Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer, Communication and Electronic Engineering

Final Dissertation

TinyML-Based Voice Recognition on Syntiant NDP101

From Keyword Spotting to Speaker Verification

Supervisor
Prof. Kasim Sinan Yildirim

Student
Matteo Gottardelli

Academic year 2024/2025

Contents

Abstract

ROWS TO ELIMINATE OR ADD:

TABLE OF CONTENTS: 1 page (COMPLETE)
ABSTRACT: 0/1 page (IN DEVELOPMENT)
1. INTRODUCTION: 1 page (COMPLETE)
2. BACKGROUND NOTIONS: 8 pages (COMPLETE)
3. KEYWORD SPOTTING AND SPEAKER VERIFICATION TRAINING: 5 pages
4. SW AND HW IMPLEMENTATION: (5.5/7 pages) (IN DEVELOPMENT)
5. RESULTS OBTAINED: (0/6 pages) (IN DEVELOPMENT)
6. CONCLUSION: 1 page
List of reference explanation:

- Edge Impulse:
  1. Edge Impulse Explaining how to manage Syntiant TinyML Deployment: [**?**]
  2. Keyword Spotting on Syntiant Stop/Go Example by Edge Impulse: [**?**]
  3. Firmware Syntiant TinyML Github Repository: [**?**]
  4. Processing Blocks Edge Impulse Code Processing (No Deployment Model Conversion code): [**?**]
- Tools and Datasets:
  1. Node Js Repository Download Source: [**?**]
  2. Dataset Edge Impulse Background Noise or General Words from Google Dataset: [**?**]
  3. Google Speech Dataset: [**?**]
  4. Netron Model Representation App: [**?**]
  5. PortAudio: [**?**]
  6. FFTW: [**?**]
  7. LibriSpeech: [**?**]
  8. TFLite PQT:[**?**]
  9. TFLite QAT:[**?**]
- Documentation:
  1. ESP32 Documentation: [**?**]
  2. Syntiant Tutorial Edge Impulse: [**?**]
- Syntiant Specifics:
  1. TinyML 2021 Summit Presentation Syntiant NDP120 -¿ Model Processing: [**?**]
  2. Data of Syntiant NDP101: [**?**]
  3. Code Experimental Implementation on NDP101: [**?**]
  4. The Intelligence of Things enabled by Syntiant's TinyML board analyzing performances: [**?**]

5. NDP101 General Usage: [**?**]
6. Description Syntiant Audio Block Processing: [**?**]
7. Hardware NDP101 Properties: [**?**]

- Theory:

1. Neural Network Theory: [**?**]
2. PDM Microphone Module Explanation: [**?**]
3. MFE Block Process Explained: [**?**]
4. TinyML Neural Network Training: [**?**]
5. Distillation Method from CNN to DNN: [**?**]
6. Knowledge Distillation: [**?**]
7. Understanding 4-bit Quantization: [**?**]
8. CNN Notions: [**?**]
9. PQT and QAT - Quantization Theory: [**?**]

- SV Model:

1. Deep Neural Network Model for Speaker Identification: [**?**]
2. D-vector Extractor implementing TinySV: [**?**]
3. D-vector Extractor implementing TinySV Code: [**?**]

1 Introduction

The microcontrollers (MCU) are computing systems, which are integrated in a larger system and in that case they are called embedded system, to perform a specific function that need a software implementation (programmed in C or assembly) and a hardware one (interconnectivity wires and sensor handle) They operate in a closed environment and elaborate environment inputs which may be visual, acustic and with more recent technology even tactile or movement and elaborate it to generate an output which may be an action in a bigger system, an audio response or a trigger depending on the microcontroller specifics. The programmer can implement a desired application for real time use, which allows personalization of the functionalities and the possibility of optimization. In an application development, the objective is achieving cost reduction in power and energy terms, the possibility of build a desired application and adding more than one feature at the same time connecting various sensors thanks to their peripherals.

1.1 TinyML Concept and Limits

MCU's technology made steps ahead in optimizing the computation velocity, meanwhile minimizing power consumption and a result is TinyML (Tiny Machine Learning). These devices enable machine and deep learning models to operate on a MCU, allowing performing actions Keyword Spotting, recognizing a specific word in an audio stream, or identifying objects in an image. These functionalities can be implemented thanks to a Neural Network, which is trained typically on cloud resources in Python programming language and this leads in performing only the inference phase on these tiny devices. This approach does not allow data exploitation directly, limiting incremental training or adapting algorithm through the device life. This is a limit on the Machine Learning side, but on the Tiny one there are some trade-offs. A direct consequence of being really small devices is having limited memory to reduce power consumption, so sometimes adapting a Neural Network which typically may occupy much memory isn't easy and requires precision reduction.

1.2 Goals - TinySV

This thesis studies how to adapt with a TinyML devices base system that performs Speaker Verification, which task consists in recognizing the identity of a user with references samples and comparing them with an input audio stream. The objective originally was creating a Keyword Spotting model (KWS) and a Speaker Verification (SV) one, trying to adapt that algorithm on two Syntiant TinyML NDP101 devices, but because of a NDA problem the access to documentation was inaccessible. The KWS development was possible thanks to Edge Impulse[**?**], but the SV approach[**?**] uses a technique not supported by Edge Impulse and because of the inability of accessing to a model compression tool. To preserve the initial idea, the model was tested as it would be a Syntiant TinyML NDP101, to show the validation of the technique. This thesis objective is to show the feasibility of this idea from a software perspective and partially hardware, using another MCU (STM32), with a single model verification. All the codes used in this thesis is provided on a GitHub repository[1]

1.3 Brief Summary

The thesis is divided into chapters. After this introduction, Chapter 2 aims to present theoretical concepts that will be used in this thesis, like Audio Processing, KWS and SV. Chapter 3 introduces the general methodology of the final objective and explains how it works. Chapter 4 explains the work flow of models training and optimization. Chapter 5 presents the software C code implementation for deploying both algorithms. Chapter 6 draws up the results obtained from computer testing.

[1] Thesis GitHub Repository - https://github.com/Gotta003/Thesis

2 Relevant Theoretical Notions

2.1 NDP101 Architecture Overview

The NDP101 is a microcontroller unit (MCU) developed by Syntiant[**?**], a company specialized in edge AI device development. "NDP" stands for Neural Decision Processor, an architecture tailored for deep learning algorithms applied to audio processing applications, like keyword speech interface, sensor recognition and speaker identification. The device is composed by two many components:
1. TinyBoard: Contains the CPU that handles the peripherals, containing the hardware part
2. Syntiant NDP101 Core: In this part, all the audio processing happens and it is where the Neural Network is stored. It is important to note that the DNN (Dense Neural Network) architecture that has to be stored inside the device is fixed. To it are dedicated 256KB with int32 bias length and int4 weights, which can be at most 589.000 total parameters in that memory location. The Neural Network for this device to be fast enough in computation avoids the use of CNN (Convolutional Neural Network), but can only support 4 Fully Connected Layers, 3 intermediate with 256 neurons and one for output with at most 64 output classes, using classification. To perform the internal software computation, there is an internal SRAM inside the chip which is a Cortex-M0 112KB size. This piece of memory contains the binary user's code along with global and local variables.

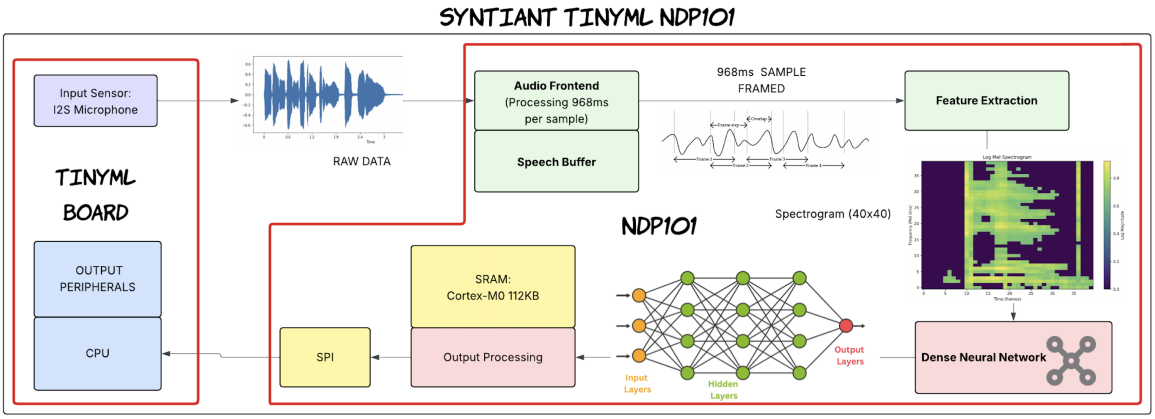


Figure 2.1: Syntiant NDP101 High-Level Workflow

A typical device behavior involves the use of an already integrated I2S microphone as an input sensor. If the MCU is active, it will always be on and it will perform a polling behavior. The input is processed by the Audio Front End, which will perform samples of 968ms, while the system feeds a 96KB speech buffer. These samples are given in input to the feature extraction that acts as a MFE Block Processing[**?**], using log-mel spectrogram. This spectrogram will always have 1600 features which will correspond to a 40x40 spectrogram image. This image is, then, processed by the Dense Neural Network. ALthough classification is the default behavior, developers can customize how outputs are handled using a custom Arduino IDE code.

2.1.1 Device Peripherals

In addition to the microphone input, the board includes:
• 9 pins: 4 for power supply and 5 for GPIO (general-purpose I/O)
• A Serial Flash Memory (SRAM)
• A micro-SD card slot used for memory extension, which, in this thesis, will not be used, but for big data storage and for the IMU functionality, supported by the device, it is required. It is estimated that with a 32GB card it will save more than 3 days of uncompressed audio data, with a frequency of 16kHz and with IMU more than 300 days of 6-axis sensor data with a frequency of 100Hz.

In the following image are shown the peripherals connections on Syntiant NDP101:

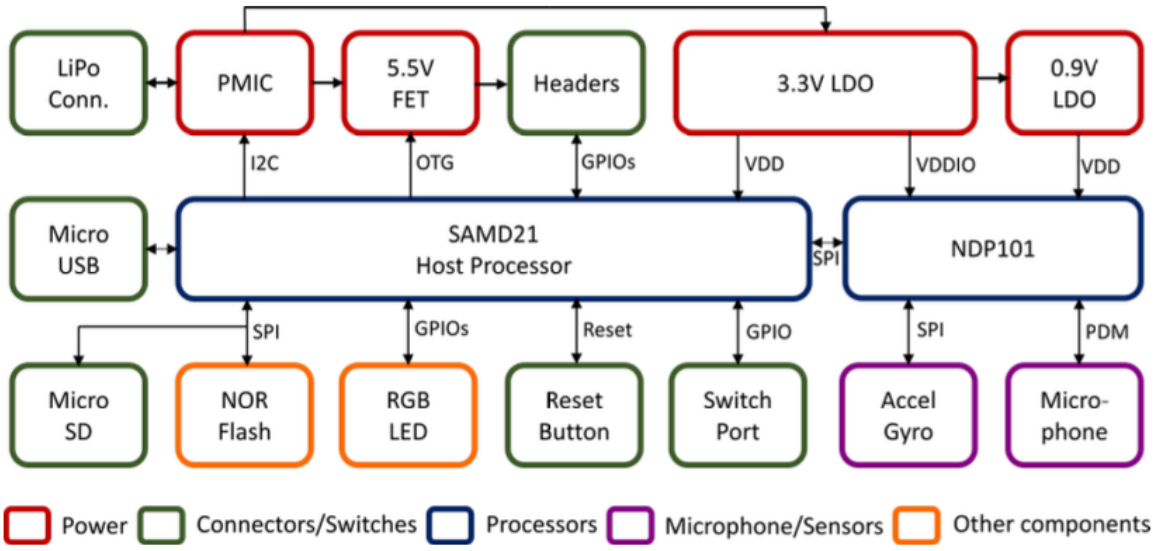


Figure 2.2: Design NDP101 with peripherals

Talk about power metrics, according to [**?**], NDP101 is an application for always on power consumption and manages to use it for audio / voice recognition applications 140μW. These results compared to others will deliver 20 times more throughput and 200 times less energy per inference. The connection with another MCU is possible with SPI communication; however, without the SDK provided by Syntiant, is challenging to program the setup of it.

2.2 Syntiant Audio Block Processing

Edge Impulse reports that Syntiant Audio Block Processing is similar to MFE one[**?**]. Its goal is to extract time and frequency features from a raw audio input. However, the block processing of Syntiant defers a little, because of a noise floor at the end of the computation. The block, which corresponds to the feature extractor, can be viewed as following:

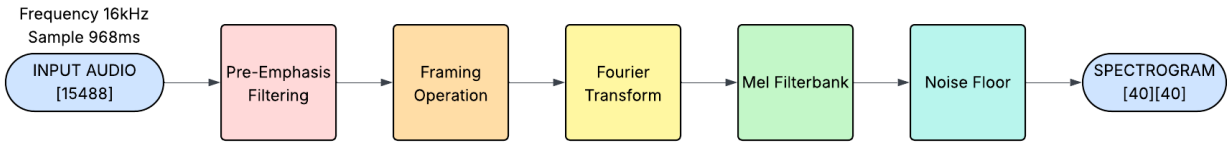


Figure 2.3: Syntiant Audio Block Processing

1. Input - The frequency of raw data input is at 16kHz and sampling 968ms, it generates 15488 raw data, with short values because the audio sound can go from -2^{15} to $2^{15} - 1$. These values come from an implicit ADC (Analog Digital Conversion), using a dual PDM microphone input which interfaces with I2S interface multiplexed with PDM[**?**]. It stands as Pulse Density Modulation and it reduces the system into a single-bit digital one. This allows signal processing operations to be performed on the audio stream easily and then the PDM can be modified by the system.
2. Pre-emphasis filter - This is a high-pass filter that enhances high-frequency components, so the microphone will capture more low-frequency noise and increase high frequencies to make the speech clear. Before this is needed an audio normalization [-1, 1], generating floats values and then apply this high-filter:

$$y[n] = x[n] - \alpha \cdot x[n-1] \tag{2.1}$$

α consists in a coefficient of filter grade and a standard Syntiant Block set it at 0.96875
3. Framing - This audio is split and segmented into small overlapping window called frames. Each

one has an overlap time with each other and in Syntiant corresponds to 128 floats, considering the size of 512 floats and the stride, how much the start position will move, of 384. In the last frame a part will overflow the initial buffer and in that case the void values are flatted to 0. Considering the input of 15488 samples in a 16kHz frequency:

$$number\,of\,frames = \frac{input\,size - frame\,size}{frame\,stride} + 1 = \frac{15488 - 512}{384} + 1 = 40 \tag{2.2}$$

For each frame of the forties computed are performed:
3.1 Windowing - Before performing fourier transform, it has to be applied a windowing to reduce spectral leakage in integration, so the following sinusoidal function is used:

$$0.54 - 0.46 \cdot (\frac{2\pi k}{size - 1}) \tag{2.3}$$

The size will be 512 and k is an incremental value that goes from 0 up to 511 and k-window is multiplied to the k-position of the array.
3.2 Fast Fourier Transformation (FTT) - This function computes the complex frequency spectrum of a real-valued signal captured in time domain. It is not required to compute all the DFT domain, because dealing with real value using Hermitian symmetry property with real values, it is required to compute only $\frac{N}{2} + 1$ unique complex outputs.

$$\begin{cases} X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{\frac{-2\pi i k n}{N}} & k = 0, 1, ..., \frac{N}{2} \\ X[l] = \overline{X[k]} & l = N - k \end{cases} \tag{2.4}$$

In this case, x[n] is the input signal with n=0,...,N-1, x[n]$\in \mathbb{R}$, i is the imaginary unit, k is the incremental value and N is the length of the signal (512 values).
3.3 Spectrogram Population - Corresponds to a magnitude computation of the FFT output, obtaining the amplitude spectrum from the complex frequency-domain data. This computed the norm of the first half of the fourier transformation output and saves it in the corresponding size [40x256], the magnitude is computed as follows:

$$|X[k]| = \sqrt{(Re(X[k]))^2 + (Im(X[k]))^2},\ k = 0, 1, ..., \frac{N}{2} - 1 \tag{2.5}$$

4. Mel-filterbank - After obtaining this matrix, the algorithm applies a mel-filterbank, which is a set of data based on human perception system via a triangular bandpass filter, making the system more sensitive to low frequencies. It is used mel-scale to obtain this phenomenon, because using a logarithmic approach allows a better sound recognition.
This implements a K+2 length filter called m, composed by linear spaced elements, with K the number of filters.

$$m[i] = M_{min} + i \cdot \frac{M_{max} - M_{min}}{K + 1},\ i = 0, 1, ..., K - 1 \tag{2.6}$$

M_{min} and M_{max} corresponds to the conversion in mel-scale of the minimum and the maximum frequency. Syntiant as default has set the minimum on 0 and the maximum on $\frac{f_s}{2}$. After the information is reconverted back in the frequency scale. The conversion formulas are for Syntiant:

$$m = 1127 \cdot log_{10}(1 + \frac{h}{700}) \quad h = 700 \cdot 10^{\frac{m}{1127}} - 1 \tag{2.7}$$

To, this scale is applied a triangular filter for each filter between 1 and K, using bins covering frequencies. The computation of the bins and of the triangular function is:

$$b_i = \lfloor \frac{2 \cdot f_i}{f_s} \cdot (N - 1) \rfloor \quad H_k[b] = \begin{cases} 0 & b < b_k - 1 or b > b_k + 1 \\ \frac{b - b_{k-1}}{b_k - b_{k-1}} & b_{k-1} \leq b \leq b_k \\ \frac{b_{k+1} - b}{b_{k+1} - b_k} & b_k \leq b \leq b_{k+1} \end{cases} \tag{2.8}$$

This computation creates a matrix 40x40, which corresponds to the filterbank filter that will be applied to the magnitudes matrix to obtain a log-mel spectrogram:

$$L(i,j) = 10 \cdot log_{10}(\sum_{k=0}^{NUM\_BINS-1} S(i,k) \cdot M(j,k) + \epsilon) \;\; i = 0, ..., N-1 \;\; j = 0, ..., F-1 \;\; \epsilon = 10^{-20} \quad (2.9)$$

The computation will generate a 40x40 matrix that will be the form of the image to be fed into the neural network. To the computation is added a small constant error value to avoiding log(0) impossible result, L is the log-mel spectrogram, M is the fixed filterbank and S is the spectrogram of the magnitudes
5. Noise Floor - To only maintain audible sound is applied a threshold noise floor flatting all normalized values below 0.65, which should be resized according to the noise floor in decibel of the system which for Syntiant is set to -40dB. The formula to perform this is:

$$final(i,j) = \frac{L(i,j) - NOISE\,FLOOR}{-NOISE\,FLOOR + 12} \;\; i,j = 0, ..., FILTERS \quad (2.10)$$

To be accepted final(i,j) should be ≥ 0.65 and the overall final matrix is the input given to the neural network.

2.3 Neural Network Concept

2.3.1 Dense Neural Network (DNN)

Neural networks consist of interconnected layers organized in a logical architecture, having inside of each layer an predecided number of neurons[**?**]. These are like nodes that connect two nearby layers. Each receives an input signal and applies to it an activation function, generating an intermediate output that will be passed to the subsequent series of neurons to the other layer. The connections between neurons are weighted, so there are some values associated to the layer that represent the influence of these connections that will modify the input to generate an output. The weights can be adjusted only during training phase and they characterize the neural network behavior and picking a layer as reference it patterns the connection of each input with each output, generating an allocation in memory equal to "input size· output size". Typically, to each layer is associate a bias array, always modifiable only during training, which performs an adjustment to output of the neuron. It is like a simple addition, so to each the array is big as the output size to have an adjustment value for each output.

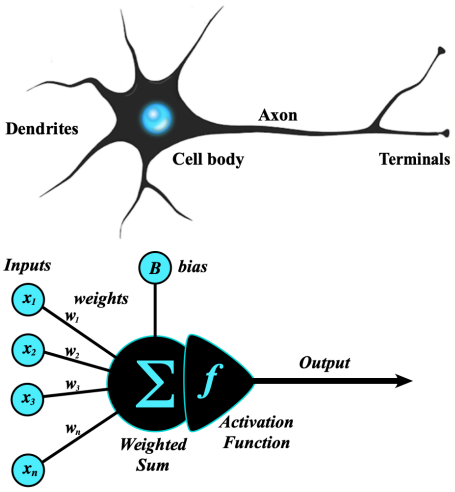


Figure 2.4: Neuron structure

To express the neuron mathematically, can be defined an input matrix x[N]=$[x_1, ..., x_N]^T$ as Nx1, the weights that involves only that output neuron, because it is a big array with size "in_dimension·out_dimension" and we can express it as matrix w[M]=$[w_1, ..., w_M]^T$ Mx1, b the bias associated to that neuron, $\phi(\cdot)$ the activation function and y as the output of the neuron, that mathematically will result in:

$$y = \phi(w^T x + b) = \phi(\sum_{i=1}^{n} w_i x_i + b) \quad (2.11)$$

More generally, the formula that will represent the output array of a layer will be considering this time the W$\in \mathbb{R}^{O \times N}$ as OxN matrix, the biases b=$[b_1, ..., b_O]^T$ as Ox1 matrix and the output as

y=$[y_1, ..., y_O]^T$ as Ox1 matrix. Results in:

$$y_i = \phi(Wx + b) = \phi(\sum_{i=1}^{n} w_i j x_i + b_i) \; i = 1, ..., O \rightarrow \begin{cases} y_1 = \phi(w_{11}x_1 + w_{12}x_2 + ... + w_1 N x_n + b_1) \\ y_2 = \phi(w_{21}x_1 + w_{22}x_2 + ... + w_2 N x_n + b_2) \\ \dots \\ y_O = \phi(w_{O1}x_1 + w_{O2}x_2 + ... + w_O N x_n + b_O) \end{cases} \tag{2.12}$$

The architecture of the neural network is composed by different layers according to their functionality principles. The most notable ones are:

• Input layer: Consists in the input of the system in the case of Syntiant neural network a spectrogram 40x40 flatted into a 1600 features array. This layer is only nominal and it does not perform any computations.

• Intermediate layer: A neural network can have one or more of these that are between the input and the output layers. The decision of the number of these depends on the memory space and application specifics, but typically more layers equal to a more complex network structure. It is characterized by an activation function typically a ReLU. On the Syntiant NDP101 device there are 3 intermediate layers each with 256 neurons

• Output layer: Consists in the final output of the network and the output depends on the behavior of the network, if it is a regression like model it will output inside a neuron a value using an activation ReLU, but if it performs a classification it means it is a multiclass model and in that case is recommended the Softmax activation to identify the probability of belonging in each class. Syntiant device can have at most 64 neurons as output, with the above neuron configuration.

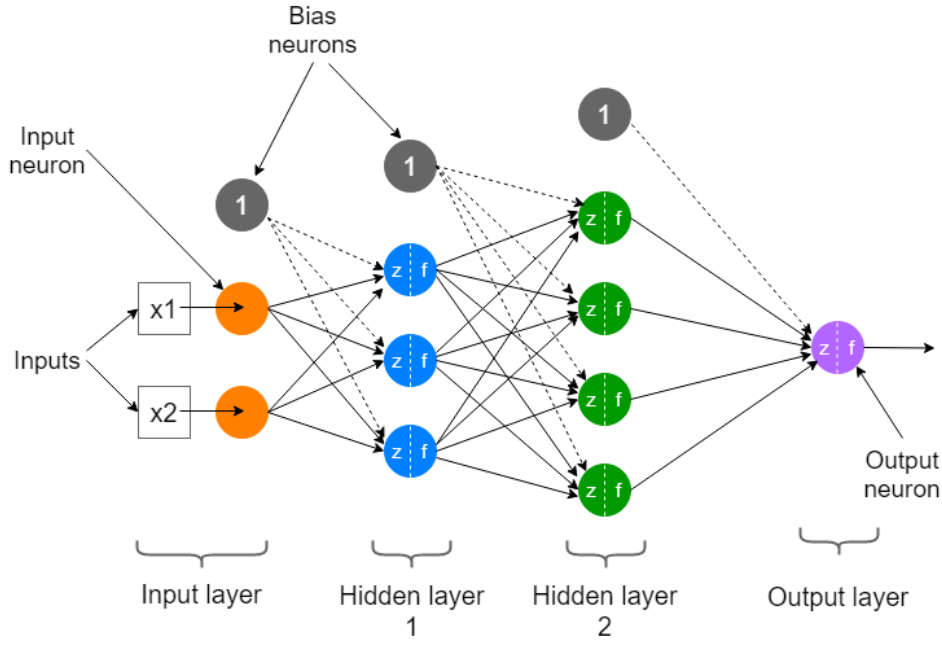


Figure 2.5: Neural Network Structure

For this thesis, we are interested in only two activation functions, already nominated:

• ReLU (Rectified Linear Unit): Performs a threshold to bring negative values to 0 without touching the positive ones. It is used for is cheapness, because it performs a maximum evaluation with 0. In general, the layers relies on this option:

$$f(x) = max(0, x) \tag{2.13}$$

• Softmax: Used for classification, it converts the output value of the layer in scores with a probability distribution by taking the exponential of each output and normalizing these values by dividing using the sum of all the exponentials. As a counterpart, if summing all the values in the output layer array the elements, the sum must be 1:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.14}$$

2.3.2 Convolutional Neural Network (CNN)

Dense Neural Network has each layer's neurons connected with all the neurons of the subsequent layer, but there are other neural network called convolutional, which are more energy expenses and computational complex and operate in 2D operations with the objective of applying small filters to scan the input image, that in the case studied would be a spectrogram, that extracts the most prominent features of each area.
Here, will be introduced the essential notions used in this thesis in CNN field.
The architecture of a standard CNN[**?**] is composed by:

1. Input Layer - Typically, a CNN receives in input cubes images, with three dimensions (height, width, depth), but in our case, the depth side is irrelevant, because the spectrogram is a 1-depth image, requiring only width and height (40x40).
2. Batch Normalization Layer - It applies a normalization to the input normalizing input to zero-mean and unit-variance. It stabilizes and accelerates training, helping a faster learning. The normalization is performed only during training, instead for model usage are training two trainable parameters one for scaling for a value γ and one for shifting for a value β.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma \cdot \hat{x}_i + \beta \tag{2.15}$$

The parameters present are:
• x_i - Input Layer
• y_i - Output Layer
• μ_B - Mean computation on a subset B. B corresponds to batch being the number of samples condensed in an average behavior minimizing errors
• σ_B - Variance computation on a subset B
• ϵ - Small constant error

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} x_i \quad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_B)^2 \tag{2.16}$$

3. Convolutional Layer - It extracts spatial patterns, from the input layer and computes only on that submatrix the neuron operation of DNN, computing a weighted sum using the kernel that will be composed by weights, adding the bias and applying a ReLU activation. It may reduce the dimension with downsampling or it may be the same, leaving the job to max-polling layer, instead adds channels.

$$y_{i,j,k} = \sum_{m=0}^{K-1}\sum_{n=0}^{K-1}\sum_{c=0}^{C-1} \omega_{m,n,c,k} \cdot x_{i+m,j+n,c} + b_k \tag{2.17}$$

K corresponds to kernel size, instead C to the number of channels.
4. Max-polling Layer - On channels, so to the whole image directly, reduces height and width by a window sliding for a pool size and among the values takes the maximum one. It provides spatial invariance reducing small translations and reducing overfitting by downsampling. Overfitting phenomenon is when model learns the training data too well, including noise, outliers rather than underlying patterns that generalize the input data. The phenomenon is seeable if the model performs good training, but fails in validations.

$$y_{i,j,c} = max_{0<m<P}\ max_{o\leq n<P}\ x_{i\cdot s+m,\, j\cdot s+n,c} \tag{2.18}$$

P is the polling window size, s the stride and max the maximum value in the pooling region
5. Fully Connected Layer - To recognizing the belong to a particular class, it requires at least on FC layer to perform a classification and it is like in DNN

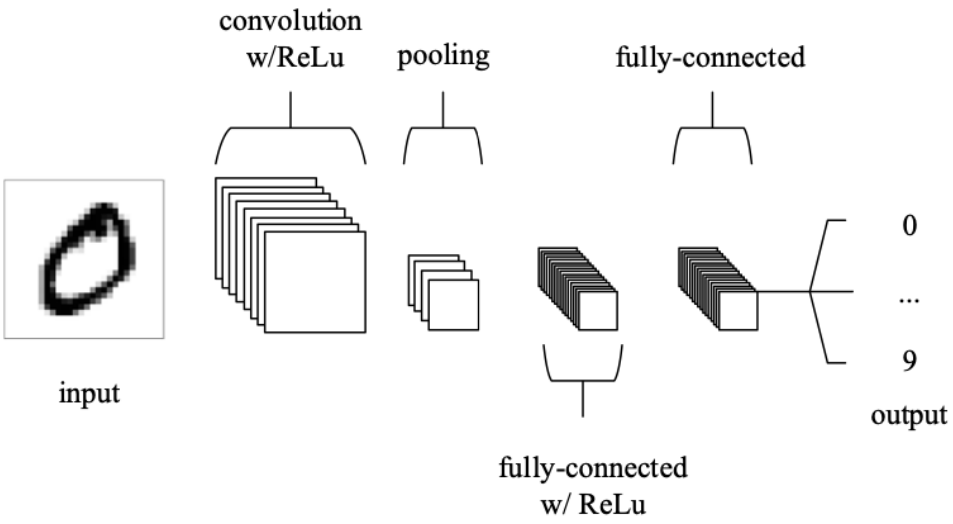


Figure 2.6: Convolutional Neural Network Architecture

2.3.3 Neural Network Parameter Performance

As follows, are listed some notions that will be recalled when talking about models and results:
• Accuracy - Measures the proportion of correct predictions over all predictions:

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

• F1 Score - Harmonic mean of precision and recall, balancing false positives and negatives:

$$F1 = \frac{2TP}{2TP + FP + FN}$$

• Loss - Represents the prediction error:

$$\mathcal{L} = -\sum y \log(\hat{y})$$

• AUC (Area Under Curve) - Evaluates classifier performance across all thresholds via ROC curve:

$$\text{AUC} = \int_0^1 \text{TPR}(FPR)\, d(\text{FPR})$$

• EER (Equal Error Rate) - The rate at which the false acceptance rate equals the false rejection rate:

$$\text{EER when } FAR(t) = FRR(t)$$

• EER Threshold - The decision threshold value at which EER occurs:

$$t_{\text{EER}} = \arg\min_t |FAR(t) - FRR(t)|$$

• Cosine Similarity - Measures how similar two vectors are:

$$\cos(\theta) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}|| \cdot ||\vec{y}||}$$

2.4 Speaker Verification Approaches

The goal of the thesis is proposing a text-dependent implementation on Syntiant NDP101, leading that for each word said by a user has to have a reference sample. From this comes the necessity of having a keyword spotting algorithm to recognize a word and the speaker verification algorithm for extraction of relevant features of the sample and comparison with the word-user reference.

The purpose of speaker verification is to identify if a user is part of a dataset or not. The networks can be trainable with specific user samples, bringing the disadvantage to require retraining each time a user is added to the system, so during inference and product release this approach is not recommended. Another solution is choosing a one-time training approach, providing a large dataset of various people. It will generate a feature extraction, which compared with a reference array with equal size can provide the cosine similarity parameter. For application, the second one is the best, but requires more training, a variety amount of data and a good amount of samples, resulting in a more difficult implementation. Another key difference of the model stays in its text-dependency. If the model is text-dependent, the references samples of two different words of a same user will be different ones, requiring more memory space allocation with more words are required, instead a text-independent approach, which is difficult to achieve without retraining, consists in a memory optimization because the user does not care about saving samples of the words he says, but relies only on his data reference. As a fact, the no-retraining required gives to the model feasibility, instead text-dependency will provide more accuracy on single words, but will have more memory expenses. It is not our case, but if dealing with a system, not requiring user addition in inference, a solution may be relying on a retraining approach and text-independency. This may be possible only if a large person dedicated dataset is provided. There are some relevant solution, to implement a SV model, however, no one is really deployable in a Tiny system, except for a d-vector extractor technique, relying on finding patterns in voice before the classification process.[**?**]

2.5 Keyword Spotting Approaches

The goal of the algorithm is training a neural network to recognize the belonging or not to a class output independently from the user. The class can be trained on a sample word or a short phrase. To perform the task efficiently on Syntiant NDP101, it is recommended to develop it using Edge Impulse Framework, which allowed a good compression and validation process of the model. The limitations rely in retraining, because in performing a text-dependent approach to perform a classification the system should be aware on how a class is composed and characterized. Complex and optimized methods already exist for Embedded Systems, like RNN-based model. This approach uses a Recurrent Neural Network[**?**], which processes a word on character level and, unlike SV models, the various solutions can be deployed for TinyML devices. Edge Impulse, as default, provides directly a model structure compatible with the MCU, because NDP101 architecture is very limited, but it would not rely on a complex datasets and it is a more feasible solution, unlike SV model.

3 KWS and SV Models

3.1 KWS Model

Keyword Spotting (KWS) is a model specialized in audio classification whose objective is to detect the presence of a spoken word or phrase according to an input sample, in this case the spectrogram extracted by MFE block. This transformation from raw data to spectrogram provides a time-frequency representation that emphasizes relevant acoustic features. A KWS model is trained to capture a predefined keyword by learn its spectral patterns across a range of samples, which must be differentiate and of a big size to ensure generalization across speakers and noise conditions. In this thesis, the KWS model is destined for deployment on the Syntiant NDP101 hardware. Due to the hardware constraints and design recommendations given by Syntiant, a DNN architecture has to be employed with a maximum output of 64 classes. This network has 3 hidden fully connected layers and one output layer, used along with a softmax activation, mapping the feature vector into a probability distribution over defined class labels. The neural network structure strikes a balance between efficiency and classification accuracy, making it suited for real-time applications on embedded devices. The focus in this work is on system verification, allowing to make a simple model consisting of one word, basically creating a binary output, that word or not. The chosen keyword for training is Sheila, chosen due to its availability within the Google Speech Commands dataset [**?**]. This offers a substantial number of audio recordings designed for keyword classification research, including various samples from multiple speakers with varying environmental conditions. The recordings in the dataset align with the Syntiant NDP101's input specifications, because they are sampled at 16kHz and have a one second duration per sample. In addition to the sheila's samples, a good dataset should include a variety of background noise sounds, samples of words non present in the dataset and others similar phonetically, which may look similar for the neural network perspective. To give more samples as possible to the dataset, some audio clips were manually recorded or sourced from a dataset curated by Edge Impulse [**?**], which is a platform specializing in machine learning for model developing and deploying on edge devices. It is important to note that in the dataset provided there are some Sheila samples, so to avoid any error in splitting is suggested a checking. The final dataset used consists of 3403 sheila-words (almost 56m 43s of audio recoding). These samples were divided 80% for training and 20% for evaluation. This division ensures that meaningful representations are learnt while it is still providing independent data for validation. To facilitate deployment on NDP101, the model is developed and trained using the Edge Impulse Studio[**?**], which provides a good and intuitive pipeline for building and optimizing models tailored to the embedded hardwares. The processes done by this framework include automatic data preprocessing, training, fine-tuning, and a quantization model conversion to int4, which is compatible with the Syntiant device. This workflow reduces development time, ensuring at the same time that the resulting model fits in the memory. During and after training, confusion matrices are generated for classification performance. These provide insights into which keywords were misclassified, highlighting potential areas of confusion and suggesting model refinement. They are useful in identifying failure cases, such as a confusion between similar-sounding words and to avoid this inconvenience, these words were already added to the dataset as other.

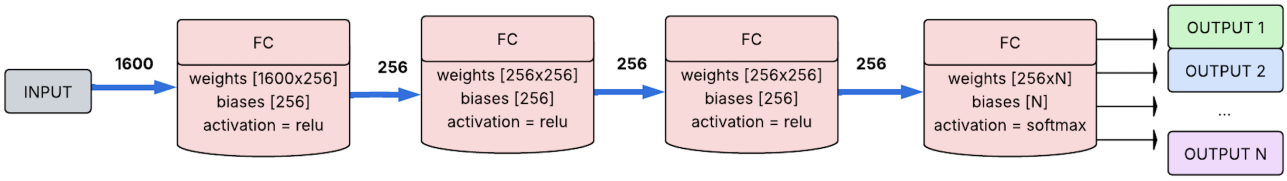


Figure 3.1: Keyword Spotting Model

3.2 SV Model

After creating the KWS model, the objective is creating a text-dependent Speaker Verification model, which requires only one train. This is known as ASV (Adaptive Speaker Verification), which relies on comparing the results of the model (d-vectors) with reference samples stored in system dataset, which have to be captured during inference phase. Speaker Verification is used in recognizing the identity of a user in a on-device learning context. To approach this way a large amount of data is required at first to train the model, performing a meticulous extraction of d-vectors to recognize patterns in user voice and possibly trying to minimize the required number of samples, maximizing the security recognition. A sample, because of text-dependency, will correspond to a word said by one user and should compare only with its similar.
The deployment of the model requires 3 caveats:
1. Adapting directly on device, meaning that a new user should be able to enroll in SV application by providing samples of its voice in real time through the target device
2. The algorithm has to operate in one-class manner and it should be able to learn to distinguish between the enrolled user and the others, only using the data from the dataset collected during inference
3. To be fit in a TinyML device the considerations should be done in memory allocation depending on the device used, so the model should fit in Flash Memory
To obtain the desired d-vector we should use a convolutional neural network, because we would like to reduce dimensionality, while obtaining significantly features. This is achieved with some expensive filters that while reducing width and heights, will generating new channels corresponding to a new feature extracted. This is a way to synthesize the input spectrogram (40 width x 40 height x 1 channel) to a lower dimension.
A valid alternative, in theory, could be i-vectors. It is a feature that represents the characteristics of frame-level features distributive pattern. The extraction is a dimensionality reduction of GMM supervector, allowing an extraction per sentence, instead d-vector generates one-hot speaker label on the output and it is an averaged activation from the last hidden layer of CNN. The advantage of d-vector is that there is no assumption on feature's distribution, instead i-vector assumes as default a Gaussian distribution.

3.2.1 D-Vector Extractor Creation

The structure of the CNN follows the theoretical one introduced and as input is given the spectrogram given in output from MFE block (40x40x1) and as output the objective is a 256-size long array of relevant features (d-vector). However, it cannot be the model output, because at first the model should classify the data given in input, through a fully connected layer. As input dataset is used a huge one with many different speakers and each providing various samples, because they comes from audio book recording. From speakers classification is not required the text-dependent approach, so the speakers will say many different words, which helps in generate the required weights and biases for the convolution layers. The dataset used was from libreSpeech which has data collected per speaker and for the purpose was taken the one with 100 hours clean speech in English Language sampled with 16kHz, which is the same of Syntiant audio processing[**?**], corresponding to 6GB memory space to make the training successful in reasonable times.
A first problem is that these samples are not already in 1 second, but variable, so they have been sampled and the parsed through the MFE Block previously introduced. The total spectrograms obtained were 136112 for a total number of classes of 94 and a total of 38 hours of recording and a memory occupation of 871MB. To have a balanced dataset all classes had the number same samples 1448 as fact at first should have been 100, but they had too few samples. A convolution Neural should perform 3 actions during its creation in which the dataset should subdivide its samples:

1. Training - The spectrogram is threated as an image, this dataset is the base of the model and on these the weights are adjusted using backpropagation, like Adam optimizer, and using loss func-

tions like cross-entropy, for classification, or MSE, for regression, to minimize the failure rate. It uses a learning rate, which is a hyperparameter, determining the step size at each iteration while moving forward with epochs. A epoch consists in a cycle of training input processing and an evaluation and can be arbitrary set according to the complexity of the network. In this case, it was set to 700. 2. Validation - This is used to see the result on data other than the ones in training set. It is used to adjust learning rate and batch size in case it starts overfitting or perform an early stopping if validation loss increases.
3. Testing - It is a dataset to which are added background noise, varying microphone quality and other modifications to the original input. It is like a validation, but is to stress-out the system and seeing if it can still working with some fluctuations.

The distribution among these 3 sets is random inside a class, but each one will have an amount in each set. For precision, 70% of samples will go in training (95278), 15% in validation (20417) and 15% in testing (20417). In training with CNN is suggested to avoid fluctuations a batch size and considering that the samples may not be complete words it performs an attenuate on those cases and the objective in having a good identification of the speaker. Knowing that a higher batch size will provide more accuracy, it was chosen 32 size, which should be stable with standard learning functions. The setup of the model, as summary, is an input shape of (32,40,40,1) [batch,width,height,channel] for 2977 inputs (95278/32) and output shape of 94 classes. The model creation required about 3 hours using a GPU. The proposed architecture is the following[**?**]:

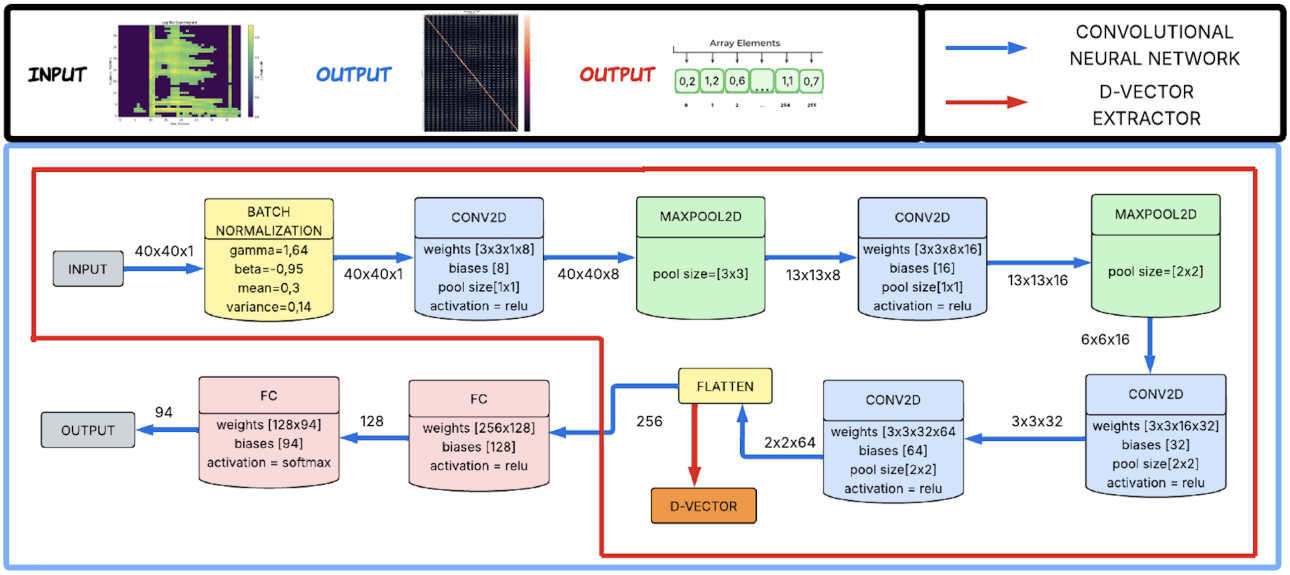


Figure 3.2: Speaker Verification Neural Network and D-vector Extractor

After obtaining the CNN, we implemented the Fully Connected Layers to catalog and what is expected is even with 700 epochs and 32 batch, to have a good accuracy, but a bad loss, because we are trying to find patterns between very different samples. However, before the classification, so the Fully Connected layers, there is a generalized array of 256-size that represents the concept of d-vector, a general array that is a compression of the initial spectrogram and with max-pooling leverage relevant features, preserving all its consistency. Knowing this, we could truncate the DNN part of CNN and set the flatten section as the output and it is possible because the two logic are separated.
The problem that will arise now is how can the accuracy be computed, because before it all depended on classification, but truncating that last part is required another method to capture that value and because this is an ASV model, it will not be trained again and if it was for a fixed user for a specific usage it would have been reasonable, but in this case it is not the solution. A notion that was introduced before consisted in cosine similarity. It takes two vectors and gives a percentage output on

how much they are similar. The model that has just been created can provide a relevant reference of the audio sample given in input and if in a dataset a similar one is provided, even if the spectrogram are not identical with feature compression and filtering they will be very similar. There are some existing solutions on which the evaluation of the models was performed:
1. Best-matching: During inference, the reference samples have to be recorded in real-time. It is recommended, because of this, to accept more than one sample and typically increasing the number of samples saved, more it is probable to find similar user samples. The best-matching technique saves per word said per user a number of vectors equal to the recommended size. It is important to notice that because the d-vector elements are floats, a single reference vector occupies 1KB.
2. Mean Reference: Instead of saving all the reference samples, occupying N KB per user's word enrolled, to save space can be computed an average one. Theoretically, we would not have an optimal cosine similarity, however it is a good trade-off in space saving and considering the operation on a TinyML the minimizing of storage and memory allocation is important.
In the case of Syntiant NDP101, because of the really low space is highly recommended to use mean reference technique to minimize power consumption.

3.2.2 Knowledge Distillation Training

If the solution may be deployable on some TinyML, it is not optimized and not compatible with Syntiant NDP101, because it can only support dense fully connected layer (DNN), but the d-vector extractor is a CNN[**?**][**?**]. Exists in machine learning a process called Knowledge Distillation. It adapts two different models with even different architecture, but with equal input and output data. To perform this, there should be a teacher and a student model. The teacher is CNN and is taken and the student is DNN that tries to replicate the results of the teacher at feeding of input data. The orientation of the student will be the loss discrepancy being 1-cosine similarity. To evaluate the model can be used both cosine similarity, but Mean Square Error is a solution, too. The advantages of this kind of approach is not only for deployment on NDP101, but dense layers have an easier computation than convolutional layers, which require more power consumption. As a consequence, the model would be faster. However, it should require more parameters to obtain similar results, because 3 hidden layers structure remains fix. In definitive, the model would be less precise and occupy more memory, but will be more faster and deployable on NDP101.

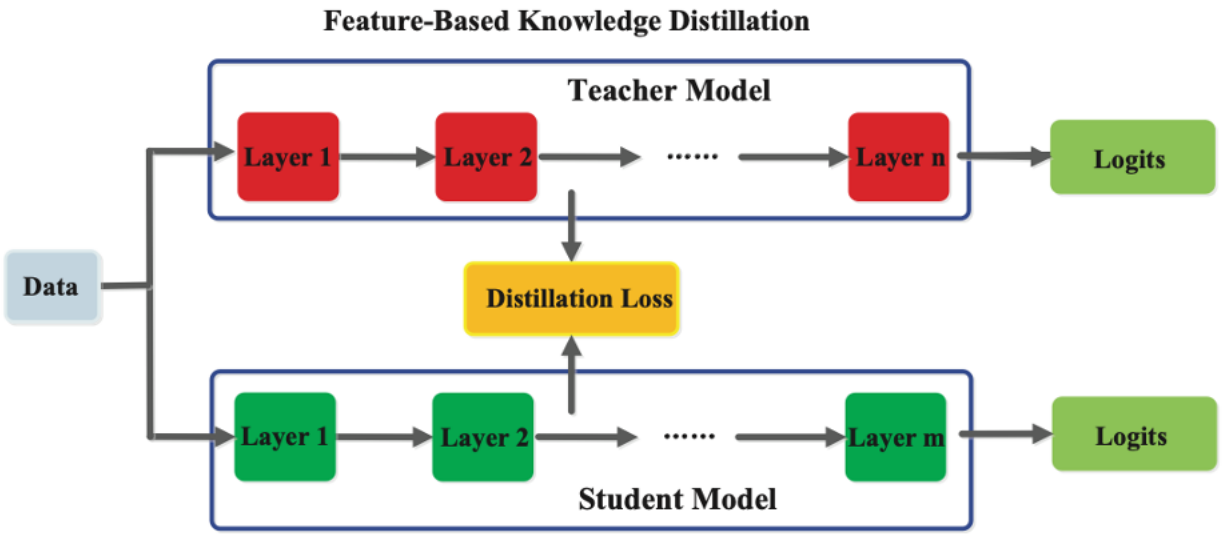


Figure 3.3: Distillation Model from CNN to DNN

The number of neurons per layers have to be chosen by the user, but there are some caveats to consider:

• Syntiant NDP101 supports at most 589.000 parameters, so the neuron choice must be taken in account
• Downscaling and then upscaling between two layers is not recommended and there should be a reduction in results performances
• The neurons typically are multiples of 2 to optimize space, in case that would be too inefficient is recommended to take at least multiples of 2's multiples.

3.2.3 Quantization of SV Model

Unfortunately, the problems do not end here. Syntiant NDP101 not only has a DNN, but requires a parameters' quantization, too. The weights should be stored in 4-bit[?] integers, meanwhile biases in 32-bit integers. Performing a quantization is the most straightforward way and in case of transposing it to int-8 it is immediate, thanks to support provided by Edge Impulse with Post-Quantization-Training (PQT)[?], which is the case considering that the distilled model is trained. However, the int-4 quantization is not optimized and as of now consists in a fake int-4 quantization, like storing in int-8 bits allocation int-4.[?] But, can be seeing that PQT int4 weights has really poor performances. Instead, and performing a Quantization-Aware-Training (QAT)[?] as of now can achieve decent results, always using fake-quantization. The problem is that it's unknown how the Syntiant binary is built and composed, so even with a good packing-unpacking technique, it could be difficult to obtain something from the model. Syntiant NDP101 provides a tool with the SDK to convert a model into binary Syntiant compatible, but it is under NDA. A solution could be using using Edge Impulse Framework, but it supports only classification and regressions models as of know and it is not what the d-vector extractor is performing.

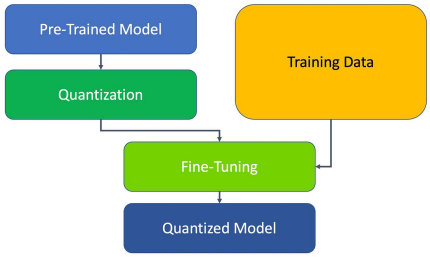


Figure 3.4: Quantization with Tensorflow

However, in theory, to successfully quantize the model it has to be performed a brute force quantization and then a fine-tuning to adjust values passing database data. An advantage of this, is the size of the model, not only the weights are adjusted, but the intermediate input and output layers, too, leading in having an output no longer of 1KB, but of 256 bytes. It will save even more space in memory, however to be sure to convert the input spectrogram in int, the model can be fake-quantize to leave the input and the output as floats adding a quantize and dequantize layer, the ones inside will be int8 and then remove manually from dequantize layer. This will allow to maintain the float spectrogram logic, but at the same time allowing inference on Syntiant, if the conversion tool was available and saving memory because of the reduction of 75% per sample.

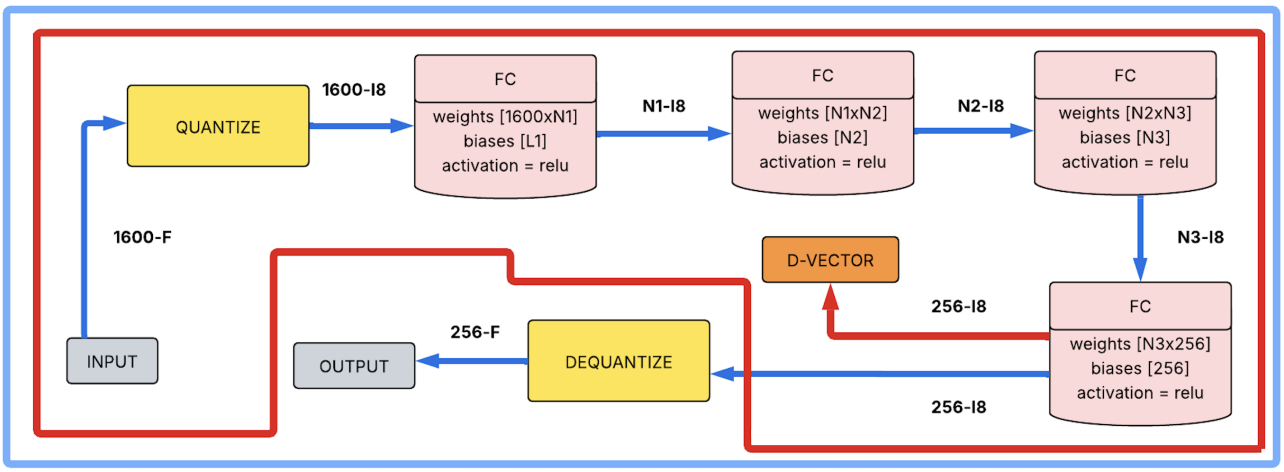


Figure 3.5: Quantized Model Int8

4 SW and HW Implementation

The overall system will work as follows:
1. The microphones capture the user voice and save their in the buffer
2. When the buffers are full in the respective devices, the KWS action performing device, will process the input in Syntiant Pipeline, meanwhile SV performing device will wait for a response
3. The KWS routine finishes and it communicates the classification output to SV device via SPI
4. The SV according to the output if it is a class computes the Syntiant pipeline
5. After d-vector extraction, it compares the result obtained with the reference vectors of the corresponding word, limiting unnecessary comparisons.

The possible outputs of the system are:
• No word recognized, both devices return sampling
• Is recognized a word, but the user is not enrolled for that word
• Is recognized a word and the user is enrolled
According to these different outputs, the programmer will be free to perform a connected action.
This thesis, at first, was developed considering the system implementation on two Syntiant NDP101 devices with the objective in creating a multi-model system. Ultimately due to a NDA was not possible to deploy, the overall idea will be presented as if the access to the SDK was granted. A multi-model system consists in having a uniformly signal processing with an appropriate reshape and MFE block processing by a NDP101 and then only then the result will be parsed to the other device to perform a different model action. During the dissertation of the methodology used in software pipeline, there were used two different approaches:

• Simulation on computer (Software Approach) - Created to verify system pipeline correctness before deployment. It does not use any hardware component, except for the microphone integrated in the computer. It shows the behavior using pure C code and saving the models in header
• Inference on MCU (Software + Hardware Approach) - Actually application deployment, which require handle of hardware components. In this case, the models are uploaded in binary files, but each one on a different MCU, not like the simulation, which had all accessible by the same compiler. The hardware and the communication (SPI) had to be managed, but the other phases are the same edited in C in simulation.

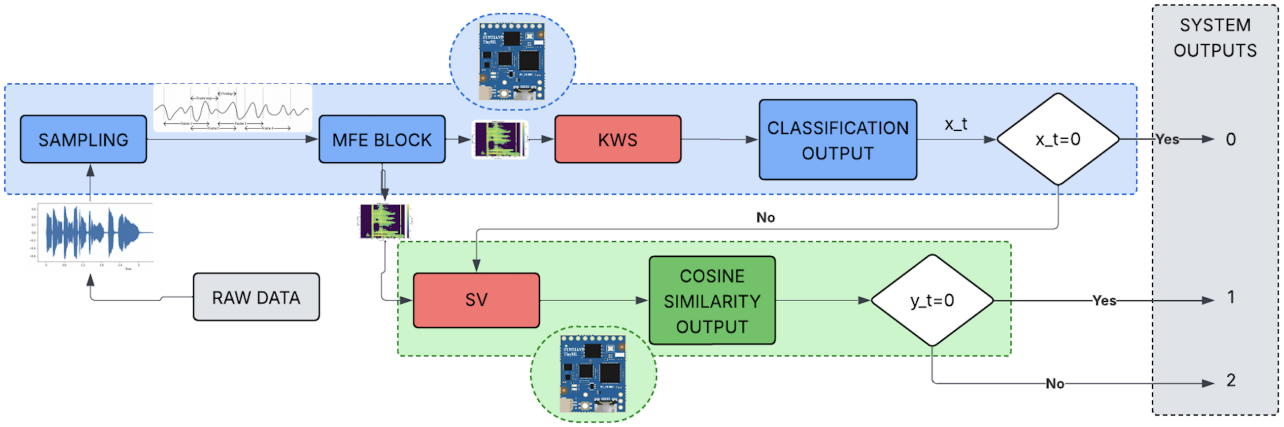


Figure 4.1: Software Pipeline

4.1 Software Pipeline

The system was developed in simulation and in validation using a simulation software approach, instead in inference only the code logic inside the single NDP101 was developed. In this section, we will talk about:

- Signal Capture (Simulation)
- MFE Block Generation and Processing (Simulation+Inference[1])
- Models processing (Simulation)
- Output Elaboration (Simulation+Inference)
- Enrollment (Simulation+Inference)

4.1.1 Signal Capture

The model requires real raw data to work. If on Syntiant NDP101 there is a dual microphone integrated that computes the ADC conversion, on software were created 3 different modes to handle the code, according to the needs:

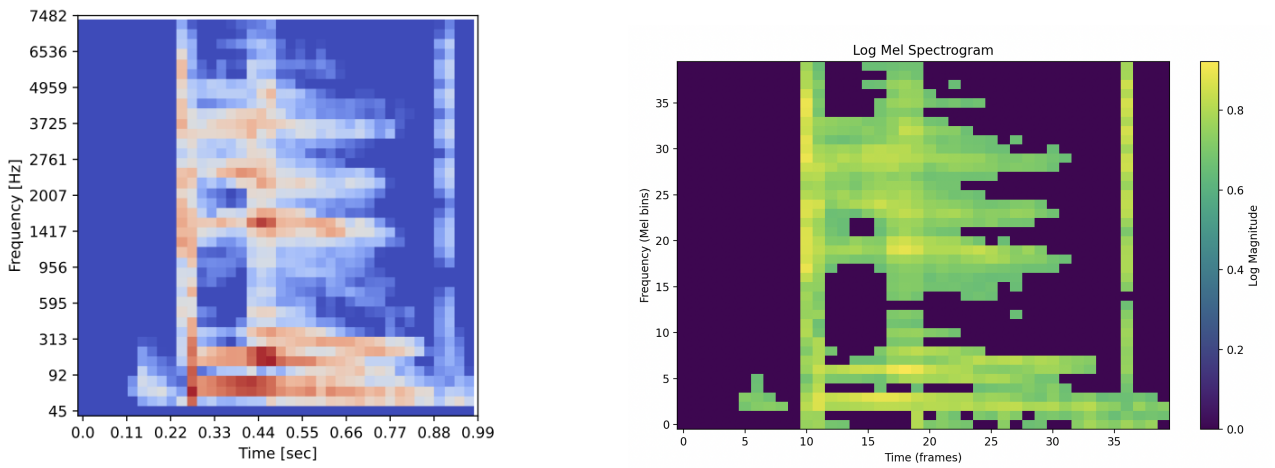
1. Live Sampling Mode (Mode 0) - The performing of live sampling from real input was performed to do a fastly debugging, without having to upload and create different files each time. The idea is to be a one shot code, so when launched it activates the audio capture for one second and then use that input audio as if it was the raw data inputted by the microphone. To do so, it was used an external library called PortAudio[**?**], which manages the input audio flow, starting it, stopping it and then calling a callback function to trigger the rest of the system.
2. Files with .wav extension (Mode 1) - To perform a validation, like a confusion matrix, should be parsed multiple files, which could not be done with live-sampling. The idea was to create this mode that takes in input a .wav file that with appropriate manipulation would give the same output as PortAudio, but it can be reiterated. More precisely, using a bash file, can be given in input a folder and the program will be called a number of times equal to the .wav files present in it. But at the same time can be parsed a single file. This is the primary technique used in graph performance model validation
3. Elaboration using data stored in a header (Mode 2) - The most primitive technique among the three proposed, but was useful in code creation and verification and it consists in copy and paste the digital values into a header file.

The properties of the audio computation was adapted from Syntiant and for easiness was sampled one second and then was truncated the last 0.032 seconds, to have all samples long 0.968 seconds, which with a frequency of 16kHz, corresponds to 15488 digital elements.

4.1.2 MFE Block Generation and Processing

Previously, was introduced the theoretical and mathematical computation of the Syntiant Block, which is similar to MFE Block, with passages consisting in pre-emphasis, framing, mel-filterbanks and noise floor. The code follows the mathematical logic, however to support Fast Fourier Transformation, to be sure in using a functional and optimized code was used FFTW library, which has already implemented functions for Fast Fourier Transformation[**?**]. To verify the correct computation and application of the theoretical concepts, I relied on Edge Impulse code[**?**]. This is a site on which can be easily perform deployment and provides a tool to convert from input raw data to spectrogram features compatible with Syntiant NDP101. However, the code of Syntiant Block is not fully available, requiring a trial and error implementation. To be sure that the code was corrected, it was performed a viewing validation, because Edge Impulse gives as output a visual spectrogram, so was implemented a python code called via a bash file after program computation that prints out the spectrogram features obtained, adjusting values until the two images matched or were very similar.

[1] According to the inability to perform the complete pipeline on NDP101, has introduced before was used for validation on MCU a STM32, which does not support the feature processing inside its architecture, so was used this software code, but for the original validation it was not necessary



(a) Edge Impulse Spectrogram

(b) Custom Code Spectrogram

Figure 4.2: Comparison of Spectrograms

Algorithm 1: Spectrogram Computation Pipeline

Input: Raw audio signal `audio[]` with `num_samples` samples
Output: Log-Mel spectrogram `log_mel_spectrogram[]`

Normalize and apply pre-emphasis:;
  for $i \leftarrow 0$ **to** $num\_samples - 1$ **do**
    $norm[i] \leftarrow audio[i]/32768.0$;
  $pre\_emphasis\_array[0] \leftarrow norm[0]$;
  for $i \leftarrow 1$ **to** $num\_samples - 1$ **do**
    $pre\_emphasis\_array[i] \leftarrow norm[i] - \texttt{COEFFICIENT} \times norm[i-1]$;
Slice into overlapping frames and apply Hamming window:;
  foreach *frame* f **do**
    $fft\_in \leftarrow$ frame of size `FRAME_SIZE` from $pre\_emphasis\_array$;
    for $i \leftarrow 0$ **to** $FRAME\_SIZE - 1$ **do**
      $fft\_in[i] \leftarrow fft\_in[i] \times (0.54 - 0.46 \times \cos(2\pi i/(FRAME\_SIZE - 1)))$;
  Compute FFT and magnitude spectrum:;
    $fft\_out \leftarrow \texttt{FFT}(fft\_in)$;
    for $b \leftarrow 0$ **to** $NUM\_BINS - 1$ **do**
      $spectrogram[f][b] \leftarrow \sqrt{fft\_out[b].r^2 + fft\_out[b].i^2}$;
Construct Mel filterbank:;
  Generate $FILTER\_NUMBER + 2$ evenly spaced Mel points between MIN_FREQ and MAX_FREQ;
  Convert Mel points to Hz and bin indices;
  foreach *filter* j **do**
    Construct triangular filter between left, center, and right bins;
    Assign weights to $mel\_filterbank[j][\,]$;
Apply Mel filters and compute log-Mel spectrogram:;
  foreach *frame* f **do**
    foreach *filter* j **do**
      $sum \leftarrow \sum_k spectrogram[f][k] \times mel\_filterbank[j][k]$;
      $log\_mel\_spectrogram[f][j] \leftarrow 10 \times \log_{10}(sum + \varepsilon)$;
Apply noise floor and quantization:;
  foreach *value in* $log\_mel\_spectrogram$ **do**
    Normalize with respect to $NOISE\_FLOOR$;
    Quantize to range $[0, 255]$, then normalize to $[0, 1]$;
    Zero out values < 0.65;
return $log\_mel\_spectrogram$

4.1.3 Models Processing

The software pipeline first inferences the KWS model and according to the result, if major than 0, triggers SV one. In simulation, they are handled with the components of neural network explicitly shown. To recall, shall be declared the weights and biases. At the same time, the input and output allocation of each layer should were allocated. Regarding the functions were created the fully connected layer process reasoning and the corresponding activation functions. The Syntiant model elaboration code for both KWS and SV models follow the following algorithm[2], following the theoretical concepts.

Algorithm 2: Neural Network Inference Example

Input: Input vector $x \in \mathbb{R}^{\texttt{INPUT\_SIZE}}$
Output: Predicted class index y
Initialize:
    Create hidden layer buffers: $fc_1 \in \mathbb{R}^{H_1},\ fc_2 \in \mathbb{R}^{H_2},\ fc_3 \in \mathbb{R}^{H_3},\ output \in \mathbb{R}^{C}$;
Feedforward pass:
    $fc_1 \leftarrow \texttt{ReLU}(W_1 x + b_1)$;
    $fc_2 \leftarrow \texttt{ReLU}(W_2 fc_1 + b_2)$;
    $fc_3 \leftarrow \texttt{ReLU}(W_3 fc_2 + b_3)$;
    $output \leftarrow \texttt{ReLU}(W_4 fc_3 + b_4)$;
Softmax normalization (if classification needs):
    $\texttt{softmax}(output)$;
Sort and classify:
    Sort *output* in descending order, track original indices ;
    Print top-C class probabilities and names ;
return *index of class with highest score*

4.1.4 Output Processing

After the model elaborates its output the result should take several directions, for KWS model classification method it consists simply in using the output class to perform the desired action, like triggering the SV model and can be simply done with a if-case in simulation and with SPI sending in inference, but different case is for SV model. To evaluate the result, it is used a cosine similarity approach, consisting in comparing two different reference d-vector finding a percentage of how much they are similar. Two different techniques of using reference features were explored in the previous chapter, but to compute the similarity has to be followed this mathematical formula, consisting in the scalar product of the two vectors long N, divided by the product of their euclidean norms:

$$cosine\_similarity(x, y) = \frac{x \cdot y}{|x| \cdot |y|} = \frac{\sum_{i=0}^{N} x_i y_i}{\sqrt{\sum_{i=0}^{N} x_i^2} \cdot \sqrt{\sum_{i=0}^{N} y_i^2}} \tag{4.1}$$

This will do among reference samples using the techniques of bestmatching and mean-reference, previously introduced. So, after the comparison with the d-vector stored in the dataset according to the system threshold the result will be binary: 0, if it was not found a d-vector similar to the one given in input in the dataset, and 1, if yes.
The structure of the dataset is first cataloged by words, then per user and, finally, per user's sample, if using benchmarking technique. This system is to simplify the overall computation reducing the number of reference d-vector to compare with the input one, simply giving the allocation in memory corresponding to the word selected and all the samples will be in a contiguous memory allocation to use spatial property, accessing only to the subsequent element and it uses an information that it should receive anyway. Another consideration is that if a samples receives a similarity above the threshold similarity the system exists directly and to output 0 it should parse all samples associated to the word given by KWS.

[2]Note that this is a general processing of a model, because in the case shown KWS performs a softmax needed for classification, but it is not mandatory for example SV performs only a ReLU. In the algorithm, is shown the KWS, but has to be adapted to user model needs.

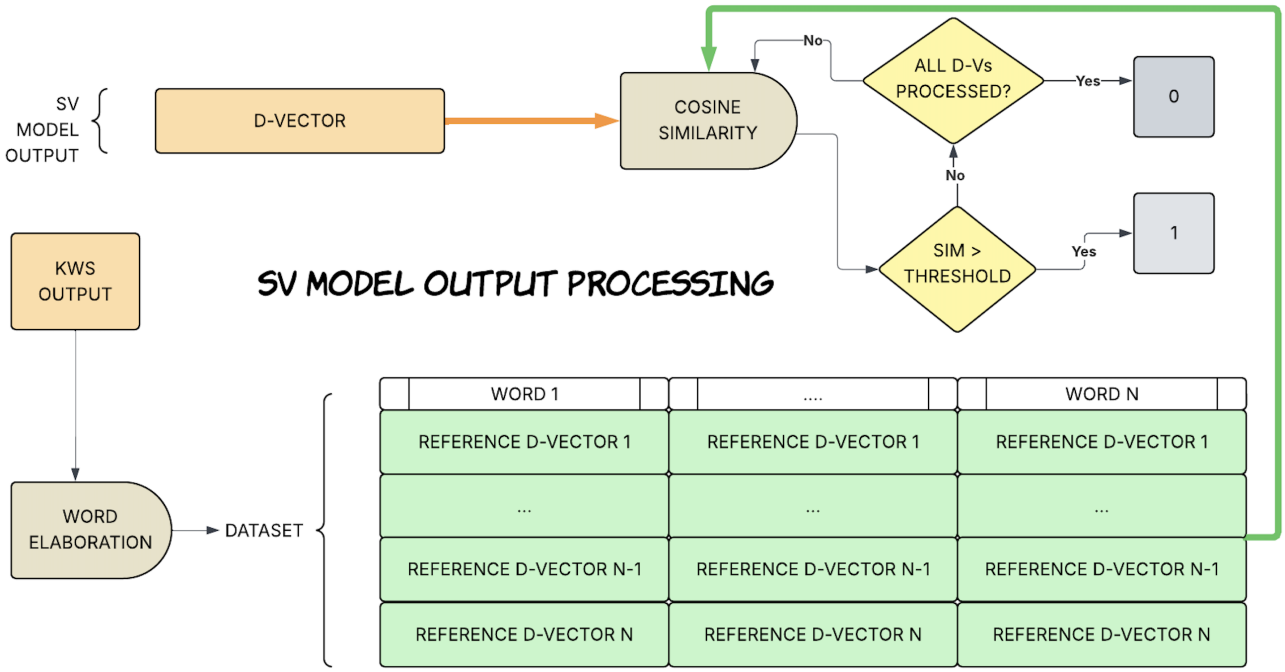


Figure 4.3: D-Vector Processing according to Database

4.1.5 Enrollment phase

On inference, the complete application of this method is restricted due to the impossibility of quantizing the model to 4-bit precision and manipulating SPI interfaces. This limitation is primarily caused by the non-disclosure agreement (NDA) surrounding the Syntiant NDP101, which prevents direct control over its internal functions. The Speaker Verification (SV) model, however, was trained on a large and diverse dataset to generate a distinct d-vector representation for each user, thus enabling a one-time training process. In this scenario, the user needs to enroll by providing a number of voice samples. This number should be balanced: not too large to avoid excessive memory usage, but sufficient to maintain good recognition accuracy. The idea is to reuse the existing pipeline by replacing only the SV model's inference stage. When enrolling, the user specifies the desired keyword, after which the system begins recording. However, it only saves samples that trigger the Keyword Spotting (KWS) model. Once the required number of valid samples is collected, the d-vectors are stored sequentially in memory by determining the current end of the dataset and appending the new d-vectors accordingly. The memory structure for the dataset is designed such that each keyword is allocated a fixed portion of memory. Additionally, a separate structure tracks the number of samples collected per keyword. Since each d-vector has a fixed length, it becomes straightforward to calculate the memory offset for appending new vectors. In simulation, this dataset resides in a header file for simplicity, whereas in the actual inference phase, only the logic has been implemented—full system integration has not been completed due to the limitations in connecting with the KWS system on hardware. Ideally, the dataset would be stored directly in internal SRAM, taking into account its typical 256-byte size constraint. However, some microcontrollers (MCUs) support external SD card storage, which could be used as an alternative. Using a memory mapping technique to manage allocation could be effective even with larger memory sizes and is unlikely to significantly impact energy consumption. Nevertheless, internal storage is generally preferable and should be tailored based on the remaining available memory after code and global variable allocations. A best-matching approach, where multiple samples per user are stored and compared, typically yields higher accuracy but consumes more memory. Alternatively, averaging the d-vectors into a single representative vector per user results in lower precision but significantly reduces memory usage, as it eliminates the need for storing multiple individual vectors.

4.2 Hardware Pipeline

4.2.1 Original System Implementation

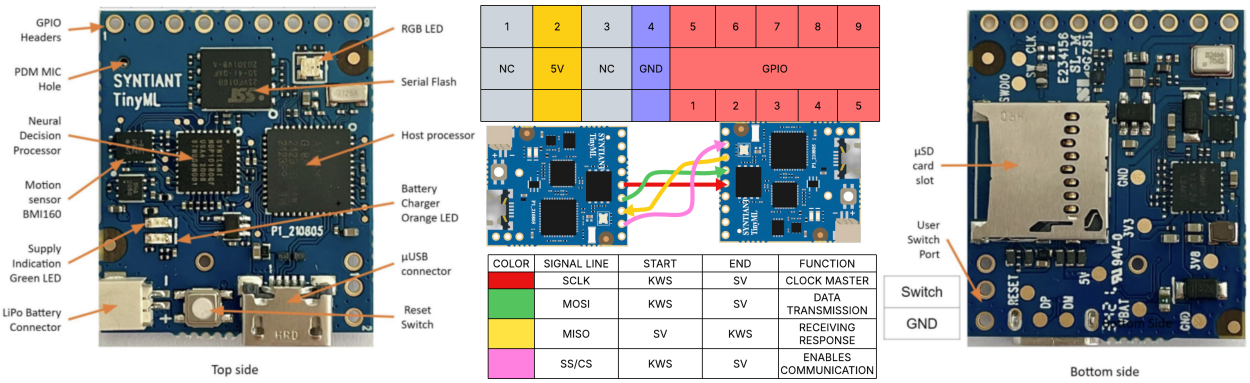


Figure 4.4: SV Model Output Processing

The interaction of the system for • Structure of the system (2 Syntiant NDP101)
-¿ SPI communication NDP101 + Arduino MKRZero
-¿ SPI communication between ESP32 + NDP101

PAGE 22

5 Results Obtained

This chapter shows the results obtained during the exploration of this system with the aim to provide the effectiveness of the models computed and the working logic of the system with an appropriate operation time. To be more precise, there will be explored the performance, the memory occupation and time inference of each of the following models:
• Keyword Spotting model extracted by Edge Impulse • Speaker Verification classification on the huge LibriSpeech dataset[**?**].
• Speaker Verification D-vector extractor in CNN, in DNN (distillated version) and quantized version in int8 and in int4.
• Software Code occupation (code + dataset)
The models were tested using a custom created dataset with personal voice samples and taking the Sheila samples from Google Speech Command not used in training of Keyword Spotting example and other people recorded samples. It is divided for the purpose in 3 classes, my own samples all saying Sheila word (256 samples), other people samples saying Sheila word (2022 samples) and various samples saying different words than Sheila (475 samples). The objective of the KWS is identifying the word Sheila and will be used all the datasets with the first two trying which should be true positive and the last one that should be true negative. Instead, for Speaker Verification, will be enrolled the ones passing KWS, so technically the first two classes and considering that I used my own samples as reference for cosine similarity, the first class should be true positive and the second should be true negative. All the samples are saved in a distinct .wav file and this will lead in using code mode 1 in cycle.

5.1 KWS Performance

Analysis using simulation data (confusion matrix)

5.2 SV Performance

5.2.1 Evaluation of CNN Model

5.2.2 Evaluation and Results of d-vector extractor, distilled and quantized models

Comparison between float32, int8 and int4 (int4 impossible to obtain good results using basic PQT, difficulties in deploying good QAT training, not true support by tensorflow and the fake quantization technique did not work)
• Database design for sample saving
• Cosine similarity EER
• Extend functionality to SV to be compatible with device (Python)
Input (40x40) → Conv+BatchNorm layers → 256-dimensional d-vector
Simulation (model-size, classification purpose, truncation explanation, verification methods (best-matching and mean-cosine)), Real (custom logic for verification)

5.3 System Integration SW & HW

• Overall Power Consumption (ESP32 + 2 Syntiant NDP101)
• Memory Usage

PAGE 24

6 Conclusion & Future Work

1 PAGE

6.1 Technical Limitations

• SDK Barriers (NDA limitations)
• Hardware Constraints (DNN-only architecture limiting SV capabilities)
• Quantization Challenges (Maintaining accuracy at 4-bit precision)

6.2 Design Trade-offs

• Accuracy vs Power (Best-matching vs mean-cosine approaches)
• Security vs User experience (Enrollment samples)
• Model Size vs Complexity (Feature extraction capabilities)

6.3 Key Contributions

• MFE/DNN Code - Full C implementation for Simulation in reproducing the behavior (the model is not quantized, so the behavior is ideal)
• Speaker Verification Distilled
• Quantization - Systematic approach to 8-bit quantization, not 4-bit (too experimental)

6.4 Future Directions

• If able to access to Syntiant, can deploy a compatible binary SV model
• Integration with existing system, that according to my system response will act as consequence
• Adaptive Training (The code is ready, but can't be tested properly)

PAGE 30

Bibliography

[1] Node js repository. deb.nodesource.com. Last Access 06/05/2025.

[2] Post training quantization tensorflow lite. https://www.tensorflow.org/model$_o ptimization/guide/quantiz$
en.

[3] Quantization aware training tensorflow lite. @miscpqt$_t ensorflow, title = Post - Training - QuantizationTensorflowLite, publisher = Tensorflow, howpublished = https : //www.tensorflow.org/model_optimization/guide/quantization/post_training?hl = en, .$

[4] Neural network definition and components. https://s.mriquestions.com/what-is-a-neural-network.html, 2024.

[5] Will McDonald Atul Gupta Mallik Moturi Alireza Yousefi, Luiz Franca-Neto and David Garrett. The intelligence of things enabled by syntiant's tinyml board. Last Access 06/05/2025.

[6] Neuhaus Christian. Code experimental implementation on ndp101. https://github.com/happychriss/Goodwatch_NDP101_SpeeechRecognition/tree/master/include, 2021. Last Access 06/05/2025.

[7] Neuhaus Christian. Ndp101 experimental implementation on ndp101. https://44-2.de/syntiant-ndp-101-always-on-low-power-speech-recognition/, 2022. No other resources that corroborates data listed, Last Access 06/05/2025.

[8] PortAudio Community. Portaudio cross-platform audio i/o library. https://www.portaudio.com.

[9] Espressif. Esp32 documentation. https://www.espressif.com/sites/default/files/documentation/esp32_datashee
2025. Last Access 06/05/2025.

[10] Ostrovan Eugeniu. Tinyml on-device neural network training. https://www.politesi.polimi.it/retrieve/e9f5774c-592d-4741-a6d4-82f5e501630e/TinyML_on_device_neural_network_trainingLast Access 06/05/2025.

[11] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] Jianping Gou, Baosheng Yu, Stephen Maybank, and Dacheng Tao. Knowledge distillation: A survey. 06 2020. Last Access 07/05/2025.

[13] Jahid Hasan. Optimizing large language models through quantization: A comparative analysis of ptq and qat techniques. https://arxiv.org/pdf/2411.06084, 2024.

[14] Kyuyeon Hwang, Minjae Lee, and Wonyong Sung. Online keyword spotting with a character-level recurrent neural network. https://arxiv.org/abs/1512.08903, 2015.

[15] Edge Impulse. Audio syntiant. https://docs.edgeimpulse.com/docs/edge-impulse-studio/processing-blocks/audio-syntiant. Last Access 06/05/2025.

[16] Edge Impulse. Edge impulse syntiant tinyml deploying. https://docs.edgeimpulse.com/docs/edge-ai-hardware/mcu-+-ai-accelerators/syntiant-tinyml-board. Last Access 06/05/2025.

[17] Edge Impulse. Keyword spotting on syntiant stop/go example by edge impulse. Last Access 06/05/2025.

[18] Edge Impulse. Audio classification - keyword spotting. https://studio.edgeimpulse.com/public/499022/latest, 2024. Apache 2.0, Last Access 06/05/2025.

[19] Edge Impulse. Edge impulse block processing repository. https://github.com/edgeimpulse/processing-blocks, 2025. Last Access 06/05/2025.

[20] Edge Impulse. Firmware syntiant tinyml github repository. https://github.com/edgeimpulse/firmware-syntiant-tinyml/tree/master, 2025. Last Access 06/05/2025.

[21] Ying Liu, Yan Song, Ian McLoughlin, Lin Liu, and Li-rong Dai. *An Effective Deep Embedding Learning Method Based on Dense-Residual Networks for Speaker Verification.* 2021. Last Access 07/05/2025.

[22] Keiron O'shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[23] Massimo Pavan, Gioele Mombelli, Francesco Sinacori, and Manuel Roveri. Tinysv d-vector extractor github reference. https://github.com/AI-Tech-Research-Lab/TinySV, 2023. Last Access 07/05/2025.

[24] Massimo Pavan, Gioele Mombelli, Francesco Sinacori, and Manuel Roveri. Tinysv: Speaker verification in tinyml with on-device learning. 2025. Last Access 07/05/2025.

[25] Lutz Roeder. Netron.app tool for model representation. https://netron.app. Last Access 07/05/2025.

[26] Syntiant. General description of ndp101 device. https://www.syntiant.com/ndp101#data_sheet. Last Access 06/05/2025.

[27] Syntiant. hardware_ndp101. https://www.syntiant.com/hardware. Last Access 07/05/2025.

[28] Syntiant. Syntiant tutorial edge impulse. https://www.eetree.cn/wiki/_media/syntiant_tinyml_tutorial_mac_.pdf, 2021. Last Access 06/05/2025.

[29] Rishabh Tak, Dharmesh Agrawal, and Hemant Patil. *Novel Phase Encoded Mel Filterbank Energies for Environmental Sound Classification.* 11 2017.

[30] Kite Thomas. Understanding pdm digital audio. https://users.ece.utexas.edu/ bevans/courses/rtdsp/lectures/10_Data_Conversion/AP_Understanding_PDM_Digital_Audio.pdf, 2012. Last Access 06/05/2025.

[31] TinyML. Tinyml 2021 summit presentation syntiant ndp120. https://cms.tinyml.org/wp-content/uploads/summit2021/tinyMLSummit2021d1_Awards_Syntiant.pdf, 2021. Last Access 06/05/2025.

[32] Daniel Povey Vassil Panayotov, Guoguo Chen and Sanjeev Khudanpur. Open dataset speech with various speakers and many hours speaking. https://www.openslr.org/12, 2015.

[33] Pete Warden. Speech commands: A public dataset for single-word speech recognition. 2017. Last Access 07/05/2025.

[34] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. Understanding int4 quantization for transformer models: Latency speedup, composability, and failure cases. https://arxiv.org/abs/2301.12017, 2023. Last Access 07/05/2025.

[35] Feng Ye and Jun Yang. A deep neural network model for speaker identification. *Applied Sciences*, 11(8), 2021. Last Access 06/05/2025.