	None	0
Dense	(128)	None	32896
Dropout	(128)	Rate = 0.3	0
Output	(94)	Activation = Softmax	12126
Total parameters			69,410

Table 4.1: Speakers classificator CNN details.

In the scope of this thesis, the d-vector has been taken as the activation of the flattening layer, after all the convolutional blocks but before the first fully connected layer. In figure 4.5, the d-vector extractor is highlighted in blue.

The final d-vector extractor model is a neural network with a total of $24,388$ parameters, which make it extremely lightweight considering the complexity of the task to be solved. Details about memory consumption and inference time will be provided in 4.4.3. The size of the d-vectors is 256, equal to the output size of the flattening layer. This is slightly larger than other approaches in literature, but still small enough to fit several in the memory of a microcontroller.

4.4.2. Training of D-Vector extractor

The network has been trained on a subset of the LibriSpeech-train-100 dataset (refer to 2.4.4). The dataset has been downloaded and each audio sample has been divided into 1-second long segments to match the expected input size of the network. The dataset contains 251 different speakers, but a subset of them has been selected among those who had the greatest number of samples to speed up the training process. 92 speakers have been identified and the related samples constitued the subset, with a total of $136,651$ samples.

All the samples have been preprocessed with the same MFCC extraction pipeline explained in 3.4; however, no data augmentation has been performed this time because of the sufficient quantity.

Data have been split into training, validation and testing sets with proportion respectively of 70%, 15%, 15%, producing a total of $95,592$ samples for training, $20,485$ samples for validation and $20,484$ samples for testing.

The neural network proposed in the previous section was trained for a total of 700 epochs with a batch size of 32, and the Adam optimizer with a learning rate of 0.0001. Details on loss function adopted can be found in section B.1 of Appendix B.

At the end of the training, the neural network showed capabilities of classifying speakers in the set: the accuracy scored was 0.5778 on the testing set. The confusion matrix represents a concentration of values on the principal diagonal, and it is reported in figure 4.6:

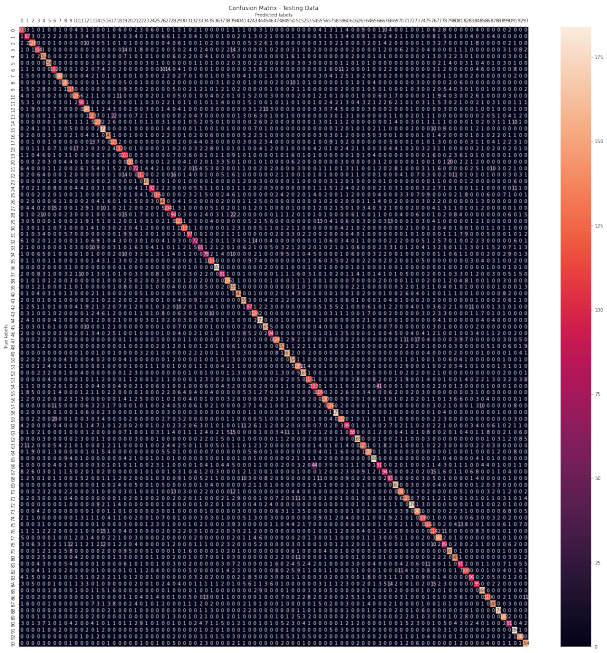


Figure 4.6: Confusion matrix of the speaker classificator model.

4.4.3. Memory and latency estimation

Memory and latency estimations for the d-vector extractor have been obtained via the ModusToolbox Machine Learning Configurator provided by Infineon Technologies, analogously to the estimation for the KWS models explained in 3.7. All the considerations related to flash memory estimation, RAM memory consumption and inference time are identical and still valid.

Model: d-vector-extractor-256			
Quantization	**Flash memory**	**Buffer size**	**Inference time**
float	98.08 kB	70.5 kB	0.036 s
int16x16	49.2 kB	43.1 kB	0.028 s
int16x8	24.8 kB	43.1 kB	0.032 s
int8x8	24.8 kB	25.5 kB	0.033 s

Table 4.2: Memory and latency estimations the d-vector extractor NN.

4.5. Similarity algorithms

The d-vector extractor backbone is just one part of the pipeline of SV. A fundamental role is played by the classification or similarity algorithm that has the burden of actually

verifying the claimed identity of a speaker, via the analysis of the produced d-vectors. This section will detail the approaches adopted in the design of such an algorithm, along with the proposed solutions.

4.5.1. Mean Cosine Similarity

This approach is the state-of-the-art regarding speaker verification systems. A pre-determined number of enrollment samples from the enrolled speaker is obtained, and is processed through the d-vector extractor to produce a set of d-vectors ("enrollment d-vectors"). The element-wise average of the d-vectors is computed to obtain a mean d-vector to be used as speaker model.

Every time a new audio utterance is detected, the system extracts the related D-Vector and confronts it with the mean D-Vector of the enrolled speaker via cosine similarity (equation 2.1).

If the similarity is above a certain threshold, the new audio utterance is attributed to the enrolled speaker. The process is represented by the scheme below:

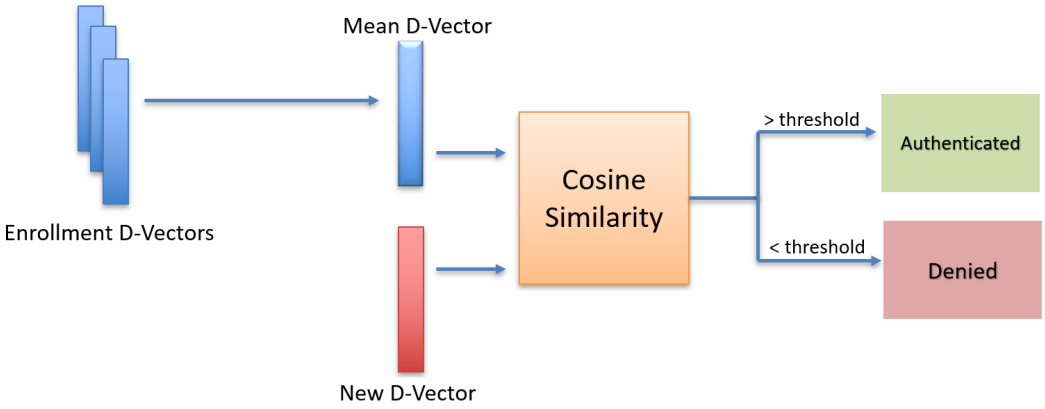


Figure 4.7: Speaker verification based on mean d-vectors and cosine similarity.

Formally, let $\boldsymbol{E}$ be a $M \times N$ matrix, $\boldsymbol{u}$ a vector of size N and t a real number such that:

- N is the size of the d-vector produced by the d-vector extractor;
- M is the number of enrollment samples chosen;
- $\boldsymbol{E}$ is the matrix composed by the M enrollment d-vectors;
- $\boldsymbol{u}$ is a d-vector coming from an utterance of an unknown speaker;
- $t \in [-1.0, +1.0]$ is the similarity threshold.

Matrix $\boldsymbol{E}$ has the following shape:

$$\boldsymbol{E} = \begin{bmatrix} e_{11} & e_{12} & e_{13} & \dots & e_{1N} \\ e_{21} & e_{22} & e_{23} & \dots & e_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_{M1} & e_{M2} & e_{M3} & \dots & e_{MN} \end{bmatrix} \tag{4.2}$$

Where e_{mn} is the n-th element of the m-th enrollment d-vector. To perform the mean cosine similarity method, there is the need of computing the mean vector $\boldsymbol{me}$:

$$\boldsymbol{me} = \begin{bmatrix} me_1 & me_2 & me_3 & \dots & me_N \end{bmatrix} \tag{4.3}$$

Where each element of $\boldsymbol{me}$ is computed as:

$$\boldsymbol{me_i} = \frac{\sum_{m=1}^{M} e_{mi}}{M} \tag{4.4}$$

Computing $\boldsymbol{me}$ ends the enrollment phase. When the d-vector $\boldsymbol{u}$ of an utterance coming from the unknown speaker must be processed, by relying on equation 2.1 we can take a decision, called *auth*, on the threshold:

$$auth = \begin{cases} 1, & \text{if } cossim(\boldsymbol{me}, \boldsymbol{u}) \geq t \\ 0, & \text{if } cossim(\boldsymbol{me}, \boldsymbol{u}) < t \end{cases} \tag{4.5}$$

Where $auth = 1$ means that we determined the unknown utterance $\boldsymbol{u}$ to belong to the enrolled speaker, and $auth = 0$ means that we determined such utterance to belong to an unknown speaker.

This approach provides several advantages if considered in a TinyML context:

Low latency: this system can provide a confidence value on the identity only by computing the cosine similarity between two vectors, which is a computation that can be carried out fast;

Low memory footprint: to store the speaker model it is required only as much space as the size of a single d-vector, and even for the computation of such mean vector only space equal to the number of d-vectors times the size of the d-vector is needed.

Threshold tuning: the threshold can be manually tuned to trade-off between false positives and false negatives.

4.5.2. Best-Match Cosine Similarity

Similarly to the previous method, a pre-determined number of enrollment samples from the enrolled speaker is used to produce a set of d-vectors. This set is kept in memory and is called enrollment set. During inference, every time a new audio utterance is detected, the system extracts the related d-vector and compares it with each d-vector in the enrollment set via cosine similarity. The system returns the best-matching similarity, i.e. the highest similarity scored during the one-to-one comparison. If the similarity is above a certain threshold, the utterance is attributed to the enrolled speaker:

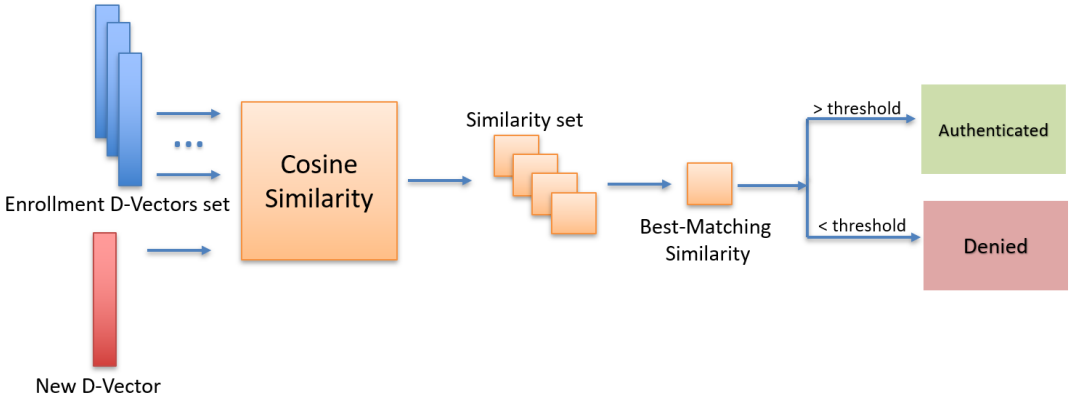


Figure 4.8: Speaker verification based on one-to-one confrontation on an enrollment set of d-vectors.

Formally, as in previous case, let $\boldsymbol{E}$ be a $M \times N$ matrix, $\boldsymbol{u}$ a vector of size N and t a real number such that:

- N is the size of the d-vectors produced by the d-vector extractor;
- M is the number of enrollment samples chosen;
- $\boldsymbol{E}$ is the matrix composed by the M enrollment d-vectors;
- $\boldsymbol{u}$ is a d-vector coming from an utterance of an unknown speaker;
- $t \in [-1.0, +1.0]$ is the similarity threshold.

Matrix $\boldsymbol{E}$ has the same shape of equation 4.2. For notation purposes, we will refer to $\boldsymbol{e_m}$ as the m-th row of matrix $\boldsymbol{E}$, representing the m-th d-vector provided by the enrolled speaker.

Differently from the previous approach, the enrollment phase ends when matrix $\boldsymbol{E}$ is available to the system. Each time a new d-vector $\boldsymbol{u}$ coming from an unknown speaker's

utterance is submitted to the verification system, a similarity vector $\boldsymbol{sm}$ of size M is computed:

$$\boldsymbol{sm} = \begin{bmatrix} sm_1 & sm_2 & sm_3 & \dots & sm_M \end{bmatrix} \tag{4.6}$$

Where each element sm_m is computed as the cosine similarity between the unknown speaker's d-vector $\boldsymbol{u}$ and the m-th enrolled speaker d-vector stored in matrix $\boldsymbol{E}$:

$$sm_m = cossim(\boldsymbol{e_m}, \boldsymbol{u}) \tag{4.7}$$

The vector $\boldsymbol{sm}$ can be called similarity vector or similarity set. Then, the similarity value sim to be compared with the threshold t to take a decision on the identity is extracted from the similarity set as:

$$sim = max(sm_1, sm_2, \dots, sm_M) \tag{4.8}$$

And as in previous approach, final decision value $auth$ can be determined as:

$$auth = \begin{cases} 1, & \text{if } sim \geq t \\ 0, & \text{if } sim < t \end{cases} \tag{4.9}$$

Where $auth = 1$ means that the utterance $\boldsymbol{u}$ has been associated to the enrolled speaker.

The pseudo-algorithm for the method is the following:

Algorithm 4.1 Best-Matching Cosine Similarity

```
variables: E[M][N], u[N], sm[M], t
for n ∈ [0, N − 1] do
    sm[i] = cossim(E[i], u)
end for
sim = max(sm)
if sim ≥ t then
    auth = 1
end if
if sim < t then
    auth = 0
end if
return auth
```

All the advantages regarding computational cost and memory footprint of the previous method are maintained. However, this approach could bring performance improvement if the context is the one-class few-shot classification we are operating in. The previous approach can work really well if the d-vector extractor model has great capabilities of producing fairly constant d-vectors from utterances coming from the same speaker; this can be surely achieved with neural networks, but it is not trivial to do if there are constraints on number of parameters and layers that can be used, which is the exact condition that TinyML enforces on designers. Let's clarify this with an example: let's take $\boldsymbol{P}$ as the enrollment d-vectors of a speaker sp_1. For simplicity, assume that 6 d-vectors have been used as enrollment samples:

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p_1} & \boldsymbol{p_2} & \boldsymbol{p_3} & \boldsymbol{p_4} & \boldsymbol{p_5} & \boldsymbol{p_6} \end{bmatrix} \tag{4.10}$$

Assume that the couple (p_1, p_2) comes from a higher pitched speech from speaker sp_1, and that (p_3, p_4) comes from a lower pitched speech. It is reasonable to assume that the similarity between the elements of the same couple is higher than similarity between elements not belonging to the same couple, with a maximization of this difference between elements from the two aforementioned sets. The production of an average d-vector could attenuate the utterance-dependent characteristics embedded in each d-vector, leading to lower similarities when a new utterance from sp_1 is submitted to the system. This can happen in particular if the new enrollment is still slightly higher or lower. Performing a one-to-one confrontation, instead, allows per-utterance differences to be taken more into

account, thus compensating for the d-vector extractor performance.

Another advantage of this method is the possibility of modifying at run time the enrollment d-vectors set, providing a fast adaptation to transitory conditions that change features of a speaker's voice, such as colds or hoarseness. It would be sufficient to provide just a few new voice samples to insert in the enrollment samples set, thus adapting the system to the new condition. With the previous method this case could not be faced unless a complete collection of enough enrollment samples to compute the mean d-vector is performed; moreover, once the transitory condition passes, a new enrollment must be performed again, to re-calibrate the system on the normal voice condition.

All of the aforementioned advantages come to the cost of having slightly more computational burden: instead of computing one single cosine similarity, each inference requires computing M cosine similarities. However, the few-shot condition implies having a relatively low number of samples in the enrolled d-vector set, so this increased computational complexity didn't lead to a perceivable reduction in inference speed.

4.5.3. One Class Neural Network

This is a classification method that involves training a one-class Neural Network (OCNN) to distinguish between d-vectors produced by the authenticated speaker and by other unknown speakers. During inference, the one-class neural network must provide a classification of each new input:

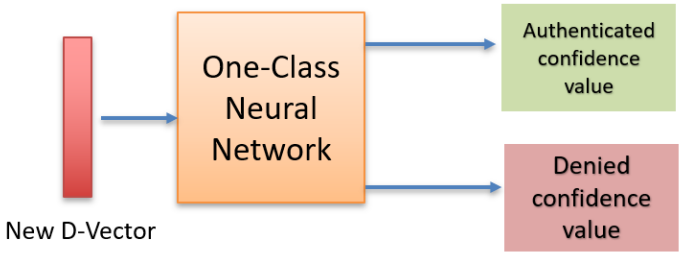


Figure 4.9: Speaker verification based on one-class neural networks.

This approach has been inspired by the work done in [35]. Since the model is one-class, no other speakers are involved in the process of training the network. A binary one-class neural network must however be trained by using positive samples and negative samples, like any other classification neural network. While in this case positive samples are the enrollment d-vectors, it is more difficult to identify what can be used as negative samples.

A solution has been proposed by [35] that relies on using randomly generated data drawn from a Gaussian distribution as negative d-vectors. In particular, during training the one-class neural network processes a batch of data in the following way:

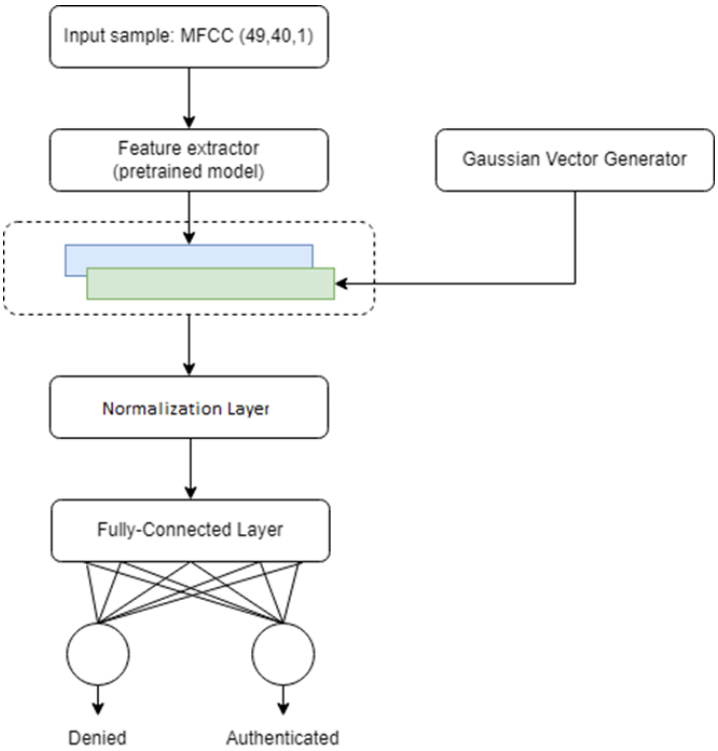


Figure 4.10: Processing of a batch of data in OCNN training.

In this figure, the feature extractor is the d-vector extractor trained as explained in section 4.4.2. The batch of training data that is processed at each training step is represented in the dotted box, and as it can be noticed it is composed by d-vectors of the authenticated speaker (blue) and fictitious d-vectors representing other speakers (green).

In particular, the process we adopted for generating the fictitious negative d-vectors is the following. Let $\boldsymbol{g}$ be a random vector of size N:

$$\boldsymbol{g} = \begin{bmatrix} g_1 & g_2 & \dots & g_n \end{bmatrix} \tag{4.11}$$

Each element of $\boldsymbol{g}$ is generated by sampling from a Gaussian distribution:

$$g_i \sim \mathcal{N}(\mu, \sigma^2) \tag{4.12}$$

Where $\mu = 0$ and $\sigma^2 = 0.1$. Each fictitious d-vector $\boldsymbol{j}$ of size N is generated starting from the random vector $\boldsymbol{g}$ and a real d-vector $\boldsymbol{d}$ as:

$$\boldsymbol{j} = sh(\boldsymbol{g} \odot \boldsymbol{d}) \tag{4.13}$$

Where $\odot$ indicates the element-wise product of two vectors, and the function $sh(\boldsymbol{x})$ is a function that performs a random shuffling of the values of the vector $\boldsymbol{x}$.

Intuitively, the process of generating a fictitious d-vector starts from an authenticated d-vector, which is then masked with a random vector, whose elements are taken from a Gaussian distribution with mean 0 and variance 0.1. The result is then shuffled element-wise, and is taken as fictitious d-vector. What is happening is that a fictitious d-vector is obtained by randomly scaling and shuffling values of an authenticated d-vector. More details on the motivations behind this choice can be found in section B.2 of Appendix B.

As it can be noticed from figure 4.10, each batch of training data is composed by a certain number of real d-vectors coming from the authenticated speaker, and an equal number of fictitious d-vectors obtained by applying the aforementioned process to the real embeddings. In a few-shot classification context, this leads to very limited training sets.

A possible advantage of using a one-class neural network is that such an algorithm, being a more powerful classifier than the simple methods based on cosine similarity, could learn to map d-vectors produced by a tiny feature extractor to the correct speaker in a more efficient way.

However, this approach requires an increased computational burden on target hardware: the full training process, along with the random d-vectors generation, must be executed directly on-device. This poses serious limitations on the one-class neural network architecture that can be built, both regarding size and type of the adopted layer. This approach is feasible thanks to recent advances in research, that identified methods for performing training on-device. Details on the structure of the one-class neural network we adopted are the following:

Model: sv-oneclass-classifier			
Layer type	**Output shape**	**Settings**	**Parameters**
Batch normalization	(256)	None	1,024
Dense	(256)	Units = 256 Activation = relu	65,792
Dense	(2)	Units = 2 Activation = sigmoid	514
Total parameters:			67,330

Table 4.3: OCNN architecture details.

4.5.4. One-Class SVM

This method involves training a one-class Support Vector Machine (OC-SVM) [46] to learn a boundary that separates efficiently d-vectors of the enrolled speaker from d-vectors of other speakers. The approach is similar to the One-Class Neural Network: a set of enrollment d-vectors is used for fitting the OC-SVM, which is then used as a discriminator on new samples.

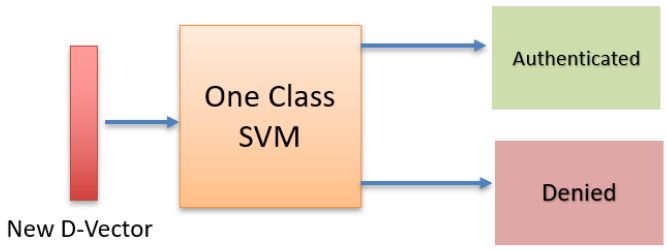


Figure 4.11: Approach of SV with one-class SVMs.

Classical Support Vector Machines (SVM) are widely used machine learning algorithm in classification tasks [8]. Formally, let's consider a set of data in a multi-dimensional space:

$$\Omega = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \tag{4.14}$$

Where x_i are data points belonging to a d-dimensional space ($x_i \in \mathbb{R}^d$) and y_i are class labels associated to each data point $y_i \in \{-1, 1\}$. This example refers to binary classification. SVM work by creating a non-linear decision boundary by projecting data through

a non-linear function $\phi : I \mapsto F$ to a higher-dimensional space. If we call I the initial dimensional space, SVM are able to project data in a different dimensional space F where is easier to find a hyperplane that is able to separate the data. The separating hyperplane is computed by maximizing its distance from samples belonging to different classes by applying optimization algorithms such as Sequential Minimal Optimization (SMO). The optimization process involves computing Lagrange multipliers α_n, one for each sample in the training dataset. At the end, some Lagrange multipliers will be equal to zero, and others will be different from zero. The classification on an unseen data point x is performed by the SVM using the following relation:

$$f(x) = \text{sgn}(\sum_{i=1}^{n} \alpha_i y_i K(x, x_i) + b) \tag{4.15}$$

Since only samples with the associate $\alpha \neq 0$ contribute to classify the new unseen sample, such samples are called Support Vectors. They are the samples that lay on the margin found by the optimization algorithm. In equation 4.15, there is also the term $K(x, x_i)$. This is called kernel function, and is fundamental to express the feature transformation ϕ that is applied to the samples:

$$K(x, x_i) = \phi(x)^T \phi(x_i) \tag{4.16}$$

Finding ϕ is a non-trivial process. However, there is a procedure called kernel trick that allows computing function $K(x, x_i)$ without knowing the explicit formulation of ϕ, as long as the result produced is the same. There are established kernel functions that corresponds to different projections in higher-dimensional feature spaces. A notable kernel function is the Gaussian kernel, also called RBF kernel, defined as:

$$K(x, x_i) = \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2}\right) \tag{4.17}$$

It depends on the squared euclidean distance $\|x - x_i\|^2$ and on the parameter σ. It is important because it corresponds to a projection ϕ on an infinite-dimensional feature space, without having to compute the mapping explicitly.

Another widespread kernel is the linear kernel, defined as:

$$K(x, x_i) = x^T x_i + c \tag{4.18}$$

The advantage of the linear kernel is that it only consists in the the inner product of x and x_i with an optional constant term c. This does not make it actually a projection in a higher-dimensional feature space, and it is typically used on data sets with large amounts of features: increasing the dimensionality of such data does not necessarily improve separability. Another advantage is that the computation required is easy, thus making this kernel particularly suited for a TinyML approach, where the computation must be carried out by a MCU.

The intuition behind a one-class SVM as explained in [46] is to perform a separation of all available data points in the feature space F by maximizing the distance of the hyperplane from the origin. This produces a binary classification function which captures portions of the input space where the probability density of the data lives. Such function returns $+1$ in an enclosed region (capturing the training data points) and -1 elsewhere. If we take $w^T x + b = 0$ as the formulation of the hyperplane, with $w \in F$ and $b \in \mathbb{R}$, its parameters can be computed by minimizing the following objective function:

$$\min_{w,\xi,\rho} \frac{1}{2}\|w\|^2 + \frac{1}{\nu n}\sum_{i=1}^{n}\xi_i - \rho \tag{4.19}$$

$$\text{subject to: } \begin{cases} y_i(w^T\phi(x_i) + b) \geq \rho - \xi_i & \text{for all } i = 1, \ldots, n \\ \xi_i \geq 0 & \text{for all } i = 1, \ldots, n \end{cases}$$

Where ξ_i are the so-called "slack variables": added to allow some samples to lie within the margin, they express the entity and cost of such violation. They are variables to be optimized along with w and ρ. It is important to notice the presence of parameter v in this formulation, which represents both an upper bound to the number of outliers, i.e. training examples belonging to the authenticated class but not inserted in the related space region, and a lower bound of the number of training samples used as Support Vectors. It is a parameter subject to hyperparameter search, so to be tuned by the designer.

Using Lagrange techniques, the solution to the optimization problem as in previous case leads to identifying one Lagrange multiplier α for each element of the training set, and the final classification function of the One-Class SVM is the following:

$$f(x) = \text{sgn}((w \cdot \phi(x_i)) - \rho) = \text{sgn}(\sum_{i=1}^{n} \alpha_i K(x, x_i) - \rho) \tag{4.20}$$

In the context of speaker verification, the OCSVM is fitted on the enrollment d-vectors provided by the enrolled speaker to try to learn a boundary that contains such samples

in the feature space:

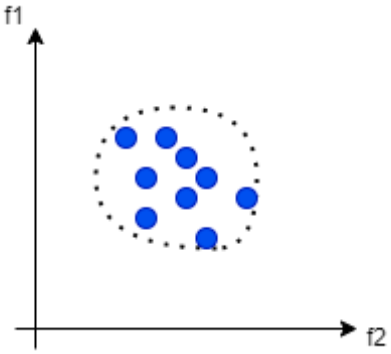


Figure 4.12: Example of learned boundary of a OCSVM in a 2-dimensional feature space.

If a new d-vector is sumbitted to the system, the identity of the speaker is assumed according to 4.20. This method does not provide a confidence value, but similarly to the OCNN approach provides a direct classification of the new input speech sample.

Using a SVM without having to manually write optimization procedures is easy, thanks to libraries that already implement the algorithm described by [46], such as the object `svm.OneClassSVM` from the `scikit-learn` Python library [37].

Fitting a one-class SVM requires however heavy computation, in particular for solving the constrained optimization problem. While this may not be a problem on powerful computers, in a TinyML context is something that must be taken into account. Even if libraries for performing light Sequential Minimal Optimization exist [16], they would require a complete porting on embedded devices. Research at time of writing of this document was not able to identify an implementation for fitting a one-class SVM directly on a microcontroller. Moreover, if the optimization found a considerable number of support vectors, there may be problem on fitting the algorithm in the limited memory of the MCU.

Details on fitting the OCSVM and the hyperparameter search will be given in section 5.1.

5 | Speaker verification testing results

This chapter will detail the approach to testing the similarity methods identified in chapter 4, reporting obtained results. After a general description of the experimental setup, providing details about datasets and evaluation metrics, two subsections will be dedicated to reporting the results. A section commenting such results will close the chapter. More information regardin approach-specific considerations can be found in section B.3 of Appendix B.

5.1. Experimental setup

All the approaches for solving the SV problem in TinyML context have been defined and explained, and we will now describe the experiments that have been done to test the proposed solutions. First and most important part is the choice of the dataset to perform the tests on. Since text-dependent datasets are hard to obtain, we decided to perform a collection of a text-dependent dataset suitable for testing our approaches. Four speakers have been asked to collect text-dependent samples, from which 1-second long utterances containing speech have been extracted. We also collected a custom text-independent dataset from the same people, to have testing benchmarks on both applications of speaker verification on the same subset of speakers. Each approach proposed has been tested against the two different datasets. Such datasets have been collected directly leveraging the target hardware available.

The experiments were designed to test the models on one-class few-shot conditions that have been explained in section 4.2. The one-class condition has been created by enrolling one speaker at a time and using only samples from that speaker to perform the enrollment. Each speaker in the datasets has played the part of the enrolled speaker, and each time samples from other speakers have been used to build the set of "unknown" speakers. The few-shot conditions have been tested by posing a limitation on the number of enrollment samples to be used. Reasonable samples number for few-shot classification were deemed

being 8 and 16 samples, but we decided to identify also two corner cases, i.e. the case with only 1 enrollment sample and the case with 64 enrollment samples. So, to recap the conditions of the testing environment the following table can be referred:

Test parameter	**Values**
Dataset	Text-Independent, Text-Dependent
Classification Method	Mean Cosine Similarity, Best-Match Cosine Similarity, One-Class NN, One-Class SVM
Enrollment samples	1, 8, 16, 64
Enrolled speaker ID	0, 1, 2, 3

Table 5.1: Testing conditions for speaker verification.

Every possible combination of the parameters reported has been tested, bringing a total of 128 different test cases. The following subsections will provide details on the datasets used during the experiments, followed by an example of a test and approach-dependent considerations. Finally, a description of the evaluation metrics adopted will be provided.

5.1.1. Text-dependent custom dataset

A collection 1-second long utterances containing the word «Hey Cypress». This locution has been chosen because it contains distinct phonemes, and is also long enough to accentuate differences among speakers while at the same time fitting in a one second long window.

Text-dependent dataset		
Speaker ID	**Gender**	**Number of samples**
0	M	100
1	M	100
2	M	100
3	F	100
Total samples:		400

Table 5.2: Text-dependent custom dataset details.

It is worth noting that a manual alignment of such samples has been performed to center the «Hey Cypress» locution in the middle of the 1-second audio window. This has not been

done by leveraging algorithms of any kind, but via visual inspection of the waveform. From such samples, MFCC spectrograms have been obtained by applying the preprocessing pipeline explained in section 3.4.

Training, testing and validation sets have been extracted from this dataset. However, since we wanted to compare performances of different models, we decided to use the same data for training, testing and validating each model. This is the following approach adopted for building such datasets:

1. 15 samples from each speaker have been taken to produce a testing dataset composed by 60 samples;
2. 15 samples from each speaker have been taken to produce a validation dataset composed by 60 samples;
3. 64 samples from each speaker have been taken for a successive build of training datasets.

Training data have been produced in the following way. Starting from the collection of 64 samples coming from a speaker:

sample 0 : taken to build the enrolled dataset composed by 1 sample;

samples [0-7]: taken to build the enrolled dataset composed by 8 samples;

samples [0-15]: taken to build the enrolled dataset composed by 16 samples;

samples [0-63]: taken to build the enrolled dataset composed by 64 samples.

To make the distinction more clear, if we want to test the performance of model A on speaker 0 using 16 enrollment sample, we would use as training set the set of samples with 16 elements from speaker 0, and as testing and validation sets the two sets with 60 samples each. The same data can be used to test model B.

This means that, once a speaker and a number of enrollment samples is chosen, each model is trained, tested and validated on the exact same data.

5.1.2. Text-independent custom dataset

A collection of text-independent 1-second long utterances resulting from the reading of Italian text.

Text-independent dataset		
Speaker ID	**Gender**	**Number of samples**
0	M	100
1	M	100
2	M	100
3	F	100
Total samples:		400

Table 5.3: Text-independent custom dataset details.

The processing applied to this dataset is the same used for the text-dependent dataset in section 5.1.1, with the only difference that no audio centering was needed in this case. Where possible, audio samples with too much silence were discarded, but this has not been applied organically to the whole dataset.

The division in training, testing and validation sets is completely analogous to the one explained in section 5.1.1.

All the datasets used in this thesis work contain English speech, but the text-independent custom datset is in Italian. The d-vector extractor itself has been trained on English audio data, and research in speech processing showed that language coherence plays a fundamental role in final performance, especially regarding speaker verification tasks [27].

5.1.3. Example of a test case

To clarify the way a test is conducted, let's take as an example the case with:

Dataset: text-independent

Enrolled speaker ID: 0

Enrollment samples: 8

To test such conditions, we need to take as training dataset the set of 8 samples taken from speaker 0 in the text-independent dataset. Validation and testing sets do not depend on the speaker or the number of enrollment samples, but are general and common to all the text-independent tests. Such datasets are taken to conduct the experiments. Then, each similarity method described in section 4.5 is tested according the following methodology:

Mean Cosine Similarity : the enrollment samples from training set are used to produce a mean d-vector. Such mean d-vector is used as speaker model. The validation

set is used to find the best threshold for the algorithm, and the final performance is measured on testing set.

Best-Match Cosine Similarity : the enrollment samples from training set are used to compute the set of enrolled d-vectors. The validation set is used to find the best threshold for the algorithm, similarly to the previous method. The final performance is measured on the testing set with the threshold found.

One-Class Neural Network : The OCNN is trained on the training set. During training, at each sample from the training set, a fictitious d-vector is associated to model the negative samples, representing unknown speakers. Validation set is used in training to control overfitting and stop training. Final performance is measured on the testing set.

One-Class SVM : Different OCSVMs are fitted on the training set, each one with a different hyperparameters combination. The best performing OCSVM on the validation set is chosen to be tested on the testing set. Performance on such test is reported.

5.1.4. Evaluation metrics

Testing datasets adopted are composed by 60 samples, where 15 of them belong to the enrolled speaker and 45 of them belong to other speakers. This means that a classifier that always predicts the negative class (i.e. speaker different from the enrolled one) would obtain an accuracy of $A_{neg} = \frac{45}{60} = 0.75$. This is why we decided to adopt F1 score as the standard evaluation metric for all the models. Accuracy for all the solutions is also reported, even if it has not been the primary metrics for performing model selection. For example, in hyperparameter search of OC-SVM (refer to B.3 for details), we chose the hyperparameter combinations that provided the highest F1-score, even if this translated into lower accuracy values.

The following table reports the metrics measured for each proposed approach:

Approach	**Principal metrics**	**Other metrics**
Mean Cosine Similarity	**F1-score, accuracy**	ROC, AUC, EER
Best-Match Cosine Similarity	**F1-score, accuracy**	ROC, AUC, EER
One Class Neural Network	**F1-score, accuracy**	none
One Class SVM	**F1-score, accuracy**	none

Table 5.4: Metrics adopted for each proposed approach.

5.2. Text-Independent Dataset testing results

This section will report the testing results for the Text-Independent custom dataset explained in 5.1.2. Data presented here have been aggregated by computing the per-speaker mean of metrics reported. The full testing results, with results of per-speaker experiments, can be consulted in Appendix A referring to table A.2.

Let's start analyzing testing results for the proposed approaches in the text-independent context. As already explained, this is the most difficult speaker verification task among the ones considered in this work, not only due to the variability in phonemes that can be submitted to the system, but also because there is a language mismatch between the data used for training the d-vector extractor (refer to section 4.4.2 for details) and the data used for testing.

Grouping by number of enrollment samples - TI			
Enrollment Samples	**Classifier Method**	**Accuracy**	**F1-Score**
1	**Mean Cosine Similarity**	**0,5315**	**0,35375**
1	**Best-Match Cosine Similarity**	**0,5315**	**0,35375**
1	OC-NN	0,6675	0,21775
1	OC-SVM	0,75	0
8	**Mean Cosine Similarity**	**0,6325**	**0,4725**
8	Best-Match Cosine Similarity	0,6275	0,4265
8	OC-NN	0,6025	0,3075
8	OC-SVM	0,47875	0,362
16	**Mean Cosine Similarity**	**0,633**	**0,4645**
16	Best-Match Cosine Similarity	0,6075	0,45425
16	OC-NN	0,5015	0,34775
16	OC-SVM	0,494	0,38425
64	**Mean Cosine Similarity**	**0,649**	**0,45375**
64	Best-Match Cosine Similarity	0,55	0,38875
64	OC-NN	0,26775	0,39775
64	OC-SVM	0,54975	0,3625

Table 5.5: Testing results on TI dataset (grouped by number of enrollment samples)

Table 5.5 is useful to compare the performance among classifiers with the same number of samples. The first thing to be noticed is that by looking at F1-scores obtained by the algorithms, the best performing method is the one based on Mean Cosine Similarity with

8 enrollment samples, with F1-score = 0.4725. Figure 5.1 reports the results:

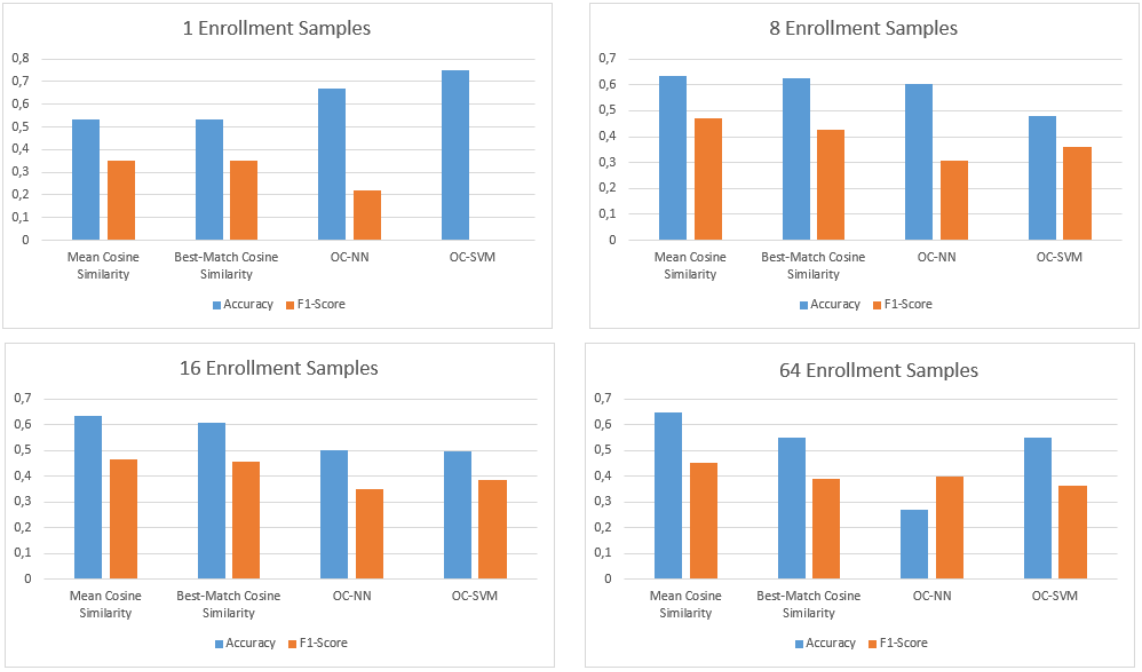


Figure 5.1: Testing results on TI dataset (grouped by number of enrollment samples).

We can notice that OC-SVM possess no relevant classification ability in the extreme case with 1 enrollment sample: this is expected since it is not possible for such algorithm to learn an effective boundary if there is only one sample in the dataset. With the increase of the number of enrollment samples, all the methods show signs of improvement, with the OC-SVM and One-Class Neural Networks showing the great difference. However, it is easy to see that no method outperforms cosine-similarity based classification, regardless of training samples size. It is important to notice that. in the corner case with only 1 enrollment sample, the behavior of the two cosine similarity methods is the same, since there is no actual possible mean to be computed and this translates into a 1:1 comparison among the enrollment d-vector and the unknown sample.

It is also useful to highlight the performance of each classifier method according to the number of samples, in order to analyze the per-classifier variation in performance with respect to the increase of number of enrollment samples, as figure 5.2 reports:

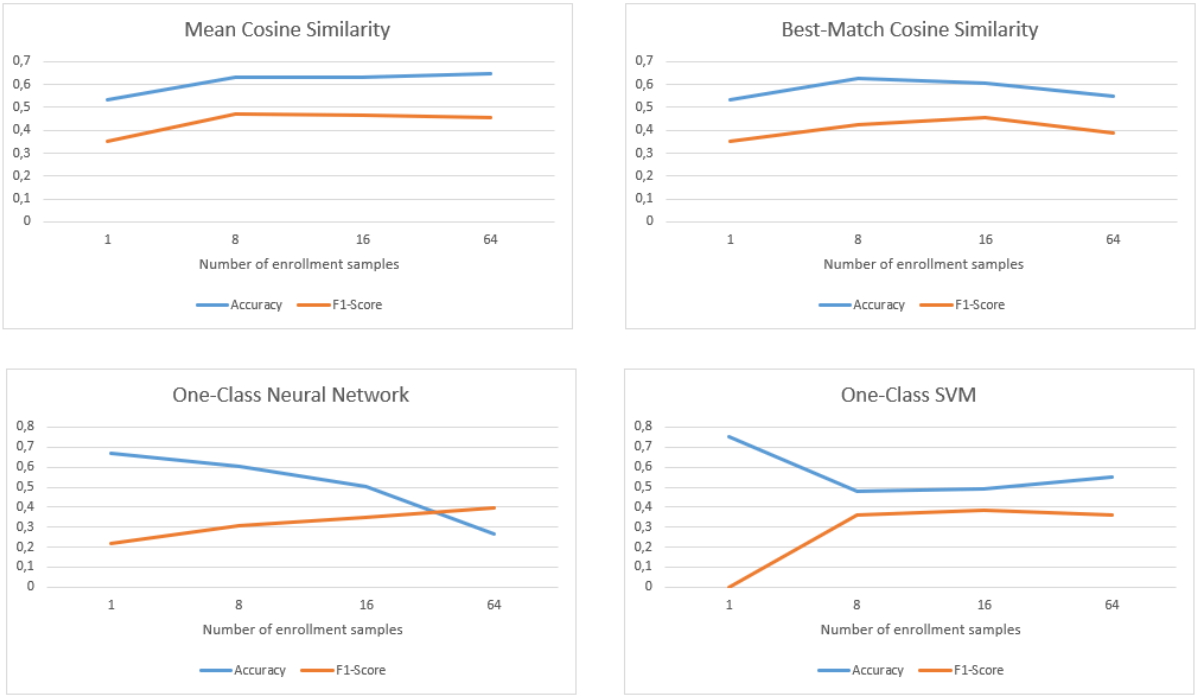


Figure 5.2: Testing results on TI dataset (grouped by similarity method).

Different approaches behave differently according to the number of enrollment samples. As it can be noticed by the plots, OCNN and OC-SVM see an improvement when the number of enrollment samples is increased, but mean cosine-similarity based methods tend to a stabilization in their accuracy the more enrollment samples are provided. Best-Match cosine similarity seems to suffer from the increase of enrollment samples: this is expected, because the more enrollment samples are there, the more is likely to find a pair of highly similar d-vectors belonging to different speakers.

5.3. Text-Dependent Dataset testing results

This section will report the testing results for the Text-Dependent custom dataset explained in 5.1.1. Data presented here have been aggregated by computing the per-speaker mean of metrics reported. The full testing results, without aggregation, can be consulted in Appendix A referring to table A.3.

Regarding Text-Dependent speaker verification, a higher performance of the methods is expected, due to the reduced phoneme variability of the problem, but also because the chosen word is in English. The following table, as in previous case, will report performance comparison among different classifiers grouping results by the number of samples:

Grouping by number of enrollment samples - TD			
Enrollment Samples	**Classifier Method**	**Accuracy**	**F1-Score**
1	**Mean Cosine Similarity**	**0,773**	**0,6395**
1	**Best-Match Cosine Similarity**	**0,773**	**0,6395**
1	OC-NN	0,737	0,0575
1	OC-SVM	0,75	0
8	Mean Cosine Similarity	0,81575	0,703
8	**Best-Match Cosine Similarity**	**0,858**	**0,742**
8	OC-NN	0,733	0,3395
8	OC-SVM	0,60775	0,4295
16	Mean Cosine Similarity	0,82475	0,72475
16	**Best-Match Cosine Similarity**	**0,93325**	**0,878**
16	OC-NN	0,683	0,46825
16	OC-SVM	0,633	0,44625
64	Mean Cosine Similarity	0,77075	0,66275
64	**Best-Match Cosine Similarity**	**0,93725**	**0,8775**
64	OC-NN	0,38725	0,43825
64	OC-SVM	0,82475	0,508

Table 5.6: Testing results on TD dataset (grouped by number of enrollment samples)

Table 5.6 shows much better results than what was obtained in a Text-Independent context. The best performing method has been the one based on Best-Match Cosine Similarity, that with 16 enrollment samples obtained a mean F1-score among speakers equal to 0.878, scoring an accuracy higher than 93%. This is a great result, considering the small footprint of algorithms involved.

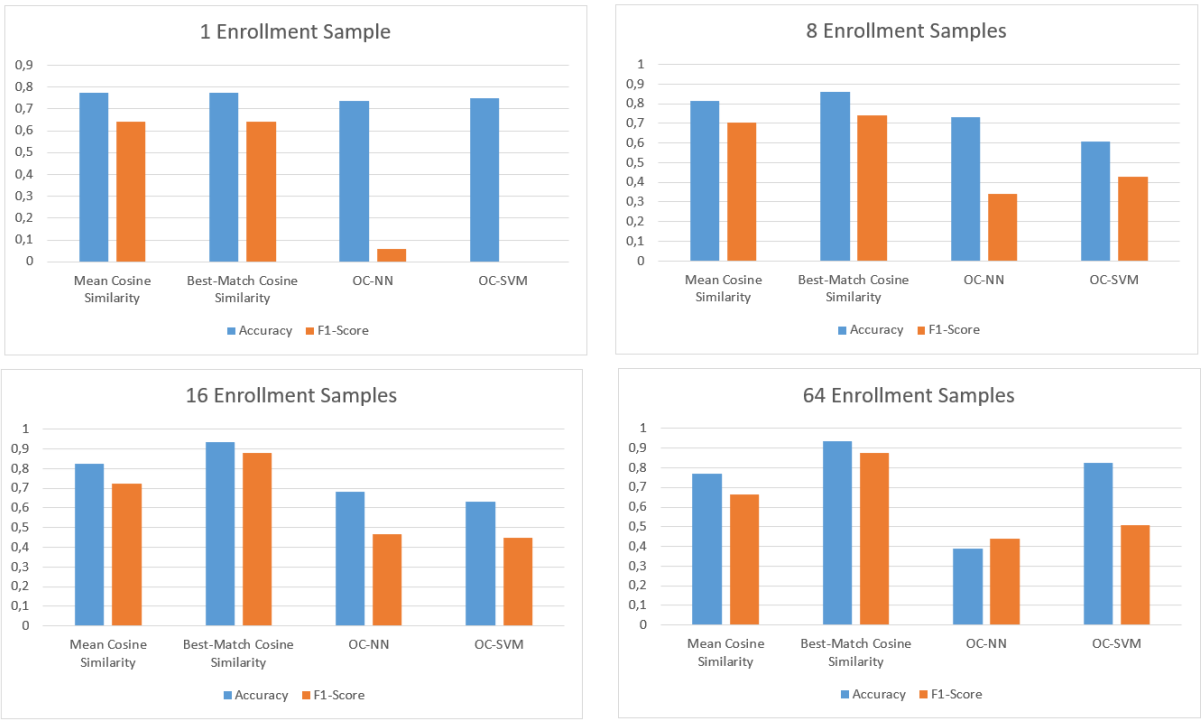


Figure 5.3: Testing results on TD dataset (grouped by number of enrollment samples).

Looking at the plots in figure 5.3, we can notice that cosine similarity based methods again outperform OC-NN and OC-SVM classifiers. Performance of the latter models are almost equal, with the OC-SVM starting as the worse-performing model in the corner case with 1 enrollment sample, but outperforming OC-NNs in the opposite case with 64 enrollment samples.

It is again useful to highlight the improvement of each method according to changes in the number of enrollment samples, as plots in figure 5.4 shows:

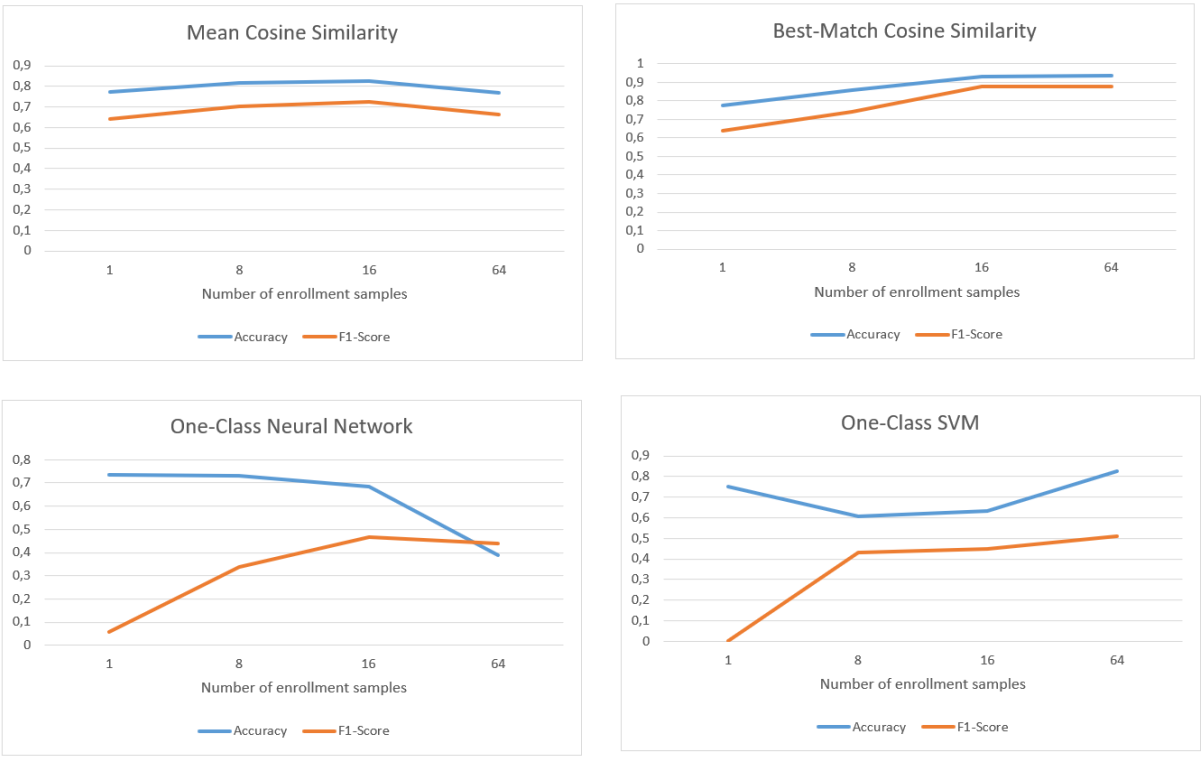


Figure 5.4: Testing results on TD dataset (grouped by number of enrollment samples).

All the methods except Best-Match Cosine Similarity show sign of improvement when the number of enrollment samples is increased. The reason of such decrease is analogous to the one identified for the text-independent case: increasing the number of samples increases the chance of a 1:1 confrontation that gives a high similarity value.

5.4. EER and AUC measurement

EER and AUC metrics are useful because allow comparing performance of our solutions against other systems in literature. As already anticipated, only approaches based on the choice of a threshold can provide EER and AUC metrics, so this section will report such measurements only for cosine-similarity based approaches. Results will be reported comparing the two approaches both in the text-independent and in the text-dependent context. As before, such values are computed by taking the mean value across different speakers.

Text-Independent - Mean Cosine Similarity		
Enrollment samples	**EER**	**AUC**
1	0,5205	0,5425
8	0,40425	0,62
16	0,372	0,64
64	0,35425	0,6425

Table 5.7: EER and AUC for Mean Cosine Similarity in Text-Independent context.

Text-Independent - Best Match Cosine Similarity		
Enrollment samples	**EER**	**AUC**
1	0,5205	0,5425
8	0,39975	0,66
16	0,39925	0,655
64	0,3495	0,6475

Table 5.8: EER and AUC for Best Match Cosine Similarity in Text-Independent context.

As it can be noticed, in a Text-Independent context the two methods tend to perform almost equally, with a slightly better performance for Best-Match Cosine Similarity regarding the AUC metric. However, the EER measurement does not show great improvements, so in a text-independent fashion the choice between one or the other algorithm can simply be dictated by computational needs; at inference time, Mean Cosine Similarity requires computing only one similarity instead of one for each enrollment d-vector stored.

More interesting results have been obtained by measuring performance on the text-dependent task, as tables 5.9 and 5.10 report:

Text-Dependent - Mean Cosine Similarity		
Enrollment samples	**EER**	**AUC**
1	0,22175	0,83
8	0,216	0,8775
16	0,209	0,89
64	0,255	0,845

Table 5.9: EER and AUC for Mean Cosine Similarity in Text-Dependent context.

Text-Dependent - Best Match Cosine Similarity		
Enrollment samples	**EER**	**AUC**
1	0,22175	0,83
8	0,095	0,94
16	0,0725	0,9625
64	0,085	0,965

Table 5.10: EER and AUC for Best Match Cosine Similarity in Text-Dependent context.

In a text-dependent context, the Best-Match Cosine Similarity method outperforms the Mean Cosine Similarity method. It is interesting to notice that 16 enrollment samples seems to be the optimal number among the one tested in this thesis work: in a real implementation, it is feasible to obtain 16 enrollment samples from the authenticated user, making this value also well-suited for the development of a demo application. In general, preferring the Best-Match cosine similarity method instead of Mean Cosine Similarity allows to obtain a mean improvement of 14.25% in EER and 8.5% in AUC:

Enrollment samples	EER improvement	AUC improvement
8	0,121	0,0625
16	0,1365	0,0725
64	0,17	0,12

Table 5.11: Improvements in EER and AUC if best-match cosine similarity is used.

Plotting EER and AUC for the methods according to the number of enrollment samples makes the difference in performance even clearer:

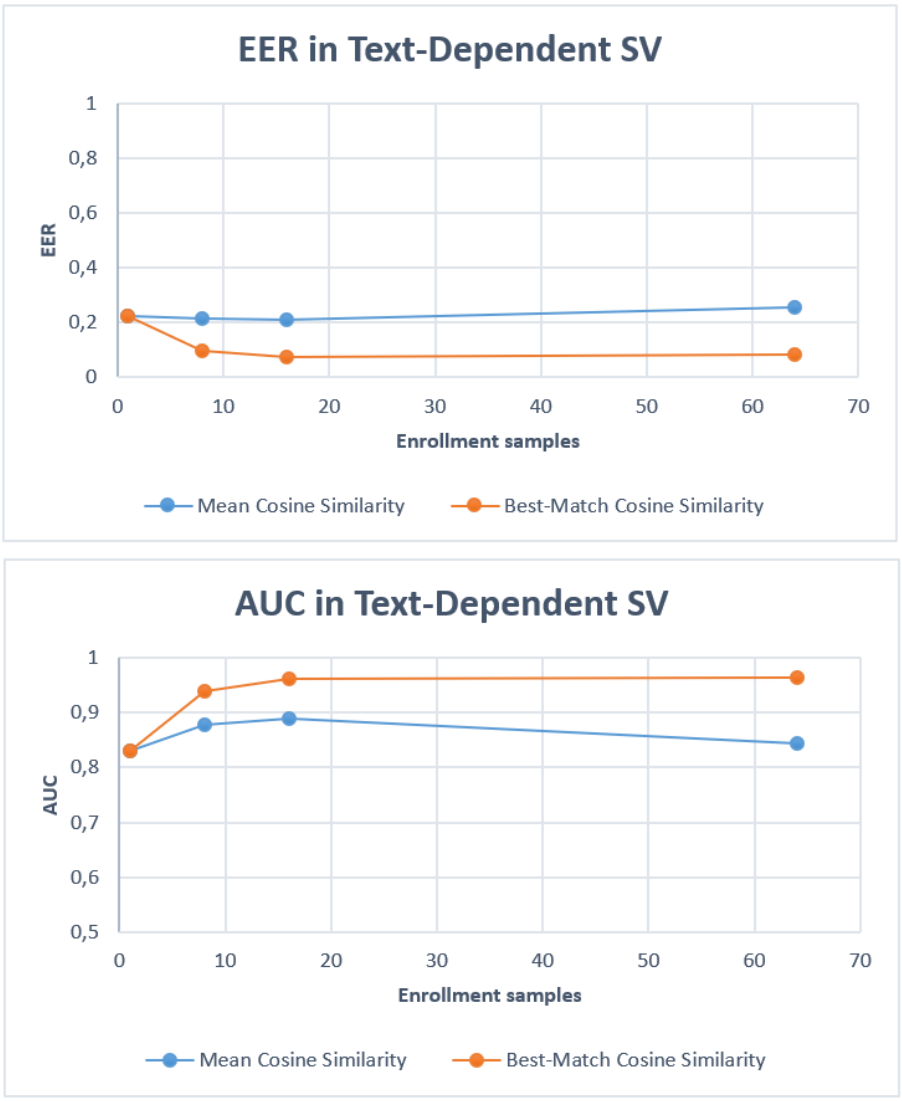


Figure 5.5: Confrontation of EER and AUC metrics for cosine similarity methods.

5.5. Final comments

Speaker verification has been proven to be a tough task to be solved, especially in a TinyML context. Simpler methods, such as ones based on cosine similarity, showed a greater performance, even better than more powerful classification methods such as neural networks and SVM. This can be explained by remarking that such methods require a lot of training data to be fitted and learn to generalize well on tasks. The few-shot condition poses a serious challenge for such models. In chapter 7.2, direction on how to improve obtained results will be given.

Strictly considering performance obtained on the tests, the new cosine similarity classi-

fication method proposed (Best-Match Cosine Similarity) seemed to outperform other classification methods in text-dependent TinyML relevant conditions (i.e. when the number of enrollment samples is 8 - 16). We can assume that the approach adopted in this thesis work showed that in a TinyML context, a text-independent training of a tiny feature extractor can produce efficient d-vectors to be used in a text-dependent context with a reasonably simple similarity confrontation system. It must be taken into account that the difference in performance may also be influenced by the preprocessing applied to data: being the pipeline described in section 3.4 optimized specifically for keyword spotting, it is reasonable to believe that it fails in capturing all the needed features to perform speaker verification in a text-independent way, while stronger behavior is shown if there is consistency in phonemes among utterances.

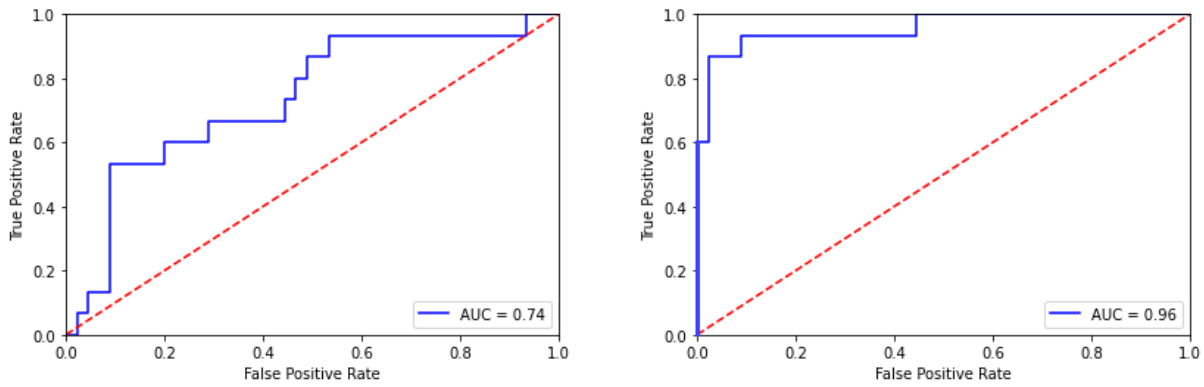


Figure 5.6: ROC curves for speaker 3 and 16 enrollment samples. Left: text-independent, right: text-dependent

In figure 5.6, ROC curves related to the same speaker (ID = 3), both computed with 16 enrollment samples and the best-match cosine similarity method are compared. The difference in performance is clear.

However, regardless of absolute performance results, all of the aforementioned methods have shown learning capabilities, thus making them suitable for being investigated further in TinyML oriented SV applications. It must not be excluded that future developments in the field will leverage better the capabilities of such algorithms, thus leading to most efficient TinyML oriented speaker verification systems.

6 | Cascading KWS and SV: on-device implementation

During the development of this master thesis, several different keyword spotting demos have been implemented on real hardware, ranging from simple wake-word detectors to more complex voice interaction applications. Also speaker verification methods have been implemented and tested, both in text-dependent and text-independent contexts with different parameters and algorithms. It would be impossible to organically present them all in a single chapter. To showcase both keyword spotting and speaker verification capability, we decided to build a custom demo combining both tasks implementing some of the TinyML solutions explored by this work. The goal of the application is to continuously listen for a specific keyword and provide a personalized answer if the utterance comes from the enrolled user, otherwise the system must provide a general answer. Moreover, the enrollment phase must be executed completely on-device only relying on voice interaction. This makes the system perform a text-dependent speaker verification.

After a section describing the process behind the choice of the models to be ported on real hardware, a brief description of the target architecture will be provided. Details about application development will follow, and a performance evaluation section will report information about usability of the final demo. This section will also analyze memory occupation and power consumption of the demo.

6.1. Choosing a method for on-device implementation

While keyword spotting models identified by this work seem to perform equally good, the same can not be said regarding speaker verification methods presented in chapter 4.

We chose among them letting our choice be driven by performance of the solution, easiness of implementation and processing power required to perform enrollment and inference. The most promising method, that outperforms the majority of all other approaches in

almost all aforementioned aspects, is the Best-Match Cosine Similarity described in section 4.5.2. Regarding keyword spotting, we chose to use in this demo the larger CNN model identified in 3, to leverage the highest possible accuracy in detecting keywords. For this specific demo, we chose the "Sheila" keyword to use as text-dependent utterance on which keyword spotting and speaker verification are performed.

6.2. Application structure

As explained in section 4.2.4, one of the goals was to obtain a speaker verification system that could work cascaded with the keyword spotting system, to provide a sort of personalized user interaction: if a keyword is deemed coming from the enrolled speaker, a personalized answer is given. Otherwise, a general answer is returned by the system. The high-level block scheme of the application is described in figure 6.1:

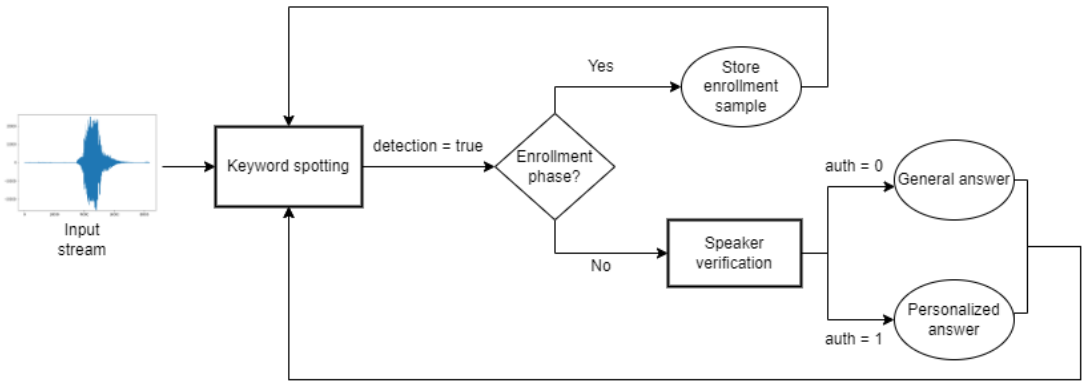


Figure 6.1: High-level scheme of the KWS-SV application.

The desired behavior of the application is the following: at startup, no user is enrolled. The system continuously listens to input audio stream from the microphone, running the KWS model. The enrolled speaker must pronounce the keyword multiple times to provide the enrollment samples, which are processed and stored according to the chosen algorithms. After enough samples from the chosen keywords have been collected, the system is ready to perform speaker verification: it still keeps listening for the chosen keyword, and if a detection happens the check on speaker identity follows. If the speaker is deemed to be the enrolled one, a personalized answer is given. Else, since the keyword has been detected anyway, a general answer is given.

The application works as a state machine, with a `ENROLLMENT` state that corresponds to the enrollment phase, and a `VERIFICATION` state which corresponds to the speaker verification phase. As this code snippet taken from the application main loop shows,

the keyword spotting system by using `RunWWModel()` and `WWProcessLatestResults()` functions controls the engagement of the speaker verification system in both states:

```
if(application_status == ENROLLMENT){
    MTB_ML_DATA_T* result = RunWWModel(input_features);
    int detected = WWProcessLatestResults(result);
    if(detected==kW1Index){
        MTB_ML_DATA_T* d_vector = RunSVModel((MTB_ML_DATA_T*)
                                             input_features);
        int v_acquired = SVStoreEmbedding(d_vector);
        application_status = SVCheckEnrollment(v_acquired);
        cyhal_system_delay_ms(500);
    }
}
if(application_status == VERIFICATION){
    MTB_ML_DATA_T* result = RunWWModel(input_features);
    int detected = WWProcessLatestResults(result);
    if(detected==kW1Index){
        float** speaker_model = getEmbeddingsAddress();
        MTB_ML_DATA_T* d_vector = RunSVModel(input_features);
        int authenticated = SVBestMatch(d_vector, speaker_model);
        application_status = SVCommandResponder(authenticated);
        printf("**********************************\n\r");
        printf("Speak to unlock.\n\r");
        cyhal_system_delay_ms(500);
    }
}
```

6.2.1. Input stream processing

One of the major differences between the tests conducted and a real world implementation is that in latter case audio comes as a continuous input stream from the microphone of the board. This audio stream must be converted in MFCC spectrograms before being fed to the neural networks for KWS/SV. The system implemented processes input audio stream continuously by extracting MFCC spectrograms from 1-second long sliding windows, as explained by figure 6.2, and by subsequently feeding them to the KWS neural network to perform detection.

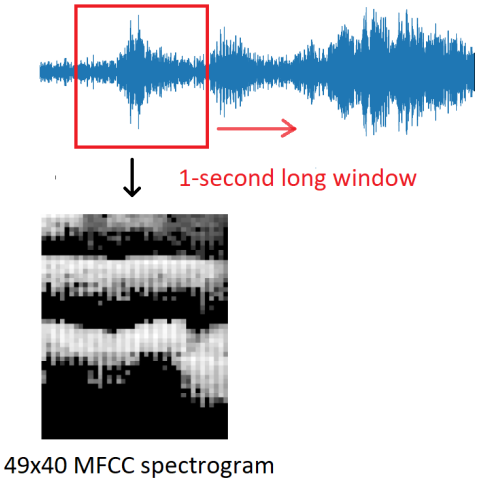


Figure 6.2: Sliding window on input audio stream that computes MFCC spectrograms.

However, computing a full 1-second long MFCC spectrogram each time is a resource-consuming task. Luckily, the sliding window mechanism can be leveraged by making the system recompute spectrogram portions only related to the new samples included after a window shift. The stride of the sliding window is related both to the preprocessing algorithm requirements and to the inference time of the models: to compute a new slice of the spectrogram at least 30 ms of audio data are needed. If the time that passes between the inference of the neural networks on the current spectrogram is longer than 30ms, the window is shifted by the amount of time passed. If the inference of the model is faster than 30ms, the system waits for having at least 30ms of audio data to perform the preprocessing computation on. If some audio portions overlap between windows (i.e. if the stride is less than 1 second), the related MFCC spectrogram slices are not recomputed.

The implementation of the MFCC preprocessing on the microcontroller is a C conversion of the pipeline adopted in [62], that closely replicates all the steps described in 3.4. Correctness of the C implementation against the Python implementation has been extensively tested to ensure coherence between training and inference phases of the models.

6.2.2. Keyword spotting system

Keyword spotting system implements the floating point version of the "Sheila" `conv-kws-nn` model, as described in section 3.7. The neural network takes as input the latest spectrogram computed as explained in previous section, and provides a confidence value on the presence of the desired keyword. Since it is possible to have the keyword slightly

outside the one-second window, or to have an early window on utterances containing similar phonemes to the desired keyword, the system actually returns a detection only if the mean confidence value of the last two detections is above a certain threshold. The choice of analyzing only two past detections is due to the long inference time needed by the network; if a faster model is deployed, such as the `smallconv-kws-nn` models, more past confidence values can be used to take a decision. The following code shows the implementation of such decision function:

```
int WWProcessLatestResults(MTB_ML_DATA_T* nn_output){
    int index = inference_index % 2;;
    inference_index++;
    for(int i = 0; i<3; i++){
        last_results[index][i] = nn_output[i];
    }
    int silence_confidence = (last_results[0][0] + last_results[1][0]) /
                                        2;
    int unknown_confidence = (last_results[0][1] + last_results[1][1]) /
                                        2;
    int ww_confidence = (last_results[0][2] + last_results[1][2]) / 2;
    printf("Silence: %d, Unknown: %d, Sheila: %d\n\r",
                                       silence_confidence,
                                       unknown_confidence,
                                       ww_confidence);
    if(silence_confidence > WW_SENSITIVITY){
        return kSilenceIndex;
    }
    if(unknown_confidence > WW_SENSITIVITY){
        return kUnknownIndex;
    }
    if(ww_confidence > WW_SENSITIVITY){
        return kW1Index;
    }
    return kUnknownIndex;
}
```

6.2.3. Speaker verification system

Once the keyword spotting system has detected the keyword, the speaker verification system starts to execute. The neural network that is used as d-vector extractor is the one described in 4.4. No quantization has been applied to this network not to reduce its performance. The latest input spectrogram that contributed to generating a detection on the KWS system is passed as input to the d-vector extractor of the SV system.

Now, the speaker verification system can be called in two different states, as figure 6.1 explains. Regardless from the state, however, a d-vector gets extracted from the MFCC spectrogram by calling `d_vector = RunSVModel(input_features)`, as can be noticed by the code snippet in section 6.2. If the system is in enrollment state, the d-vector extracted contributes to forming the speaker model. Since the similarity method adopted is the Best-Match Cosine Similarity, the speaker model is simply a matrix containing all the enrollment d-vectors from the authenticated speaker. Once enough enrollment d-vectors have been collected, the speaker verification system starts operating in `VERIFICATION` mode. The d-vector extracted is used for a comparison with the stored speaker model via the Best-Match Cosine Similarity, as shown by the call to `authenticated =SVBestMatch (d_vector, speaker_model)` in code snippet of section 6.2. The following code snippet reports the implementation of the Best-Match Cosine Similarity function:

```
int SVBestMatch(MTB_ML_DATA_T* nn_output, float** stored_embeddings){
    float scores[N_ENROLLMENT_SAMPLES];
    for(int i=0; i<EMBEDDING_SIZE; i++){
        d_vector[i] = nn_output[i];
    }
    for(int i = 0; i<N_ENROLLMENT_SAMPLES; i++){

        float similarity = CosineSimilarity(d_vector, stored_embeddings+
                                            (i*EMBEDDING_SIZE));
        scores[i] = similarity;
    }
    float best_score = -1.0;
    for(int i = 0; i<N_ENROLLMENT_SAMPLES; i++){
        if(scores[i] > best_score){
            best_score = scores[i];
        }
    }
    printf("Similarity: %f\n\r", best_score);
    if(best_score > SV_SENSITIVITY){
        return 1;
    }
        return 0;
}
```

If the speaker verification system returns `authenticated =1`, the input utterance is attributed to the enrolled speaker, and a personalized message is communicated to the user. Otherwise, a general message is communicated. The decision on user identity is taken by comparing the similarity value obtained against a user-defined threshold. Such threshold can be manually tuned at any time during execution via a potentiometer, to allow users

balancing false positives and false negatives.

6.3. Target architecture description

The target architecture for the implementation of our solution is the Infineon PSoC 62S2 Wi-Fi BT Pioneer Board. PSoC 6 MCU is a programmable embedded system-on-chip, integrating a 150-MHz Arm® Cortex®-M4 as the primary application processor, a 100-MHz Arm Cortex-M0+ that supports low-power operations, up to 2 MB Flash and 1 MB SRAM, and the compatibility with Arduino™ shields. For the purpose of this master thesis, only the Arm® Cortex®-M4 processor has been addressed as the main target for developing the application, disregarding dual-core implementations.

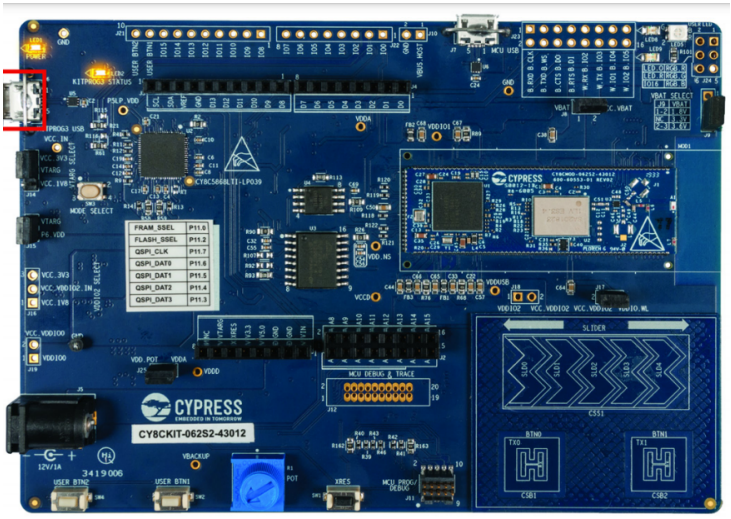


Figure 6.3: Infineon PSoC6 62s2 Wi-Fi BT Pioneer Board.

The microphone needed for collecting audio samples has been connected to the board by relying on the Infineon CY8CKIT-028-SENSE shield, is a low-cost Arduino™ UNO compatible shield board that can be used to easily interface a variety of sensors with the PSoC™ 6 MCU platform, specifically targeted for audio and machine learning applications. Of our interest is the presence of two Infineon XENSIV™ digital MEMS microphones, that can be used to capture sound and generate digital audio data, which is then transferred through the PDM interface.

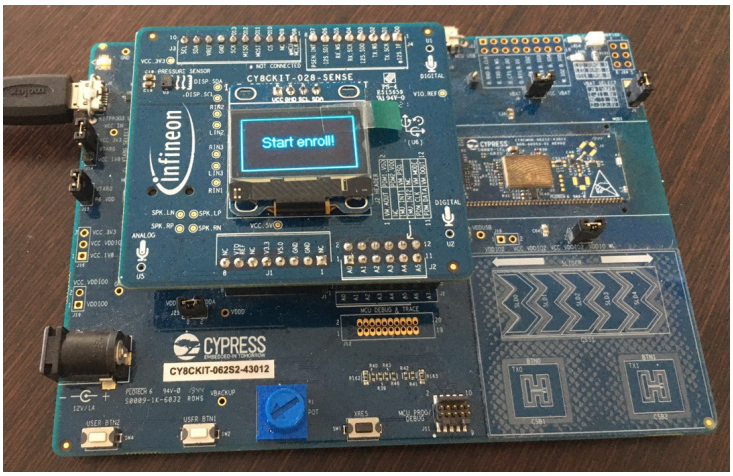


Figure 6.4: CY8CKIT-028-SENSE IoT sense expansion kit connected to a Infineon PSoC 62S2 Wi-Fi BT Pioneer Board.

6.4. Performance evaluation

The demo in its test on the field has shown capabilities of recognizing the enrolled speaker, although its performance is far from perfect. With respect to testing results obtained in chapter 5, performance on real hardware seems to be slightly worse. However, this can be attributed to the lack of optimization of the demo. For example, there are no guarantees that the MFCC spectrogram used for extracting the d-vector contains the keyword utterance in its full length. Moreover, testing results in chapter 5 were obtained with the "Hey Cypress" utterance, which is longer and therefore easier for the SV system to process. If the environment is quite enough and speakers pronounce the keyword clearly, however, the application shows promising signs of adaptation to the enrolled speaker voice. There is still a lot of space for improvements: voice activity detection, noise reduction and audio alignment could be implemented to obtain better performance in real-world testing conditions.

A video showcasing the demo working, along with code and other useful resources, can be found at `https://github.com/lolepls/cascaded-tiny-KWS-SV`.

6.5. Memory and power consumption

An evaluation of consumption of the system has been done regarding both memory consumption and power consumption. However, it is worth to mention that this demo has not been subject to processes for optimizing memory footprint and power consumption. Application engineering could allow obtaining much better results in both fields, for ex-

ample by leveraging the smaller networks identified in chapter 3. Such measurements have been done on the aforementioned developer kit running the Arm® Cortex®-M4 at full power.

6.5.1. Memory consumption

The whole application requires about 354.32 kB of flash memory to be stored. This includes the two neural network models, all the preprocessing code and all the code needed to setup the board, manage the audio acquisition and the I/O interface via LCD screen and LEDs.

Final demo requirements	
Flash memory	354.32 kB
RAM memory	504.12 kB

Table 6.1: Final KWS-SV demo memory requirements.

At runtime, the neural networks require additional 70.5 kB for the d-vector extractor activations, and 79.3 kB for the KWS activations, leading to a total of RAM memory required equal to 504.12 kB. Table 6.2 will report contribution of each allocated element to the total RAM consumption:

Memory consumption	
Input audio stream buffer	32 kB
MFCC input spectrogram (float)	7.5 kB
KWS model weights	112.5 kB
KWS model activations	79.3 kB
d-vector extractor weights	98.08 kB
d-vector extractor activations	70.5 kB
Test d-vector	1 kB
Speaker model	16 kB
Other	87.24 kB
Total	504.12 kB

Table 6.2: Runtime memory consumption for KWS-SV demo.

The input MFCC spectrogram consists of 1960 floating point values. Each d-vector extracted by the neural network for speaker verification occupies 1 kB of memory, being

stored as 256 `float32` values. Since the demo uses 16 enrollment d-vectors, 16 kB of memory are allocated for storing the speaker model. The d-vector that is extracted during the verification phase requires one additional kB of memory allocated. "Other" memory is allocated for the .text section of the executable, along with other .data and .bss sections containing variables not strictly related to machine learning models.

If the neural network used for performing KWS and SV are quantized to the lowest bit depth possible, the size of the application can be reduced to:

Final demo requirements	
Flash memory	196.38 kB
RAM memory	247.68 kB

Table 6.3: Final 8-bit quantized KWS-SV demo memory requirements.

This measurement has been obtained with the Infineon ModusToolbox software when 8-bit quantization of the chosen models has been adopted.

7 | Conclusions

This chapter will provide a summary of the results obtained in the scope of this thesis, along with possible future investigations on the problems addressed.

7.1. Conclusion

This Master Thesis described an organic approach for combining two fundamental speech processing tasks in TinyML, namely keyword spotting and speaker verification. Keyword spotting is the application that raised interest in Tiny Machine Learning, making scientists and engineers understand the benefits of deploying artificial intelligence algorithms on low-power edge devices. Due to its relevance, research extensively focused on such application; results obtained by the scientific community have been analyzed in this Master Thesis and used as background to tackling the second task: speaker verification. This thesis is the first work in the TinyML literature to carry out a complete top-down analysis of the design process for a cascaded TinyML oriented KWS-SV system. Moreover, we framed the TinyML speaker verification task in the novel one-class few-shot classification context, highlighting constraints and limitations that have to be faced when developing such a system and analyzing ways for overcoming them. This thesis work also showed how developing complex intelligent systems targeted at ultra low-power computers is a challenge that requires engineers to take into account TinyML limitations even from early stages of the development: not only AI design, but all the steps from data preprocessing to final testing need a TinyML driven design. This work described in detail such design process, laying basis for future works on the field.

Strictly regarding testing results obtained, hardware boards equipped with the keyword spotting system designed in this thesis work showed great performance, in some cases even outperforming other state-of-the-art systems; this made the keyword spotting demos commercially viable for the company that supported this work. The speaker verification system also obtained promising results, showing signs of recognition capabilities among different speakers. The relevant result obtained is the proof that solving such task in ultra low-power MCUs is possible. The speaker verification demo presented in this thesis work

has been adopted as a proof-of-concept by the company supporting this work. Regarding application engineering, the implementation of keyword spotting and speaker verification adopted in this master thesis allows designing pure keyword spotting systems, pure speaker verification systems or a combined KWS/SV system that merges capabilities of both.

Moreover, a novel similarity system for performing speaker verification has been proposed, in some cases outperforming other state-of-the-art approaches.

7.2. Future works

Still a lot has to be investigated regarding speech processing tasks in a TinyML context, but the advantages of developing such technologies would be countless, making worth the research. While keyword spotting systems in TinyML contexts are being addressed more and more both in industry and academy, with very promising results, speaker verification is still a novel research field.

To progress in the research, following features could be addressed. However, this is not meant to be an exhaustive list; it contains general ideas developed during the thesis work based on the acquired experience.

Power of the d-vector extractor: the capability of the d-vector extractor is fundamental for providing a good separation in the feature space of the utterances submitted to the system, which in turn eases the work of similarity algorithms. Being constrained by the TinyML condition, that requires designing small models, obtaining such a model is a difficult task that influences the final performance. The adoption of end-to-end loss functions, such as the Generalized End To End loss [58], could improve the performance of the architecture without having to add more layers or parameters.

Data preprocessing: the data preprocessing used in this work has been specifically designed for keyword spotting. While MFCC spectrograms are the state-of-the-art for speech processing systems, fine-tuning on the process can be implemented to highlight desired characteristics. Keyword spotting and speaker verification require almost opposite features: one case needs speaker information almost completely removed to focus on similarity of phonemes pronounced, while the other need such features highlighted to give relevance to per-speaker peculiarities.

Short duration of utterances: Speaker Verification systems can perform better if longer utterances are submitted to the analysis, because it is easier to spot speaker-dependent characteristics when more time-related data is available. However, given

the needed synergy with the Keyword Spotting system and the constraints of embedded devices, longer utterances were not suitable to be used in this case.

Investigating hyperparameters: OC-SVM and OC-NN could benefit from deeper investigations of their hyperparameters to find the best combination. Regarding OC-NN, better fictitious vectors generators can be adopted, while OC-SVM could benefit from a more organic hyperparameter search.

Processing on similarity values: Cosine similarity methods have been proven to be powerful, but there is still space for improvement. The production of the sim value in the Best-Match Cosine Similarity method, for example, is performed using a simple maximum operation on the values of $\boldsymbol{sm}$, but other more articulated solutions could be tested. One idea could be not to focus only on the maximum value, but also the minimum could be considered when providing a confidence threshold on the identity of the speaker: we want our embedding to be similar enough also to the "less similar" stored embedding.

Application engineering: Optimizations can also be done at application level when the SV system is implemented in real-hardware. A preliminary alignment at runtime of audio utterances in case of text-dependent speaker verification could lead to improvement, and the same goes for noise reduction systems or voice activity detection algorithms to use in conjunction with the speaker verification pipeline.

Bibliography

[1] m2cgen - python library, 2020. URL `https://github.com/BayesWitnesses/m2cgen`.

[2] *Transactions on embedded computing systems*. 7 2023.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg ; Martin Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: learning functions at scale. *ACM SIGPLAN Notices*, 51(9):1–1, 2015. doi: 10.1145/3022670.2976746.

[4] Apple. Hey siri: An on-device dnn-powered voice trigger for apple's personal assistant. Online, Oct. 2017. URL `https://machinelearning.apple.com/research/hey-siri`.

[5] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J.-s. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav. Benchmarking tinyml systems: Challenges and direction. Mar. 2020. doi: 10.48550/ARXIV.2003.04821.

[6] A. Banerjee, A. Dubey, A. Menon, S. Nanda, and G. C. Nandi. Speaker recognition using deep belief networks. May 2018. doi: 10.48550/ARXIV.1805.08865.

[7] K. B. Bhangale and K. Mohanaprasad. A review on speech processing using machine learning paradigm. *International Journal of Speech Technology*, 24(2):367–388, jan 2021. doi: 10.1007/s10772-021-09808-0.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2008. ISBN 9780387310732.

[9] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. F. Wang, and J.-B. Huang. A closer look at few-shot classification. Apr. 2019. doi: 10.48550/ARXIV.1904.04232.

[10] Y.-h. Chen, I. L. Moreno, T. Sainath, M. Visontai, R. Alvarez, and C. Parada.

Locally-connected and convolutional neural networks for small footprint speaker recognition. 2015.

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

[12] R. David, J. Duke, A. Jain, V. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems. 2020.

[13] N. Dehak, P. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet. Front-end factor analysis for speaker verification. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(4):788–798, 5 2011. doi: 10.1109/tasl.2010.2064307.

[14] S. R. Department. Internet of things - number of connected devices worldwide 2015-2025, Nov. 2016. URL `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`.

[15] M. R. Eugeniu Ostrovan, Massimo Pavan. Tinyml on-device neural network training. 2022.

[16] fbeilstein. Simplest smo ever. Github Repository, Feb. 2021. URL `https://github.com/fbeilstein/simplest_smo_ever/blob/master/simple_svm.ipynb`.

[17] F. K. Felix Johnny. Tinyml talks - enabling ultra low-power machine learning at the edge, 2018.

[18] E. Hardy and F. Badets. An ultra-low power rnn classifier for always-on voice wake-up detection robust to real-world scenarios. 2021.

[19] *ModusToolbox™ Machine Learning user guide*. Infineon Technologies AG, 2022.

[20] B.-H. Juang and L. Rabiner. Speech recognition, automatic: History. In *Encyclopedia of Language &amp Linguistics*, pages 806–819. Elsevier, 2006. doi: 10.1016/b0-08-044854-2/00906-8.

[21] A. S. Lampropoulos and G. A. Tsihrintzis. The learning problem. In *Machine Learning Paradigms*, pages 31–61. Springer International Publishing, 2015. doi: 10.1007/978-3-319-19135-5_3.

[22] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015. doi: 10.1038/nature14539.

[23] S. Lins, K. D. Pandl, H. Teigeler, S. Thiebes, C. Bayer, and A. Sunyaev. Artificial intelligence as a service. *Business Information Systems Engineering*, 63(4):441–456,

2021. ISSN 1867-0202. doi: 10.1007/s12599-021-00708-w. URL https://doi.org/10.1007/s12599-021-00708-w.

[24] I. López-Espejo, Z.-H. Tan, J. Hansen, and J. Jensen. Deep spoken keyword spotting: An overview, 2021.

[25] M. Mazumder, S. Chitlangia, C. Banbury, Y. Kang, J. Ciro, F. Mlcommons, K. Achorn, D. Galvez Nvidia, M. Sabini, P. Mattson, D. Mlcommons, G. Diamos, P. Warden, J. Coqui, and V. Reddi. Multilingual spoken words corpus. 2021.

[26] D. Miller, M. Kleber, C.-L. Kao, O. Kimball, T. Colthurst, S. Lowe, R. Schwartz, and H. Gish. Rapid and accurate spoken term detection. 2007. doi: ng.

[27] A. Misra and J. H. Hansen. Modelling and compensation for language mismatch in speaker verification. *Speech Communication*, 96:58–66, 2018. ISSN 0167-6393. doi: https://doi.org/10.1016/j.specom.2017.09.004. URL https://www.sciencedirect.com/science/article/pii/S0167639317300444.

[28] S. Mittermaier, L. Kürzinger, B. Waschneck, and G. Rigoll. Small-footprint keyword spotting on raw audio data with sinc-convolutions. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7454–7458, 2020. doi: 10.1109/ICASSP40776.2020.9053395.

[29] A. Mobiny and M. Najarian. Text-independent speaker verification using long short-term memory networks. May 2018. doi: 10.48550/ARXIV.1805.00604.

[30] D. Morawiec. sklearn-porter, 2020. URL https://github.com/nok/sklearn-porter.

[31] M. M. Moya and D. R. Hush. Network constraints and multi-objective optimization for one-class classification. *Neural Networks*, 9(3):463–474, apr 1996. doi: 10.1016/0893-6080(95)00120-4.

[32] S. Narang and M. Divya Gupta. Speech feature extraction techniques: A review. 2015.

[33] N. S. Nehe and R. S. Holambe. DWT and LPC based feature extraction methods for isolated word recognition. *EURASIP Journal on Audio, Speech, and Music Processing*, 2012(1), jan 2012. doi: 10.1186/1687-4722-2012-7.

[34] D. O'Shaughnessy. *Speech communication.* Addison-Wesley Pub. Co., 1987. ISBN 0201165201.

[35] P. Oza and V. M. Patel. One-class convolutional neural network. *IEEE Signal Processing Letters*, 26(2):277–281, feb 2019. doi: 10.1109/lsp.2018.2889273.

[36] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, apr 2015. doi: 10.1109/icassp.2015.7178964.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[38] P. Perera, P. Oza, and V. M. Patel. One-class classification: A survey. *ArXiv*, abs/2101.03064, 2021.

[39] W. Rawat and Z. Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, sep 2017. doi: 10.1162/neco_a_00990.

[40] G. Rebala, A. Ravi, and S. Churiwala. Machine learning definition and basics. In *An Introduction to Machine Learning*, pages 1–17. Springer International Publishing, 2019. doi: 10.1007/978-3-030-15729-6_1.

[41] R. F. O. R.J. Baken. *Clinical Measurement of Speech and Voice, 2nd Edition (pp. 177)*. London: Taylor and Francis Ltd. ISBN ISBN 1-5659-3869-0.

[42] J. Rohlicek, W. Russell, S. Roukos, and H. Gish. Continuous hidden markov modeling for speaker-independent word spotting. In *International Conference on Acoustics, Speech, and Signal Processing,*, pages 627–630 vol.1, 1989. doi: 10.1109/ICASSP.1989.266505.

[43] M. Sahidullah and G. Saha. Design, analysis and experimental evaluation of block based transformation in MFCC computation for speaker recognition. *Speech Communication*, 54(4):543–565, may 2012. doi: 10.1016/j.specom.2011.11.004.

[44] T. N. Sainath and C. Parada. Convolutional neural networks for small-footprint keyword spotting. In *Interspeech 2015*. ISCA, sep 2015. doi: 10.21437/interspeech.2015-352.

[45] R. Sanchez-Iborra and A. F. Skarmeta. TinyML-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020. doi: 10.1109/mcas.2020.3005467.

[46] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt. Support vector method for novelty detection. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 582–588. The MIT Press, 1999. URL `http://papers.nips.cc/paper/1723-support-vector-method-for-novelty-detection`.

[47] H. Shimodaira and S. Renals. Speech signal analysis. 1 2019.

[48] D. Snyder, D. Garcia-Romero, D. Povey, and S. Khudanpur. Deep neural network embeddings for text-independent speaker verification. In *Interspeech 2017*. ISCA, aug 2017. doi: 10.21437/interspeech.2017-620.

[49] D. Snyder, D. Garcia-Romero, G. Sell, D. Povey, and S. Khudanpur. X-vectors: Robust DNN embeddings for speaker recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, apr 2018. doi: 10.1109/icassp.2018.8461375.

[50] D. Sztahó, G. Szaszák, and A. Beke. Deep learning methods in speaker recognition: a review. *Periodica Polytechnica Electrical Engineering and Computer Science*, 65 (4):310–328, oct 2019. doi: 10.3311/ppee.17024.

[51] D. Thailappan. An introduction to few shot learning. Blog article, May 2021. URL `https://www.analyticsvidhya.com/blog/2021/05/an-introduction-to-few-shot-learning/`.

[52] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The computational limits of deep learning. July 2020. doi: 10.48550/ARXIV.2007.05558.

[53] S. Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[54] R. Tronci, G. Giacinto, and F. Roli. Dynamic score combination: A supervised and unsupervised score combination method. In *Machine Learning and Data Mining in Pattern Recognition*, pages 163–177. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-03070-3_13.

[55] Y. Tu, W. Lin, and M.-W. Mak. A survey on text-dependent and text-independent speaker verification. *IEEE Access*, 10:99038–99049, 2022. doi: 10.1109/ACCESS.2022.3206541.

[56] E. Variani, X. Lei, E. McDermott, I. L. Moreno, and J. Gonzalez-Dominguez. Deep neural networks for small footprint text-dependent speaker verification. In *2014 IEEE*

International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, may 2014. doi: 10.1109/icassp.2014.6854363.

[57] J. Villalba, N. Brümmer, and N. Dehak. Tied variational autoencoder backends for i-vector speaker recognition. In *Interspeech 2017*. ISCA, aug 2017. doi: 10.21437/interspeech.2017-1018.

[58] L. Wan, Q. Wang, A. Papir, and I. L. Moreno. Generalized end-to-end loss for speaker verification. Oct. 2017. doi: 10.48550/ARXIV.1710.10467.

[59] Y. Wang, P. Getreuer, T. Hughes, R. F. Lyon, and R. A. Saurous. Trainable frontend for robust and far-field keyword spotting. July 2016. doi: 10.48550/ARXIV.1607.05666.

[60] P. Warden. Speech commands: A dataset for limited-vocabulary speech recognition. 4 2018.

[61] P. Warden. Why isn't there more training on the edge? Online, Sept. 2020. URL `https://petewarden.com/2022/09/06/why-isnt-there-more-training-on-the-edge/`.

[62] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020. ISBN 9781492052043. URL `https://books.google.it/books?id=sB3mxQEACAAJ`.

[63] M. Weintraub. Keyword-spotting using sri's decipher large-vocabulary speech-recognition system. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 463–466 vol.2, 1993. doi: 10.1109/ICASSP.1993.319341.

[64] J. Wilpon, L. Miller, and P. Modi. Improvements and applications for key word recognition using hidden markov modeling techniques. In *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, pages 309–312 vol.1, 1991. doi: 10.1109/ICASSP.1991.150338.

[65] X. Yuan, G. Li, J. Han, D. Wang, and Z. Tiankai. Overview of the development of speaker recognition. *Journal of Physics: Conference Series*, 1827(1):012125, mar 2021. doi: 10.1088/1742-6596/1827/1/012125.

A | Appendix A - Tables

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

Table A.1: Speech commands dataset details.

Testing results on Text-Independent Dataset - CNN				
Speaker	**Enrollment Samples**	**Classifier Method**	**Accuracy**	**F1-SCORE**
0	1	Mean Cosine Similarity	0,51	0,35
0	1	Best-Match Cosine Similarity	0,51	0,35
0	1	OC-NN	0,8	0,5
0	1	OC-SVM	0	0
0	8	Mean Cosine Similarity	0,7	0,59
0	8	Best-Match Cosine Similarity	0,73	0,528
0	8	OC-NN	0,75	0,51
0	8	OC-SVM	0,483	0,392
0	16	Mean Cosine Similarity	0,7	0,526
0	16	Best-Match Cosine Similarity	0,66	0,523
0	16	OC-NN	0,66	0,476
0	16	OC-SVM	0,483	0,392
0	64	Mean Cosine Similarity	0,73	0,55
0	64	Best-Match Cosine Similarity	0,65	0,361
0	64	OC-NN	0,3	0,41
0	64	OC-SVM	0,6	0,478
1	1	Mean Cosine Similarity	0,516	0,325
1	1	Best-Match Cosine Similarity	0,516	0,325
1	1	OC-NN	0,66	0
1	1	OC-SVM	0	0
1	8	Mean Cosine Similarity	0,65	0,48
1	8	Best-Match Cosine Similarity	0,55	0,34
1	8	OC-NN	0,45	0
1	8	OC-SVM	0,483	0,279
1	16	Mean Cosine Similarity	0,616	0,465
1	16	Best-Match Cosine Similarity	0,55	0,366
1	16	OC-NN	0,33	0,13
1	16	OC-SVM	0,616	0,549
1	64	Mean Cosine Similarity	0,616	0,465
1	64	Best-Match Cosine Similarity	0,5	0,372
1	64	OC-NN	0,26	0,405
1	64	OC-SVM	0,533	0,3
2	1	Mean Cosine Similarity	0,55	0,37
2	1	Best-Match Cosine Similarity	0,55	0,37
2	1	OC-NN	0,55	0,371
2	1	OC-SVM	0	0
2	8	Mean Cosine Similarity	0,53	0,39
2	8	Best-Match Cosine Similarity	0,65	0,42
2	8	OC-NN	0,68	0,457
2	8	OC-SVM	0,533	0,333
2	16	Mean Cosine Similarity	0,616	0,439
2	16	Best-Match Cosine Similarity	0,56	0,43
2	16	OC-NN	0,6	0,5
2	16	OC-SVM	0.633	0,352
2	64	Mean Cosine Similarity	0,6	0,4
2	64	Best-Match Cosine Similarity	0,45	0,352
2	64	OC-NN	0,266	0,388
2	64	OC-SVM	0,633	0,352
3	1	Mean Cosine Similarity	0,55	0,37
3	1	Best-Match Cosine Similarity	0,55	0,37
3	1	OC-NN	0,66	0
3	1	OC-SVM	0	0
3	8	Mean Cosine Similarity	0,65	0,43
3	8	Best-Match Cosine Similarity	0,58	0,418
3	8	OC-NN	0,53	0,263
3	8	OC-SVM	0,416	0,444
3	16	Mean Cosine Similarity	0,6	0,428
3	16	Best-Match Cosine Similarity	0,66	0,498
3	16	OC-NN	0,416	0,285
3	16	OC-SVM	0,383	0,244
3	64	Mean Cosine Similarity	0,65	0,4
3	64	Best-Match Cosine Similarity	0,6	0,47
3	64	OC-NN	0,245	0,388
3	64	OC-SVM	0,433	0,32

Table A.2: Full text-independent dataset testing results for speaker verification task.

Testing results on Text-Dependent Dataset - CNN				
Speaker	**Enrollment Samples**	**Classifier Method**	**Accuracy**	**F1-SCORE**
0	1	Mean Cosine Similarity	0,716	0,564
0	1	Best-Match Cosine Similarity	0,716	0,564
0	1	OC-NN	0,716	0,105
0	1	OC-SVM	0,75	0
0	8	Mean Cosine Similarity	0,783	0,682
0	8	Best-Match Cosine Similarity	0,816	0,717
0	8	OC-NN	0,75	0,347
0	8	OC-SVM	0,766	0,461
0	16	Mean Cosine Similarity	0,8	0,7
0	16	Best-Match Cosine Similarity	0,95	0,909
0	16	OC-NN	0,683	0,486
0	16	OC-SVM	0,4	0,181
0	64	Mean Cosine Similarity	0,9	0,812
0	64	Best-Match Cosine Similarity	0,95	0,903
0	64	OC-NN	0,3	0,399
0	64	OC-SVM	0,933	0,866
1	1	Mean Cosine Similarity	0,96	0,93
1	1	Best-Match Cosine Similarity	0,96	0,93
1	1	OC-NN	0,733	0
1	1	OC-SVM	0,75	0
1	8	Mean Cosine Similarity	0,93	0,87
1	8	Best-Match Cosine Similarity	0,933	0,857
1	8	OC-NN	0,65	0
1	8	OC-SVM	0,166	0,264
1	16	Mean Cosine Similarity	0,916	0,848
1	16	Best-Match Cosine Similarity	0,95	0,896
1	16	OC-NN	0,566	0,35
1	16	OC-SVM	0,516	0,355
1	64	Mean Cosine Similarity	0,85	0,769
1	64	Best-Match Cosine Similarity	0,966	0,933
1	64	OC-NN	0,516	0,508
1	64	OC-SVM	0,75	0
2	1	Mean Cosine Similarity	0,8	0,625
2	1	Best-Match Cosine Similarity	0,8	0,625
2	1	OC-NN	0,766	0,125
2	1	OC-SVM	0,75	0
2	8	Mean Cosine Similarity	0,8	0,666
2	8	Best-Match Cosine Similarity	0,833	0,705
2	8	OC-NN	0,816	0,56
2	8	OC-SVM	0,716	0,413
2	16	Mean Cosine Similarity	0,75	0,615
2	16	Best-Match Cosine Similarity	0,85	0,742
2	16	OC-NN	0,733	0,555
2	16	OC-SVM	0,883	0,72
2	64	Mean Cosine Similarity	0,733	0,6
2	64	Best-Match Cosine Similarity	0,85	0,709
2	64	OC-NN	0,383	0,412
2	64	OC-SVM	0,833	0,615
3	1	Mean Cosine Similarity	0,616	0,439
3	1	Best-Match Cosine Similarity	0,616	0,439
3	1	OC-NN	0,733	0
3	1	OC-SVM	0,75	0
3	8	Mean Cosine Similarity	0,75	0,594
3	8	Best-Match Cosine Similarity	0,85	0,689
3	8	OC-NN	0,716	0,451
3	8	OC-SVM	0,783	0,58
3	16	Mean Cosine Similarity	0,833	0,736
3	16	Best-Match Cosine Similarity	0,983	0,965
3	16	OC-NN	0,75	0,482
3	16	OC-SVM	0,733	0,529
3	64	Mean Cosine Similarity	0,6	0,47
3	64	Best-Match Cosine Similarity	0,983	0,965
3	64	OC-NN	0,35	0,434
3	64	OC-SVM	0,783	0,551

Table A.3: Full text-dependent dataset testing results for speaker verification task.

B | Appendix B - Additional notes

B.1. Notes on loss functions for d-vector extractors

The d-vector extractor neural network has been trained on a dataset with N samples belonging to K different classes by using a softmax classifier as the last layer, where the activation function is the softmax function [8]. The softmax activation function takes in a vector $\boldsymbol{z}$ of raw outputs of the neural network and returns a vector of probability scores, where each element is computed as:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K \tag{B.1}$$

Where K is the size of the input vector. Then, the loss function chosen for training is the categorical cross-entropy, defined as [8]:

$$L(\boldsymbol{Y}, \boldsymbol{T}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \tag{2}$$

Where:

- $\boldsymbol{T}$ is the ground truth matrix of size $N \times K$;
- $\boldsymbol{Y}$ is the output matrix of the model of size $N \times K$;
- $t_{nk} = \{0, 1\}$ is the target of input n for class K;
- y_{nk} is the probability that sample n belongs to class k as produced by the softmax layer on the model.

However, it could be argued that training a neural network in an end-to-end manner, already to produce distinct d-vectors for different speakers, would be a better approach to the problem. This is why literature produced loss functions suitable for an end-to-end training of a d-vector extractor model, such as the Generalized End-To-End Loss

[58]. The purpose of an end-to-end loss function for speaker verification is to enforce producing similar d-vectors for utterances of the same speaker, maximizing the difference between d-vectors of different speakers. While this approach could in principle bring some advantages, it has been observed that in TinyML contexts the approach based on training with the softmax loss and extracting the embedding from the hidden layer works with comparable results [61]. This is why we adopted the loss described in this sappendix, in addition to the advantages of being it an established and widely adopted approach.

B.2. Computing fictitious d-vectors in OC-NNs

The operation for computing fictitious d-vectors to be fed into the OC-NN described in this thesis work has been done to maintain the sparsity of generated real d-vectors to have plausible negative samples. The following code snippet will clarify the generation process:

```
# Generate denied gaussian sample:
gaussian_vector = generate_gaussian(mean, variance)
gaussian_vector_masked = mask_function(auth_embedding.reshape(256),
                                       gaussian_vector)
```

Where `auth_embedding` is a real d-vector produced from an authenticated speech sample, and the functions `generate_gaussian` and `mask_function` are defined as:

```
def generate_gaussian(mean, variance):
    return np.random.normal(mean, variance, embedding_size)
```

```
def mask_function(mask, data):
  for i in range(0, len(data)):
    data[i] = data[i] * mask[i]
  random.shuffle(data)
  return data
```

As it can be noticed from the following image, fictitious and real d-vectors present similarities in magnitude and sparsity of the elements:

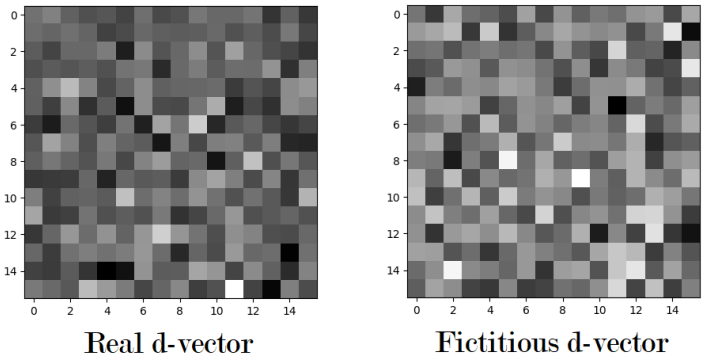


Figure B.1: Visual representation of real vs fictitious d-vectors.

It could be argued that the process of masking and shuffling the d-vectors is not needed, since drawing from a Gaussian distribution seems enough to produce a plausible sample. While this may be true if the d-vector produced does not show signs of sparsity, in case of having sparse d-vectors (for example, if the d-vector is obtained as the activation of a layer that uses the Rectified Linear Unit activation function) this may not be true. As an example, here there is a confrontation of a real d-vector, a fictitious d-vector without our proposed approach and a fictitious d-vector obtained with our approach, where the real d-vector is extracted as the activation of the dense layer of 4.4.1:

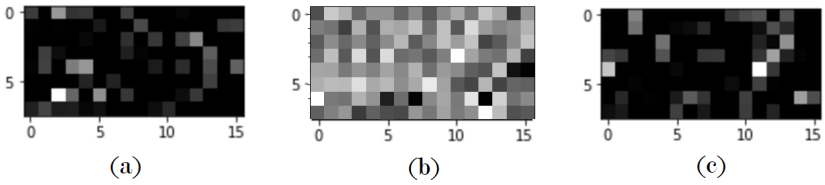


Figure B.2: (a): Real d-vector (b): Purely random fictitious d-vector (c): Masked-shuffled fictitious d-vector

Since in development of SV systems different d-vectors extracted from different part of the network could be tested, using the proposed approach can help producing plausible negative samples for the one-class training process, regardless of the origin of the d-vector.

B.3. SV approach-specific considerations

It may seem strange to think about validation set for some of the methods for speaker verification proposed in this work. For example, cosine similarity based methods do not have a proper training or fitting phase, so in principle they would not require validation sets. However, a useful way of leveraging validation set is to find a threshold for the

similarity produced by the classifiers. In general, validation sets have been used in the following way:

1. To **find the best threshold** for Cosine-Similarity based methods;
2. To **monitor the training** of the One-Class Classification Neural Networks;
3. To **perform hyperparameter search** on the One-Class SVMs

Thresholds and parameters found with this method have been maintained during the evaluation on the testing set.

Case 1) refers to the fact that a similarity threshold must be determined to establish if an utterance belongs to the enrolled speaker or not. The validation set is used to find the best threshold on that set, and the same value is used for testing the model on the testing set. On a real hardware implementation, the threshold should be selected manually according to the required sensitivity, but tests can be conducted to find at least an average value to use as default starting point.

Regarding case 2), the One-Class Classification NN could in principle be trained without a validation set. We used the validation set to control overfitting and stop the training when no improvement were longer made, even if given the small size of training datasets this condition happened right after few epochs. In case of a real hardware execution and a truly one-class condition, the number of training epochs has to be determined in advance.

Regarding case 3), with OC-SVM the validation set has been used to perform hyperparameter search. Various SVMs were fitted on enrollment data, and performance according to hyperparameters was measured on the validation set. The combination of hyperparameters is the following:

```
    kernel = ['rbf', 'linear']
gamma = ['auto', 'scale']
nu = [0.0001, 0.001, 0.01, 0.1,
      0.0002, 0.002, 0.02, 0.2,
      0.0003, 0.003, 0.03, 0.3,
      0.0004, 0.004, 0.04, 0.4,
      0.0005, 0.005, 0.05, 0.5,
      0.0006, 0.006, 0.06, 0.6,
      0.0007, 0.007, 0.07, 0.7,
      0.0008, 0.008, 0.08, 0.8,
      0.0009, 0.009, 0.09, 0.9]

PCA_Enabled = [True, False]
```

```
pca_var = [0.05, 0.1, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.
                                        55, 0.60,
           0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 0.99]

# PCA can not be performed on a single sample.
if(len(x_train_svm)==1):
  PCA_Enabled = [False]
```

Where `nu` is the value of v as expressed in 4.5.4, `kernel` refers to the kernel function, and gamma is the coefficient in the kernel formula. As it can be noticed, PCA has also been tried with different values of explained variance. An optimal value has not been found, but the dimensional reduction can in principle help the SVM to distinguish between d-vectors.

List of Figures

List of Tables