UCLouvain

École polytechnique de Louvain

Towards an embedded offline voice assistant using state-of-the-art hardware implementation of artificial intelligence

Author: **Jimmy Weber**
Supervisor: **Flavio Abreu Araujo & Luc Piraux**
Readers: **Luc Piraux, Flavio Abreu Araujo, Nicolas Marchal, Christophe De Vleeschouwer, Charlotte Frenkel**
Academic year 2020–2021
Master [120] in Electrical Engineering

Contents

Abstract

Deep learning is very popular because of its exceptional performance in classification[1]. However, it is a highly complex model and its internal mechanisms are not always well understood. In addition, the parameters adjustment is often obscure [2]. Finally, its training is long and the solutions found are not global optima [3, 4].

However, there is a neural network architecture that can potentially overcome some problems of deep learning: Reservoir computing [5, 6]. This model allows a fast training and using simple concepts of linear algebra. The solution it finds is global and deterministic [7, 8]. Also, it allows hardware integration, and can therefore be much more energy efficient [9].

The aim of this thesis is to integrate an hardware neuron in the reservoir computing architecture. The device chosen to act as a neuron is a spin-torque nano-oscillator (STNO), a nano-object at the heart of spintronics research. This device offers non-linear and causal dynamics, properties conducive to neuromorphic simulation. In addition, it shares the same structure as current magnetic memory cells and is therefore compatible with CMOS technology [10].

In order to demonstrate the lightness of such an architecture, it has been implemented on an embedded system. To show that the interaction between the software model and the hardware neuron is possible, an ASIC prototype simulating the behaviour of the STNO was implemented.

The software was developed on a Raspberry Pi 4 [11] while the ASIC was prototyped on a DE0-nano SoC including a Cyclone IV FPGA [12].

To prove the effectiveness of the system constructed, a speech recognition task was challenged. It was a success.

Key words : Machine learning, Deep learning, Reservoir computing, Spintronic, Spin-torque nano-oscillator (STNO), Neuromorphic computing, Embedded system, System on Chip (SoC), Hardware accelerator, Application-specific integrated circuit (ASIC), Field-programmable gate array (FPGA), Speech recognition.

Acknowledgements

First of all, I would like to thank the people who agreed to read my manuscript. Thanks to Christophe De Vleeschouwer, Charlotte Frenkel and Nicolas Marchal. Also, I would like to sincerely thank my two supervisors, Luc Piraux and Flavio Abreu Araujo.

Flavio trusted me and put all his efforts into my well-being at work. He gave the research group an atmosphere that favoured productivity, sharing and collaboration. I was very impressed by his social and intellectual ambivalence. Indeed, technically speaking, he was a great advisor and understood the challenges with a high degree of professionalism. Despite my lack of qualification in spintronics, he was able to explain the principles I needed to carry out my work. Finally, I would like to thank him for his availability and our long off-topic discussions, which enriched me a lot.

Also, I wanted to thank all the team members. Although they didn't help me technically, they were the main actors of my good mood coming to the lab. Nicolas and Tristan, with whom I enjoyed playing football and spending informal moments after work. Simon, a work partner who seemed to be very talented in incomprehensible subjects for myself. And finally Chloe, who joined us in February but who seems to have been there from the beginning. I also wanted to thank her for her careful review of my thesis and her frequent encouragement.

Finally, I wanted to thank all my relatives.
All of my roommates, who I had to turn down several times for activities but who were very understanding.
I also wanted to thank my best friend, Tom, who took an interest and asked me simple questions but sharpened my understanding of the problem when I explained.
I would also like to thank all my family. Sometimes I had to be away for several weeks to complete my research in time. I am very grateful to them for their understanding and their permanent support.
To conclude, I would like to thank my girlfriend Aurore, without whom this work would not have been possible at all. For all the sacrifices, the attentions, the questions, the support,... Thank you.

Introduction

The purpose of this thesis is to integrate a hardware neuron into a neural network architecture. To demonstrate the ability of the implemented model, a voice recognition task was challenged. The goal here being to make spoken digit recognition. Several stages were necessary to succeed in this.

First of all, it was essential to acquire the voice waveform and to isolate the voice segments. For this, several techniques of voice detection and signal cleaning were considered.

Next, it was a matter of classifying which digit the waveform referred to. For this purpose, a machine learning model was used. In order to compare the final model with a well-known one, an implementation of a state-of-the-art model was conducted.

Afterwards, the model for hardware integration was designed. Due to its many advantages, the reservoir computing architecture was chosen. Indeed, this model allows the use of a single physical neuron to act as a whole neural network. Also, its training is very efficient and deterministic allowing an optimal solution to the neural weights.

A complete system integrating a spin-torque nano-oscillator as neuron was first simulated in software. Indeed, it was preferable to know if the system was computationally viable using such a neuromorphic device.

Then, to demonstrate the feasibility of a hardware-software combination, a hardware accelerator simulating the neuromorphic device was developed. As expected, it delivered the same results as its software equivalent.

Finally, in order to situate the new model in comparison with conventional methods, a benchmarking study is carried out. The results are very promising.

The structure of this report is consistent with the chronological development of the system. To help the reader to understand the interactions and the signal flow, the Fig. 1 illustrates the global work-flow and will be included at the beginning of each chapter.

Chapter 1 is focused on the acquisition and preprocessing of the input data. It covers the different ways in which the inputs are cleaned. It also details how voice segments are isolated from the overall speech waveform.

Chapter 2 covers the voice recognition of commands. Different well-known machine learning techniques are described to perform speech recognition. This chapter explains how the different models work and one state-of-the-art model is implemented in order to benchmark the final one.

Chapter 3 explains a model for integrating a hardware device. The model is called Reservoir computing and the integrated device chosen is an spin-torque nano-oscillator. This chapter describes these two and proposes a simulated implementation of them.

Chapter 4 details how the model has been implemented in an embedded way. It demonstrates the the feasibility of communication between the software model and the hardware neuromorphic device simulation.

Chapter 5 presents the results of the final system compared to the state of the art model presented in chapter 2. It also explains the limiting factors and possible area for improvements.

Figure 1 represents the overall flow of the signal. It will be the main thread of this work and will help the reader to visually understand the different parts.

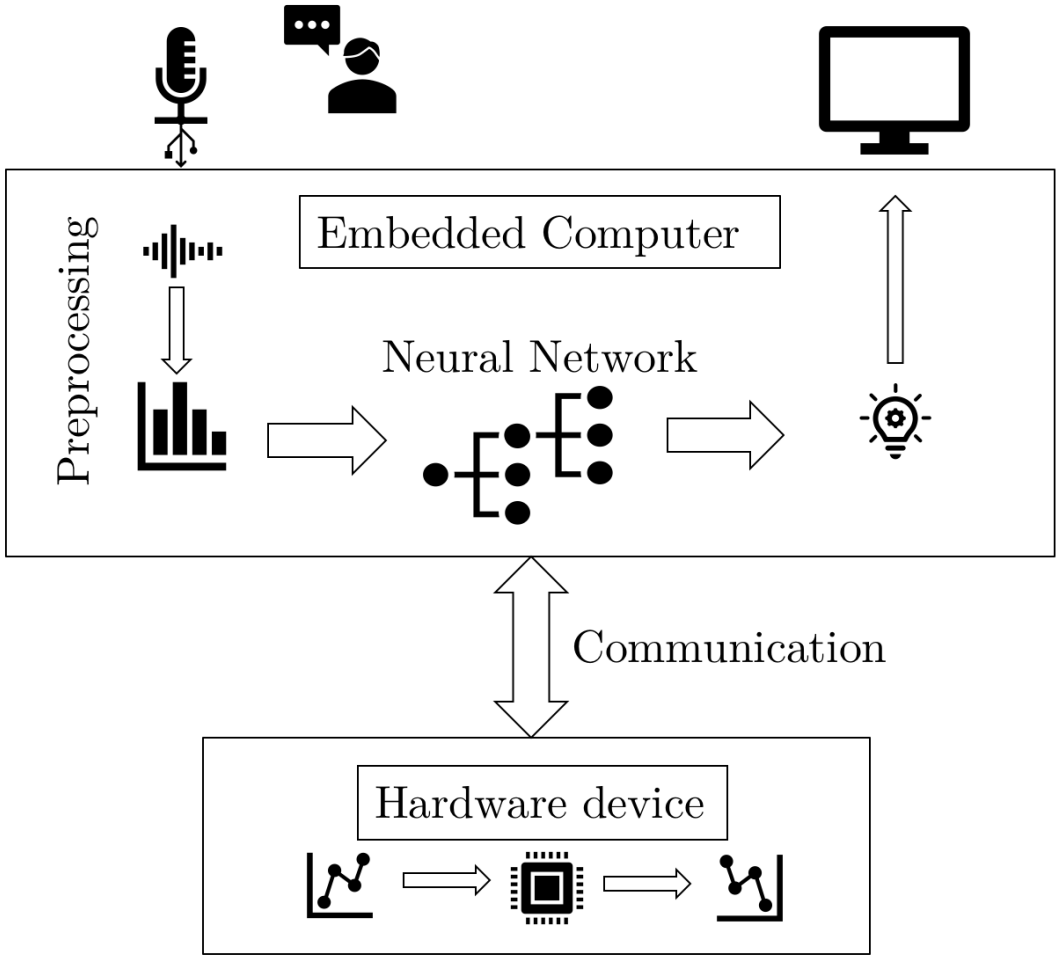

Figure 1: Global data and workflow of the project.

In practice

Figure 2 shows the assembly on which the complete system is mounted :

- The software including preprocessing and machine learning (i.e. Reservoir computing) is implemented in Python on a Raspberry Pi 4. [11].
- For the hardware part, it was prototyped on a DE0-Nano including a Cyclone IV FPGA [12] using ModelSim for simulation and Quartus Prime programming of the FPGA.
- The two elements are communicating through a bridge and using Serial Peripheral Interface (SPI).

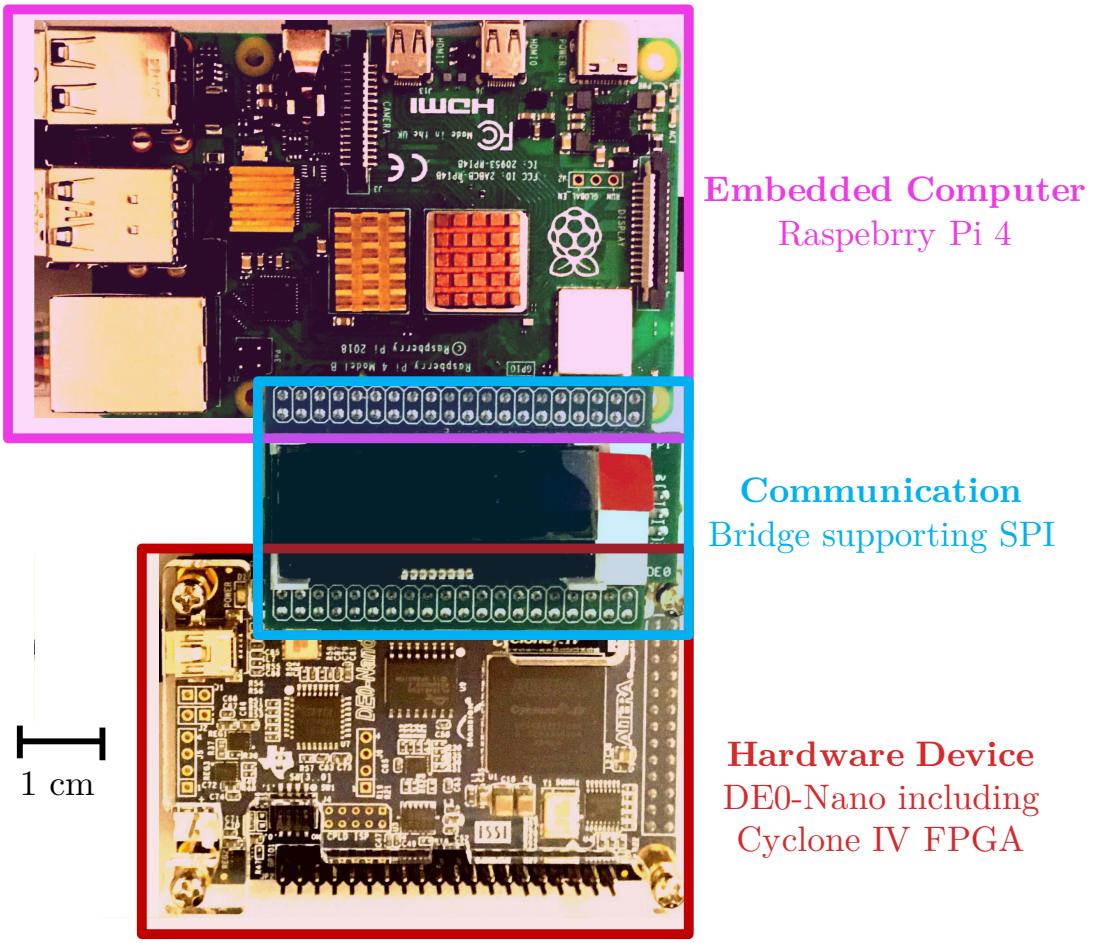

Figure 2: Picture of the real prototype.

Reader forewords

To avoid untimely references to a chapter of the state of the art, the report includes in each chapter a description of the tools used to build this block, the section "**In Practice**". It is thus explicitly indicated when it is self-made work.

All the code resources can be accessed from
`https://gitlab.flavio.be/jimmy/masterthesiscode`

Note that for readability reason, a matrix is denoted bold letter, e.g. $\mathbf{X}$ while its elements are denoted in lowercase letter , e.g. x and such that :

$$\mathbf{X} = \{x[i] \text{ for } i \in [0, |\mathbf{X}|]\} \tag{1}$$

Chapter 1

Preprocessing

This chapter covers preprocessing and is divided into two parts. The first part concerns the acquisition of the voice waveform and extracting its relevant frequency and speech data. The other section consists in its cleaning and denoising.

Figure 1.1 emphasises where this part is located in the work-flow.

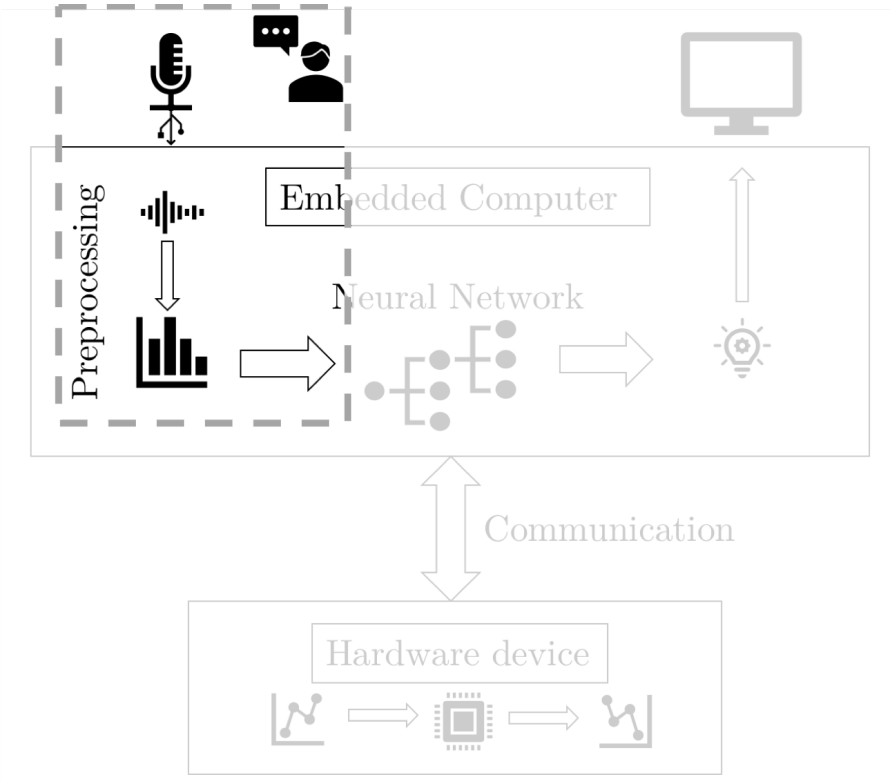

Figure 1.1: Preprocessing in the work-flow.

1.1 Acquisition and pre-treatment

When the sound waveform is acquired with the microphone, it has to be transformed into a machine-readable file. Fortunately, several libraries already exist to perform this transformation. However, during the acquisition process, various types of noise are added to the signal. These are undesirable and must be removed. Also, to avoid processing a sound without voice, it is necessary to add a voice activity detector (VAD). This section deals with these different topics.

1.1.1 In practice : Data acquisition using a microphone

First, a USB microphone is connected to our module. This microphone samples the vocal signal at 44.1 kHz. Indeed, the Shannon-Nyquist [13] law states that the frequencies to be recorded must be at most half the sampling frequency. With this microphone it is possible to record sound waveform containing frequencies from 0 to 22.05 kHz. This range corresponds approximately to the spectrum of the human ear, which is essential for a model attempting to mimic human behavior.

The PyAudio python library [14] is used to record the acoustic stream and store it in a `.wav` [15] file.

1.1.2 Voice Activity Detector - VAD

Then, it is necessary to isolate the periods when the voice is present in the sound signal. To do so, two tools are useful: a bandpass filter and a power activation function.

Bandpass filter

To isolate the voice in the signal, it is first required to filter the signal with only the frequencies specific to the human voice. The fundamental frequency of the voice varies between 50 and 300 Hz. However, since a lot of information is also found in the harmonics, it is decided to use the same range as in wideband telephony, i.e. 50 to 7000 Hz [16]. For this application, a Butterworth IIR filter is used because it has a flat maximum gain [17]. Indeed, it is important to keep a maximum of information at this stage while not introducing undesirable ripples. Figure 1.2 shows its dynamics on which it can clearly be seen that the gain is maximally flat.

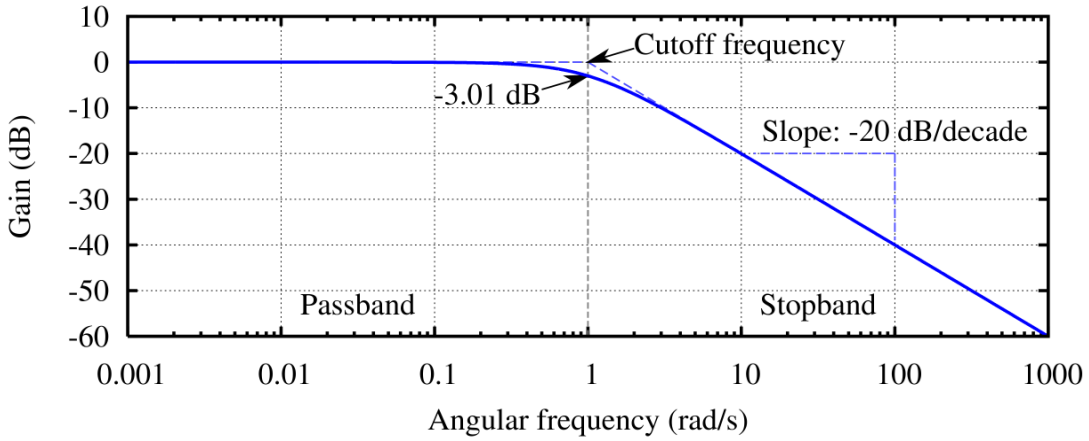

Figure 1.2: Bandpass Butterworth IIR filters, from [18].

Power activation function

Now that the signal contains only relevant frequencies, it is important to discriminate human speech from silence, i.e. when the signal has the maximum power. To do so, the signal $\mathbf{x}$ is divided into several parts of the same duration N called time windows. The power of each of these windows is then computed as :

$$\text{Power} = \frac{\sum_{i=0}^{N} x[i]^2}{N} \tag{1.1}$$

If the power exceeds a certain threshold, then the window may be considered as a speech section of the signal.

In practice : Python `TorchAudio.VAD`

Initially, a home-made implementation of these two methods has been implemented in python. However, after further investigation, it turns out that a python sub-library from Pytorch [19] allows this filtering and classification. The implemented function used in the project is called `torchaudio.VAD`.

1.2 Wavelet denoising

Then, to complete the signal cleaning, it is necessary to remove the noise. The noise is typically assumed to be white (uniform power across the frequency band) and following a random Gaussian distribution (AWGN : Additive White Gaussian Noise). To remove it, the wavelet denoising method[20] is used and the underline theory is introduced hereafter.

1.2.1 Wavelet basics

The wavelet transform allows a decomposition of a signal according to several levels of details. For the sake of clarity, the Haar system is considered which represents an easy way to discretize the signal [21].

Given the function $f(t)$ and defining its discretization of step Δt (also written Δ_t) as the projection of this function on the space $\mathcal{V}(\Delta_t)$. Its discretization/approximation is therefore given by $a_{\Delta_t}(t) = \mathcal{P}_{\mathcal{V}(\Delta_t)} f(t)$. Figure 1.3 shows $f(t)$ and its projection.

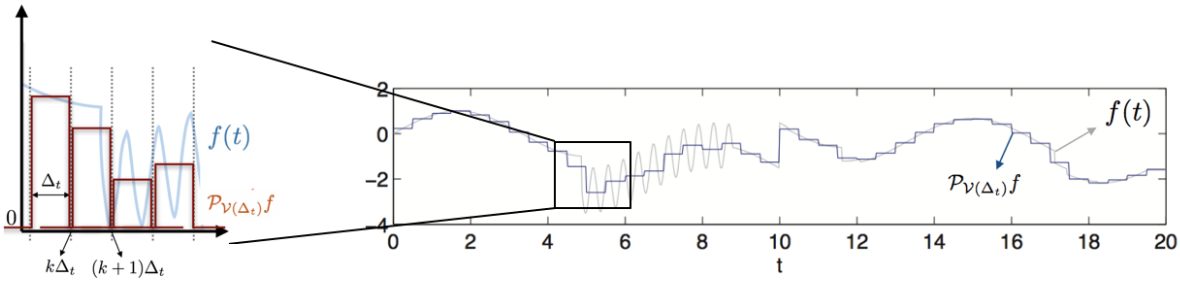

Figure 1.3: Function $f(t)$ and its projection on $\mathcal{V}(\Delta_t)$, adapted from [22].

$\mathcal{V}(\Delta_t)$ is therefore defined as a set of piecewise approximation functions $\phi_{\Delta_t,k}(t)$. This function $\phi_{\Delta_t,k}(t)$ is actually a scaling (by Δ_t) and a translation (by k) of the rectangle function $\phi(t) = \phi_{1,0}(t)$, as shown in Fig. 1.4 and defined in Eq.1.2:

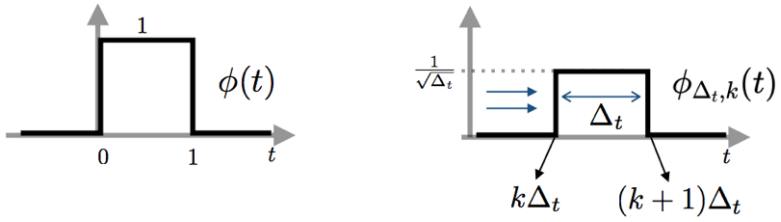

Figure 1.4: Definition of $\phi(t)$ and $\phi_{\Delta_t,k}(t)$, adapted from [22].

$$\phi_{\Delta_t,k}(t) = \frac{1}{\sqrt{\Delta_t}} \phi\left(\frac{t}{\Delta_t} - k\right) \tag{1.2}$$

where $\phi(t)$ is a rectangle function and where the term $\frac{1}{\sqrt{\Delta_t}}$ ensures normalization such as $\|\phi_{\Delta_t,k}\| = \|\phi\| = 1$

For easier notation, the resolution j is defined such that $\Delta_t = 2^{-j}$. Thus, an increment of the resolution leads to divide Δ_t by two, the signal is therefore twice more precise.

Finally, $\mathcal{V}(\Delta_t)$ can be generated from its orthogonal basis given in Eq.1.3

$$\begin{aligned} \text{ONB for } \mathcal{V}(\Delta_t) &= \left\{ \phi_{\Delta_t,k} : \phi_{\Delta_t,k}(t) = \frac{1}{\sqrt{\Delta_t}} \phi\left(\frac{t}{\Delta_t} - k\right), k \in \mathbb{N} \right\} \\ &= \left\{ \phi_{j,k} : \phi_{j,k}(t) = 2^{\frac{j}{\Delta_t}} \phi\left(2^j t - k\right), k \in \mathbb{N} \right\} \end{aligned} \tag{1.3}$$

$$\text{where : } \langle \phi_k, \phi_{k'} \rangle = \delta_{kk'} := \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases}$$

It is then interesting to compute the difference for two $f(t)$ at two consecutive resolutions. This difference is called the details. Figure 1.5 shows the details obtained after the difference between $a_j(t) = \mathcal{P}_{\mathcal{V}_j} f(t)$ and $a_{j+1}(t) = \mathcal{P}_{\mathcal{V}_{j+1}} f(t)$.

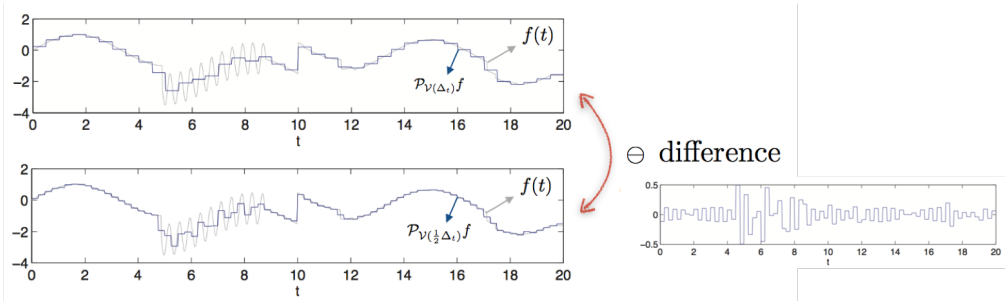

Figure 1.5: Detail obtained from the difference between $\mathcal{P}_{\mathcal{V}_j} f(t)$ and $\mathcal{P}_{\mathcal{V}_{j+1}} f(t)$, adapted from [22].

As for the approximation, the details can be derived from the projection of $f(t)$ onto a certain space $\mathcal{W}_j$ such that :

$$d_j(t) = a_j(t) - a_{j+1}(t) = \mathcal{P}_{\mathcal{V}_j} f(t) - \mathcal{P}_{\mathcal{V}_{j+1}} f(t) = \mathcal{P}_{\mathcal{W}_j} f(t) \tag{1.4}$$

The space $\mathcal{W}$ can then be derived from the space $\mathcal{V}$ as the subtraction of their elements for two different scales.

$$\begin{aligned} \text{ONB for } \mathcal{W}_j &= \{ \psi_{j,k} : \psi_{j,k}(t) = \phi_{j,k} - \phi_{j+1,k}, k \in \mathbb{N} \} \\ &= \left\{ \psi_{j,k} : \psi_{j,k}(t) = 2^{\frac{j}{2}} \psi\left(2^{-j} t - k\right), k \in \mathbb{N} \right\} \end{aligned} \tag{1.5}$$

where : $\psi = \psi_{j=0,k=0} = \phi_{j=0,k=0} - \phi_{j=1,k=0}$ as described in Fig. 1.6

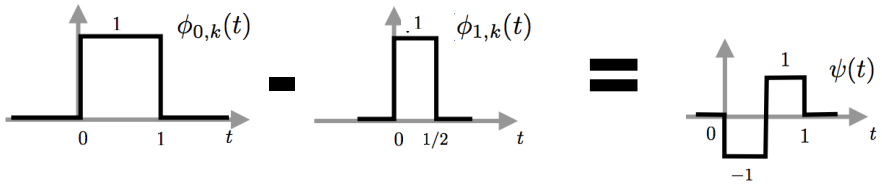

Figure 1.6: Definition of $\psi(t)$, adapted from [22].

The function $\psi(t)$ is in fact called a wavelet. In Haar's system, $\phi(t)$ is a rectangular function that discretizes the signal. However, it can be any function depending on the system. This implies several types of wavelet functions with different properties.

From there, it is possible to build a multi-resolution analysis of a signal, by taking an approximation of the signal and adding a number of details to recover the original.
Indeed, from Eq.1.4, one can derive $a_{j+1}(t) = a_j(t) + d_{j+1}(t)$. By extension :

$$f(t) \approx a_N(t) = a_{j_0}(t) + \sum_{j=j_0}^{N} d_j(t) \tag{1.6}$$

where $a_0(t)$ is the mean of the signal.
The matrix formed by the vector of details is called the details matrix $\mathbf{D} = \begin{pmatrix} \mathbf{d}_{j_0} \\ \mathbf{d}_1 \\ \dots \\ \mathbf{d}_N \end{pmatrix}$
Figure 1.7 shows the reconstruction of the function $f(t)$ with a resolution of $N = 10$ starting from an approximation of level $j_0 = 5$.

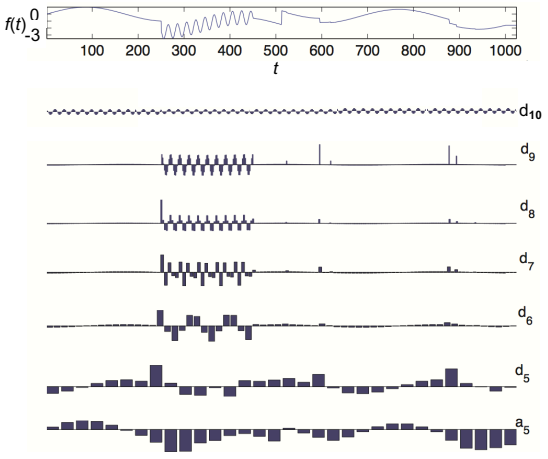

Figure 1.7: Multiresolution analysis of $f(t)$, adapted from [22].
An artificial noise has been added for the sake of clarity.

1.2.2 Denoising

As it can be observed on the Fig. 1.7, the details matrix is quite sparse. However, an additive white Gaussian noise (AWGN) ε implies non-sparse details at some level of decomposition. This is noticeable on the Fig. 1.7 for the d_{10} detail. Indeed, the signal can be decomposed as follows:

$$f(t) = f_{\text{pure}}(t) + \varepsilon, \quad \varepsilon(t) \sim \mathcal{N}\left(0, \sigma^2\right) \tag{1.7}$$

$$\mathcal{P}_{\mathcal{W}_j} f(t) = \mathcal{P}_{\mathcal{W}_j} f_{\text{pure}}(t) + \mathcal{P}_{\mathcal{W}_j} \varepsilon(t) \tag{1.8}$$

It can be established that the projection of an f_{pure} is sparse. However, as the noise ε is random, its projection is therefore not sparse [20].

The idea of wavelet denoising is to identify the details implying non-sparse (full) details matrix such that they can be avoided. This is performed by setting a threshold T below which some of the $\mathbf{d}_j$ coefficients are cancelled. There are several thresholding methods with different properties. The two most common are hard and soft thresholding [20].

Hard thresholding

$$\mathcal{T}_h(d_j(t), T) = \begin{cases} d_j(t) & \text{if } |d_j(t)| > T_j \\ 0 & \text{else.} \end{cases}$$

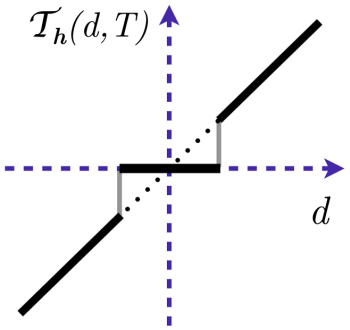

Figure 1.8: hard thresholding, adapted from [22].

Soft thresholding

$$\mathcal{T}_s(d_j(t), T) = \begin{cases} d_j(t) - T_j & \text{if } d_j(t) > T_j \\ d_j(t) + T_j & \text{if } d_j(t) < -T_j \\ 0 & \text{else} \end{cases}$$

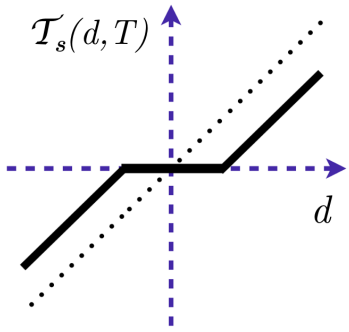

Figure 1.9: Soft thresholding, adapted from [22].

Then, it is necessary to set the threshold value T. Many methods exist such as `sqtwolog, rigrsure, heursure, minimaxi`. The common metric to evaluate the performance of a denoising technique is the root mean square error (RMSE).

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=0}^{N}\left(f_{\text{pure}}[i] - f[i]\right)^2} \tag{1.9}$$

According to this study [23], the best technique for this metric is `rigrsure` combined with soft thresholding. The `rigrsure`, defined in Eq.1.10, minimizes the SURE (Stein's unbiased risk estimate) estimator which is an unbiased estimator of the mean square error [24]. It is defined as follows:

$$T_{\texttt{rigrsure},j} = \sigma_j\sqrt{\omega_b} \tag{1.10}$$

where ω_b is the b^{th} coefficient of $\mathbf{\Omega}$, a matrix of the squared wavelet coefficients (minimum risk coefficient) sorted in ascending order. σ is the standard deviation of the noisy signal.

Figure 1.10 shows the improvements in RMSE using soft `rigrsure` thresholding at a level of $j = 5$.

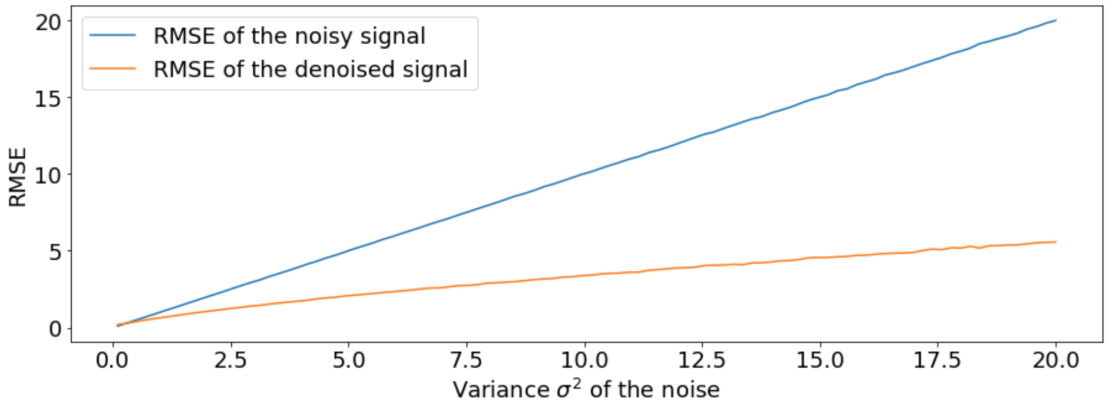

Figure 1.10: Comparison of the RMSE between a noisy signal and its denoising version using soft `rigrsure` thresholding at a level of $j = 5$.

1.2.3 In practice : Python `PyYawt.wden`

The library used in this project is the `PyYawt` library that offers many different tools for wavelet denoising [25]. This includes the `wden` function that denoises the signal. It first evaluates the threshold with the `rigsure` method, and then denoises it.

Conclusion

This concludes the preprocessing part.

First it was seen how data was acquired with the microphone. Then it was a question of isolating the frequencies and the periods proper to the speech. Finally, it was necessary to denoise the signal with the help of wavelet theory.

The result of this part, is a waveform ready to be analysed by the classification models which will be the subject of the following chapter.

Chapter 2

Machine learning

Machine learning is at the heart of the system. It allows to associate a sound with the word it represents. This chapter describes the main concepts of machine learning. It discusses the well-known architecture and algorithms for voice processing. It covers several elements ensuring a good classification:

First, an annotated dataset is needed for training, i.e. a set of elements and their correspondence. In this case, sounds associated with the word spoken. To ensure efficient and optimal learning, feature extraction and cross validation methods are used respectively.

Then, a model is needed. In this analysis, two machine learning techniques are described: the feed-forward neural network (FFNN) and the convolutional neural network (CNN).

To conclude this chapter, a state of the art model is implemented using CNN. This will be further used as a comparison point with the final model using reservoir computing.

Figure 2.1 presents the machine learning in the global work-flow.

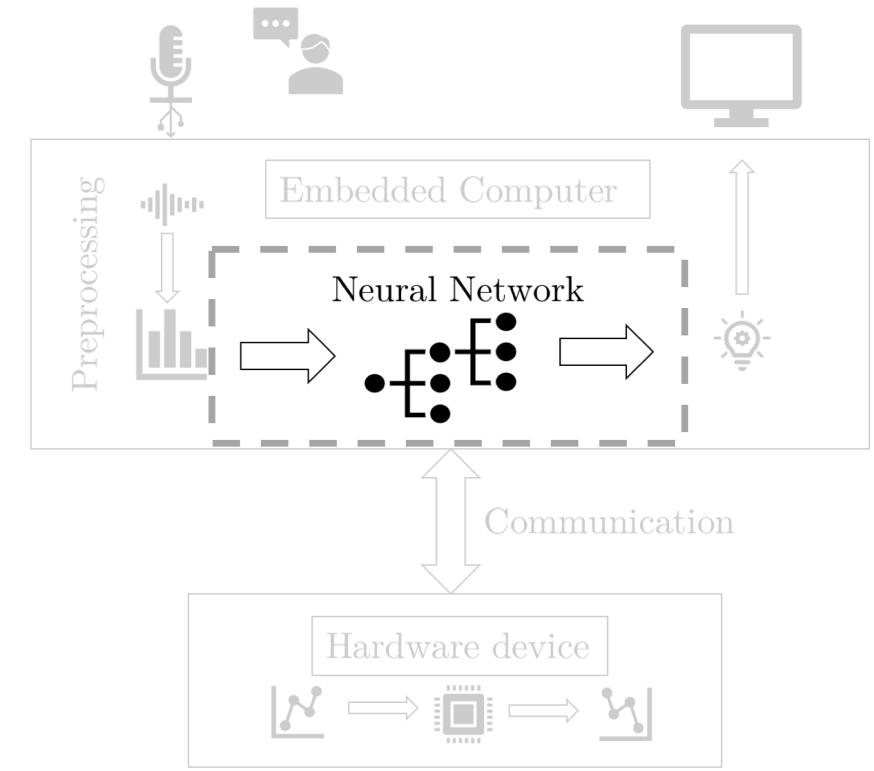

Figure 2.1: Machine learning in the global work-flow.

2.1 The Dataset

Before the model can classify sounds, it must first practice. For this purpose, it is provided with an annotated dataset.
However, for efficiency reasons, the dataset must be as light as possible. It is therefore essential to extract as much information as possible. For this, feature extraction is performed.
Also, to ensure optimal training and generalisation, the dataset must be used methodically. Thus, the K-fold cross-validation technique is applied to the set.

In practice

But first here is the two sets used for this work :

- TI-46 [26] : This licensed set contains 500 samples. Five women clearly pronounce the ten digits (from zero to nine) 10 times. It is selected because it is usually used in hardware reservoir computing [27, 28].
- SPEECHCOMMANDS [29] : This free set is a standard for Tensorflow (both developed by Google Brain). It is also one of the standard TorchAudio sets (from Pytorch [19]). It contains 38908 samples for all digits.

2.1.1 Features extraction

Feature extraction aims to reduce the data size by maximizing information and avoiding redundancy. Two standard methods are considered for feature extraction: MFCC [30] and Cochlear[31, 32]. Both methods are biological models that attempt to mimic the perception of the human ear. Indeed, one can assume that the ear has adapted to voice recognition. Here is their working.

1. The waveform m is first splitted into windows of τ_{window}. These windows can overlap by τ_{overlap}. For MFCC and cochlear, τ_{window} is respectively equal to 25 ms and 50 ms. The τ_{overlap} is set to 10 ms for both technique.

2. Then a feature extraction technique is applied to it. From each window is extracted respectively 13 and 78 scalars for MFCC and cochlear technique.

3. Everything is finally gathered in the feature set $\mathbf{X}_m \in \mathbb{R}^{N_\tau \times N_f}$:

   - N_f is the number of extracted features.
   - N_τ corresponds to the number of windows of size τ_{window} that the signal contains. If the signal is 1 second long then $N_\tau = 1000/25 = 40$.

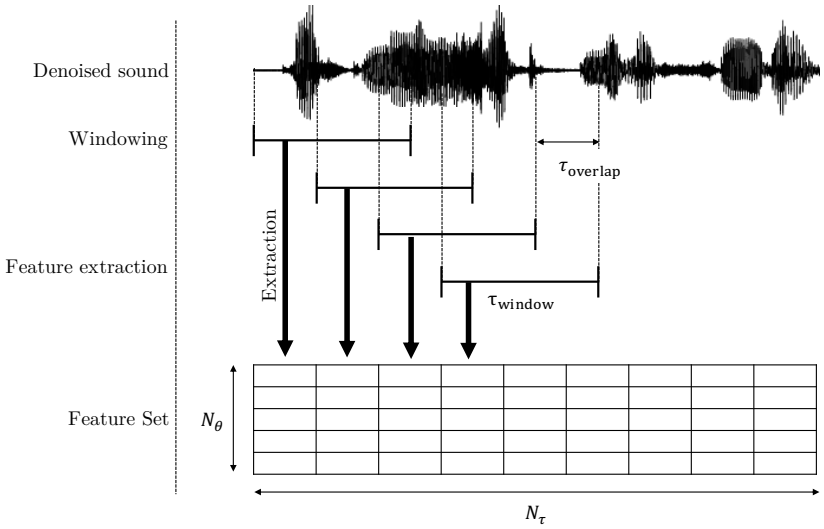

Figure 2.2: Voice feature extraction general description.

In practice : Python Library's

- For the MFCC, the torchaudio.MFCC librairy is used [19].
- For the cochlear transformation, the `Lyon-Cochlea` library is used [33]. It is a python library using the original C code of M. Slaney. [34].

2.1.2 K-fold Cross-Validation

During the learning process, it is important to avoid overfitting the model to the data presented. Indeed, its first goal is to be able to classify new data, to generalize. To avoid overfitting, the set is divided into two subsets:

- The learning set: This set is used to train and improve the model. In spite of this, to evaluate the performance of the model with new data, the K-fold cross-validation is used [35]. This technique consists in a successive subdivision of the learning set into sub-folders on which the model will either be trained or evaluated : The training and the validation set. Thus, it is possible to prevent overfitting by checking the behavior of the model on the pseudo-new set that is presented to it.
- The test set: This set allows to check the quality of the classification model. This fresh set is compared to the model and its performance is checked. Contrary to the validation set, this set cannot be involved in the training, otherwise the quality would lose objectivity.

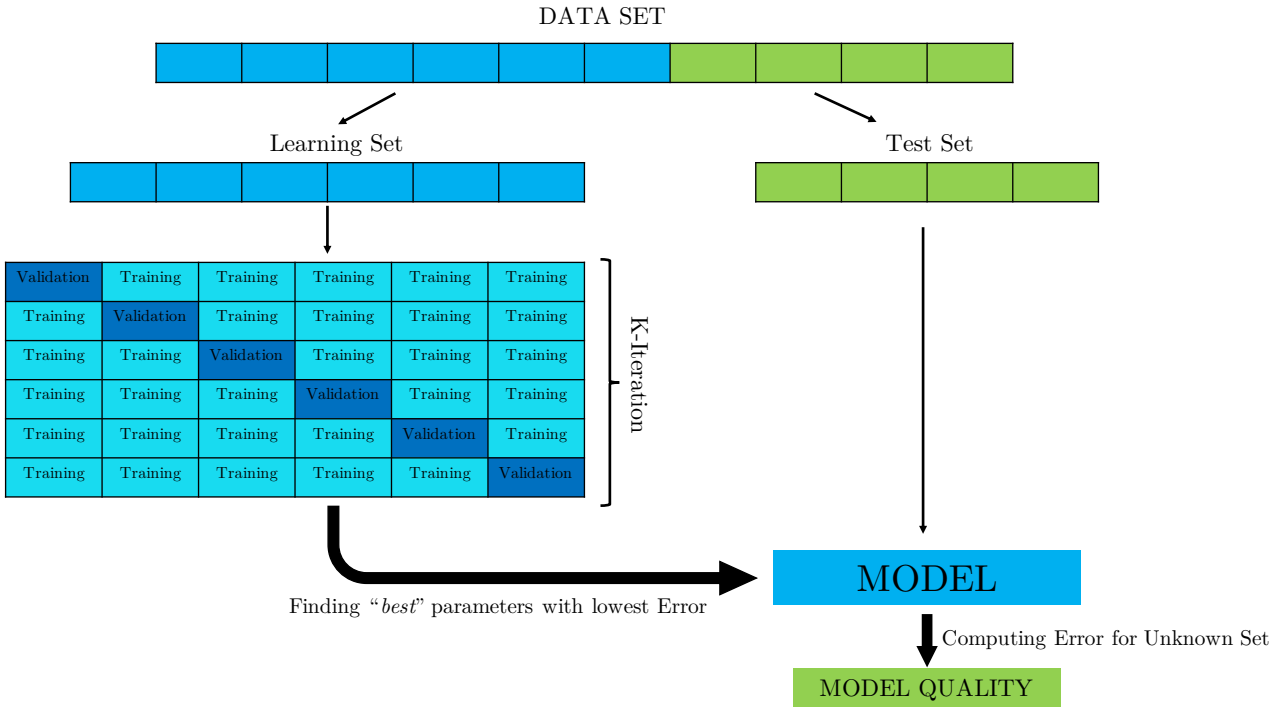

Figure 2.3: K-Fold dataflow to prevent from overfitting.

In practice

For both sets used, the learning set represents 10% of the dataset. The validation set is also a portion of 10% of the learning set.

2.2 Classification generalities

Example

To illustrate the point of a classification, here is an example of a classification of animals, based on their characteristics. For the example, one must classify if some features describe a cat or not. For this, the variable t_m is introduced, it describes the nature t of the animal m and such that :

$$t_m = \begin{cases} 1 & \text{if animal}_m \text{ is a cat} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The TABLE 2.1 gives an example of a possible survey. It can be seen that M animals have been studied according to N characteristics such as their size, their weight,...

Index m	$\mathbf{x_{n=1}}$ **height [cm]**	$\mathbf{x_2}$ **length [cm]**	$\mathbf{x_3}$ **weigth [cm]**	$\mathbf{x_4}$ **longevity [year]**	$\mathbf{x_5}$ **gestation [day]**	**...**	$\mathbf{x_N}$	**Target** t
m = 0	20	35	4,2	8	61	...		*Cat (t=1)*
1	25	30	4,3	7	26	...		*Duck (t=0)*
2	26	40	5,6	10	60	...		*Cat (t=1)*
...	...	...	...	...	...	...	...	...
m = M	10	70	6,7	14	101	...		*Snake (t=0)*

Table 2.1: Example : sample of animals surveyed, the $\mathbf{X}$ matrix is framed.

Definitions

More generally, let there be a set of experiments $\mathbf{X} \in \mathbb{R}^{N \times M}$ describing M items according to N features. Note that in the case of speech recognition, $N \to N_\tau \times N_f$. The goal of the classification is to build a function $f(\mathbf{X}_m)$ allowing, on the basis of $\mathbf{X}_m$ features, to establish the y_m predictions. The prediction concerns the nature of the m^{th} element and tries to get as close as possible to its true nature, the target t_m.

$$y_m = f(\mathbf{X}_m) \approx t_m \tag{2.2}$$

During training, the targets are available and the function refines its predictions y_m so that they tend to t_m.

During inference, the targets are no longer accessible and the model makes predictions on elements that are unknown to it.

2.3 The feed-forward neural network

The following section describes the feedforward network. First, the simple perceptron is presented. Then, the multi-layer perceptron is discussed.

2.3.1 Perceptron

Definition

The perceptron is essentially a classifier [36]. As a reminder, a classifier is a function f making, on the basis of N features $\mathbf{X}$, predictions y about the nature of elements m. The predictions y having as target the true nature t.

$$y_m = f(\mathbf{X}_m) \approx t_m = \begin{cases} 1 & \text{if animal}_m \text{ is a cat} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Moreover, the perceptron is a linear classifier. This means that it makes a decision based on a linear combination of the features $\mathbf{X}_m$:

$$y_m = f(\mathbf{X}_m) = \varphi(\mathbf{W}\mathbf{X}_m) \tag{2.4}$$

- $\mathbf{X}$ is a set of N features for the M elements.
- $\mathbf{W}$ is the matrix of weight. They act as synapse in neuronal system. They are the ones that get trained.
- In the perceptron a weighted sum of the inputs is operated (that is covered by the matrix product in the Eq.2.4). Note that $x_{n=0} = 1$ so that w_0 can act as a bias in the weighted sum.
- φ is the activation function, the one that allows the decision to be made. It is usually non-linear.

Figure 2.4 is the graphical representation of the perceptron (also called neuron).

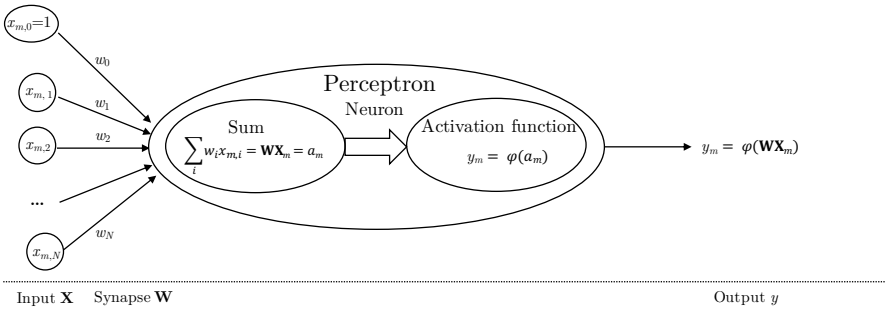

Figure 2.4: Graphic representation of the perceptron.

Training

Recall Eq.2.3, the objective is to train the classifier so that the predictions y_m tend to the targets t_m for all m. To this end, the error function E is defined.
The activation function being fixed, the adjustable parameters are the weights. This is why the error function must be minimized according to these. To do this, the derivative is set to zero:

$$\left.\frac{\partial E}{\partial w}\right|_x = 0 \qquad \forall x, w \tag{2.5}$$

Note that the limitation of this technique is that the minimization is local and not a global one. Also, as E is not known at all points, the resolution of this derivative is done using numerical methods. In particular, the gradient descent [37].

$$w^{new} = w^{old} - \alpha \left.\frac{\partial E}{\partial w^{old}}\right|_x \qquad \text{where } \alpha \text{ is the learning rate} \tag{2.6}$$

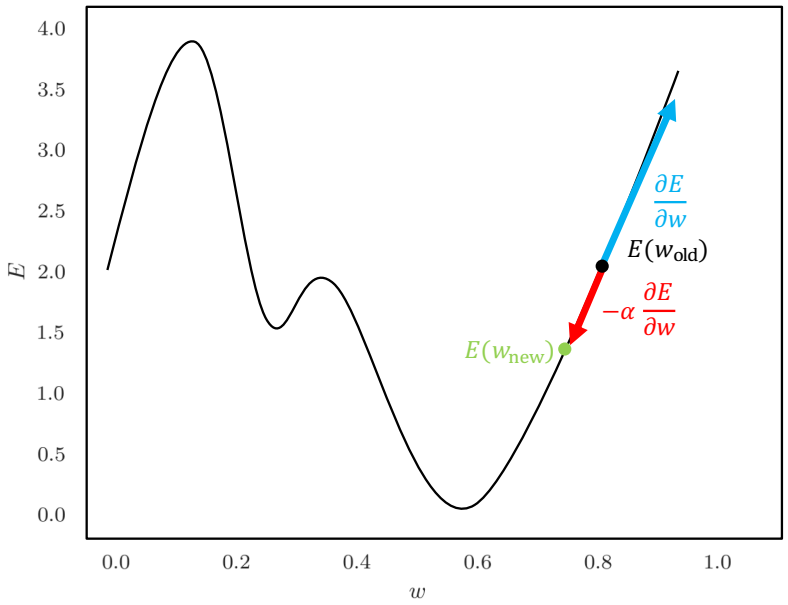

Figure 2.5: Gradient descent iteration.

The complete learning algorithm can be summarized in a cycle of three steps [3, 4]. This cycle is called an epoch and is repeated until the model converges:

- Inference: The model predict y_m for each sample x_m.
- Evaluation of the error: The results y_m are compared with the target t_m.
- Update of the weights: The weights are updated according to the Eq.2.6.

Activation function

For the basic case, the activation function φ can be considered as the sign function such that :

$$\varphi(x) = \text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

Thus, the function of the perceptron is :

$$y_m = f(\mathbf{X}_m) = \begin{cases} 1 & \text{if } \mathbf{W}\mathbf{X}_m > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

It is therefore understandable that the objective is to adapt the weights so that the weighted sum is positive when $t = 1$ and negative otherwise.

Recall Eq.2.9 and Eq.2.6, the activation function has to be derivable. However, the sign function is not. So it is often replaced by one of the derivable functions presented in the Fig. 2.6. This is done at the detriment of y_m that will no longer have a binary value but between 0 and 1 (except for ReLu).

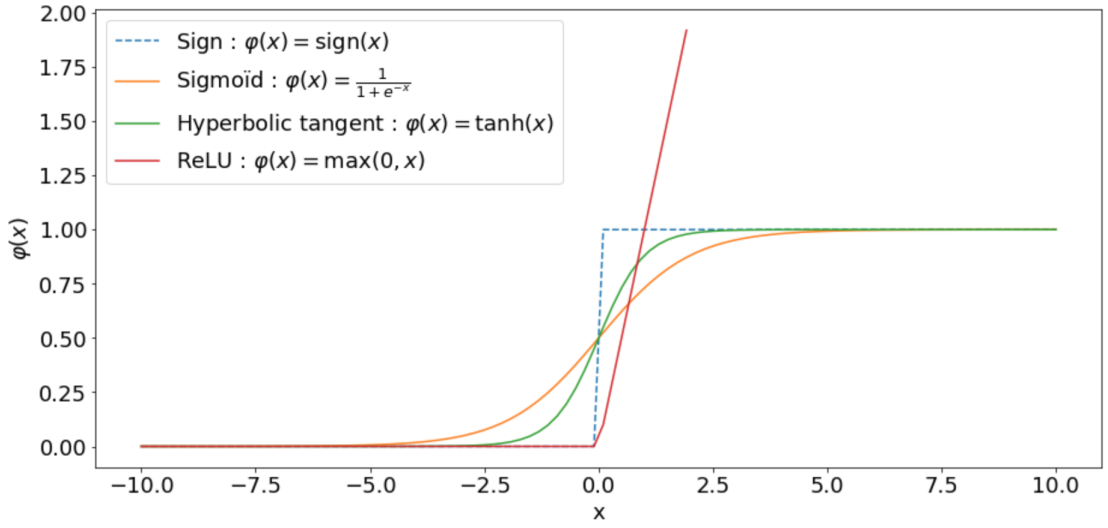

Figure 2.6: Different standard activation function.

Error function

The naive way to implement the error function would be too compute an average error of the predictions:

$$E = \frac{1}{M} \sum_{m=0}^{M} |t_m - y_m| \qquad \text{with } y_m = \varphi(\mathbf{W}\mathbf{X}_m) \tag{2.9}$$

But, the absolute value not being derivable, it is often replaced by another derivable error function such as the MSE.

$$\text{MSE} = -\frac{1}{M} \sum_{m=0}^{M} (t_m - y_m)^2 \tag{2.10}$$

Graphic representation

Finally, the Fig. 2.7 is the graphic representation of the trained classifier. One can see the linear separation allowing to determine if it is a cat or not. Note that only two features are shown for readability reasons. Indeed, in reality it is a hyper-planar separation in a hyper-space of dimension N.

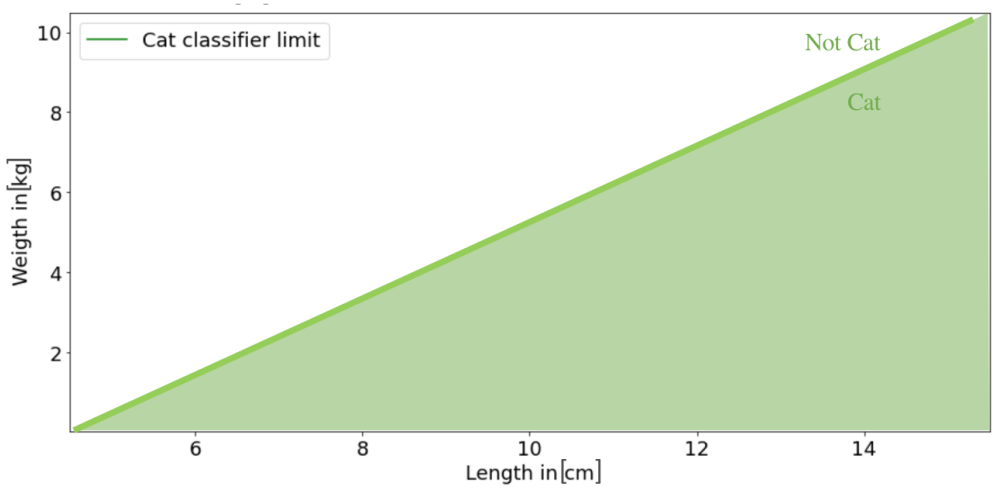

Figure 2.7: Graphic representation of the linear classification of animals according to their length and weight.

2.3.2 Multi-layer Perceptron : MLP

The Multi-Layer Perceptron (MLP) is a stack of layers of several perceptrons. If the simple perceptron allows a linear separation of elements, the MLP allows combinations of classifiers and hence to treat more complex data.

Example

For example, it is possible to imagine combining animal classifiers to determine where the survey comes from as shown in the Fig. 2.8. The zoo being the only place to find cats, fish and snakes. As for the Fig. 2.7 , for reasons of readability, only two dimensions are presented.

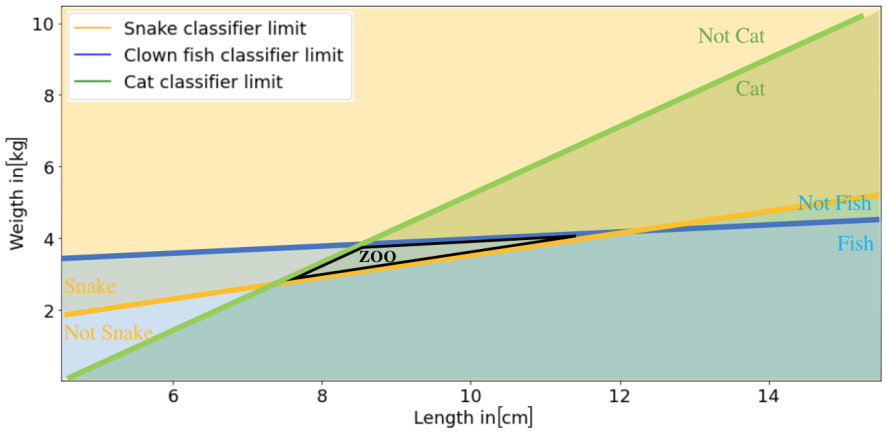

Figure 2.8: Graphic representation of the linear classification of animals according to their length and weight.

Here is the classifier that is used for this classification. It combines different classifiers to determine if the survey comes from a zoo.

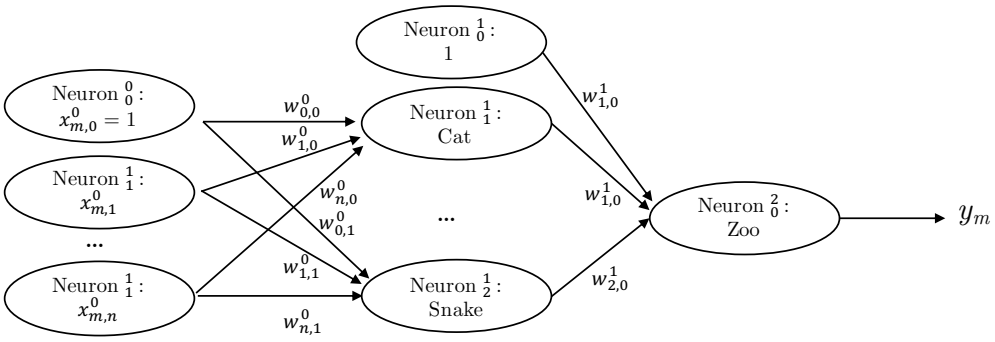

Figure 2.9: Graphic representation of the linear classifier for survey origin.

Definitions

Several layers of varying size can be assembled. It is said that there is [38] :

- Input layer : The layer in which the features are entered. There is no perceptron in this layer.
- Hidden layers : These are the layers that actually form the neural network.
- Output layer : The one that gives the results of the classifications.

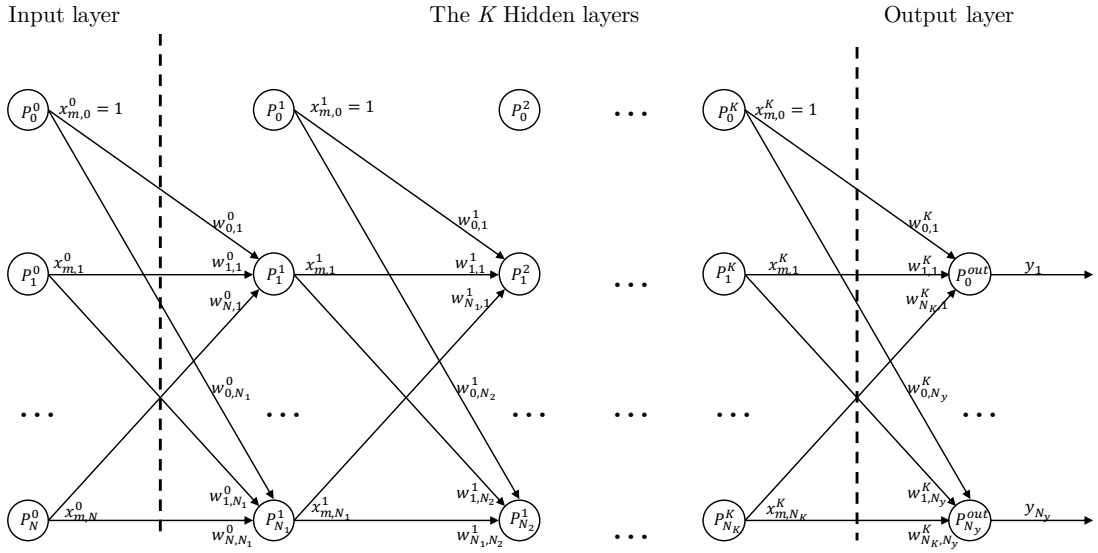

Figure 2.10: General representation of Multi-layer perceptron.

The size of a layer (the number of neurons that it contains) allows to generate several types of classifier that will be used in the next layer. Thus, one can see on the Fig. 2.8 that the classifier "Zoo" uses a combination of the linear classifier of "cat", "snake" and "fish".
Hence, the Universal Approximation Theorem indicates that any function can be approximated with an MLP if it contains at least one hidden layer [39, 40, 41]. Indeed, visually interpreted on the Fig. 2.8, this indicates that any area, no matter its shape, can be framed by a combination of lines.

The depth of the network (the number of layers) allows to build more complex separations. If one adds a layer to the network of the Fig. 2.9, one could then make more complex classification using polygons in the Fig. 2.8. This advantage is therefore widely used to create Deep Neural Networks ([42]).

It is important to note that the output layer can contain several neurons. Indeed, it is often a question of a multi-class classification. For example, in the case studied, it is necessary to know to which digit the sound refers. This is why the output layer of the system consists of 10 classifiers, one for each digit.

Recall that the activation functions are no longer discrete, the predictions are between 0 and 1. In the case of a classification in N_d class, it is interesting to normalize the outputs. The softmax function S allows to give a pseudo probability for each class d, such that:

$$y_d = S(x_d) = \frac{e^{x_d}}{\sum_{d'=0}^{N_d} e^{x_{d'}}} \qquad \text{with} \sum_d S(x_d) = 1 \tag{2.11}$$

Finally, if one wants to have only one prediction per digit, one can adopt the winner-takes-all strategy which consists in choosing the one with the maximum probability such that :

$$\hat{d} = \underset{d}{\text{argmax}}\ y_d \tag{2.12}$$

Note that it is on y_d that the model is trained and not on $\hat{d}$, the argmax function not being derivable. However it is usual to evaluate the final quality of the model on $\hat{d}$.

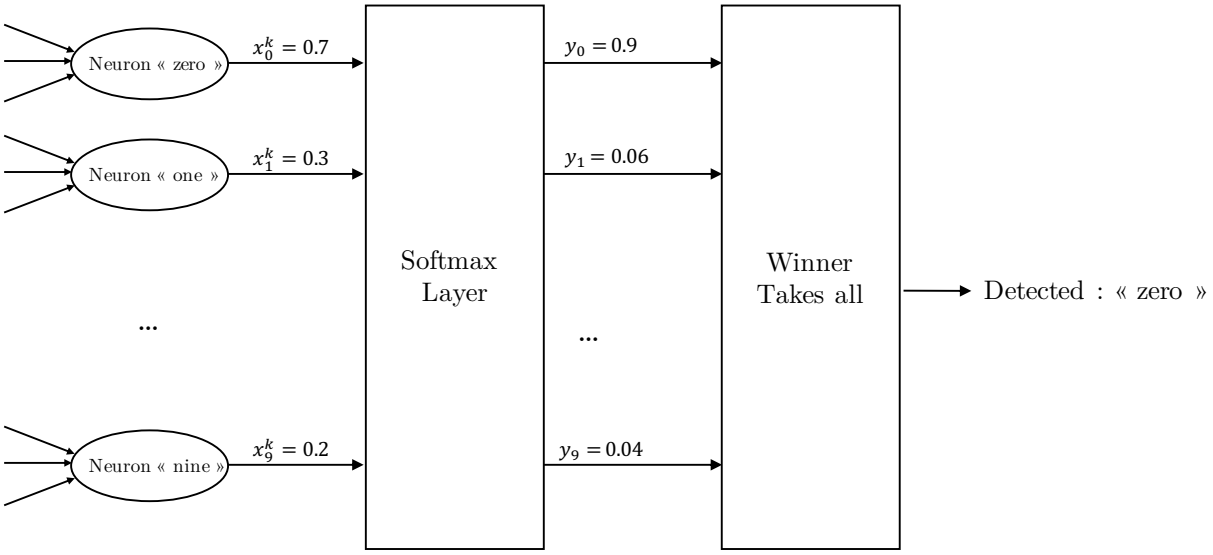

Figure 2.11: Usual output layer extension.

Training

To train such a network, the same method is used as for the simple perceptron. The error function must be minimized.

However, to ensure that all weights are involved in minimizing the error, the gradient backpropagation is applied [3, 4]. This technique allows to calculate the impact of each weight on the error by going back along the network. First, here is a brief reminder of the variables involved:

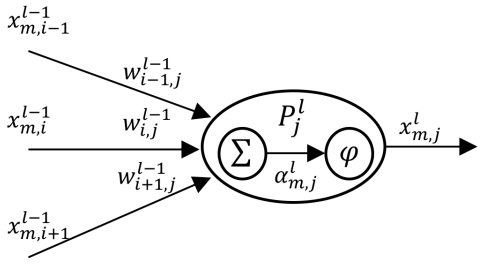

Figure 2.12: Zoom on a singe general neuron.

- $x_{m,i}^l$: stands for the output for the m^{th} sample from the i^{th} neuron of the l^{th} layer.
- $w_{i,j}^l$ stands for the weight of the synapse coming from the i^{th} of the layer l going to the j^{th} neuron of the layer $l+1$.
- P_j^l stands for the j^{th} neuron/perceptron in the l^{th} layer.
- $a_{m,j}^l$ stands is the weighted sum given as input for the activation function φ

To calculate the influence on the error of each weight, the chain rule is applied:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{i,j}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}} \tag{2.13}$$

With a_j the weighted sum such that :

$$a_{m,j}^l = W_j^{l-1} X_m^{l-1} \tag{2.14}$$

Thus, as a_j is a function of the weights and outputs of the previous layers, it is possible to compute all the $\frac{\partial E}{\partial w_{i,j}}$ in an iterative way. Therefore, it is possible to generalize the gradient descent [37]:

$$w_{i,j}^{new} = w_{i,j}^{old} - \alpha \frac{\partial E}{\partial w_{i,j}^{old}}\bigg|_x \qquad \text{where } \alpha \text{ is the learning rate} \tag{2.15}$$

2.4 Convolutional neural networks

The Convolutional Neural Network (CNN) is a particular architecture of neural networks. It is a more efficient architecture for image processing [1] (and 2D features with similarities in their neighborhood).
In the case of speech recognition, it can be applied. Indeed, one of the axes is the time while the other is the frequency (or a cepstrum) both having this characteristic. Here is how this architecture works.

2.4.1 Convolutionnal layer

One assumes that in an image the neighboring pixels have a great similarity. So, instead of processing each pixel independently, it processes pixel neighborhoods. Instead of having fully-connected layers, it has layers that are slightly connected. This ensures a fast and effective learning process due to the strategic choice of the kept connections.
The processed neighborhood is called the receptive field. To extract the information from each neighborhood, one covolves a kernel with the image (one multiplies each pixels of each receptive field with a kernel and then sums, explaining the term convolution). On the Fig. 2.13 one applies a kernel of 3×3 pixel. The result of the convolution between an image and a kernel is called a feature map.

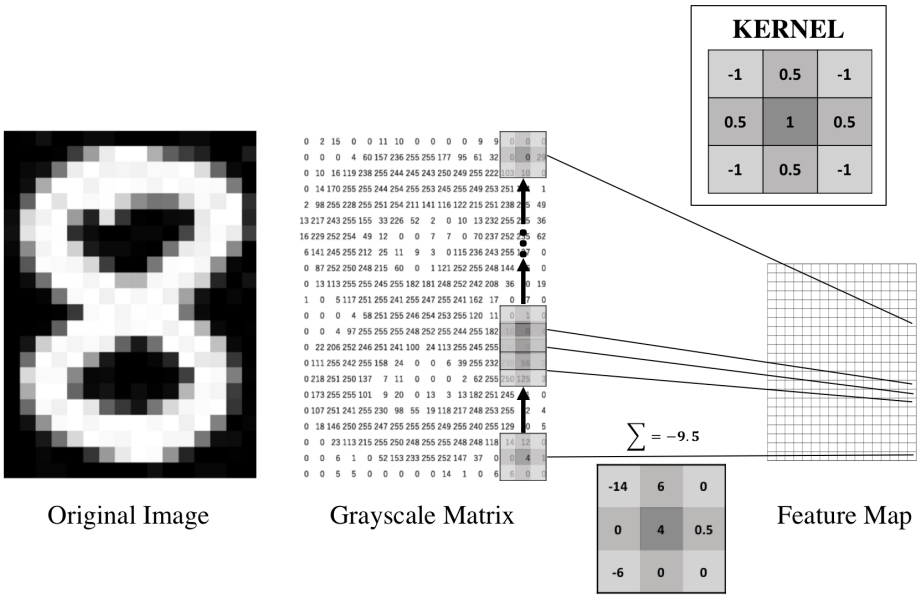

Figure 2.13: Kernel convolution with a greyscaled image.

To extend the model, one can apply several different kernels and thus create several feature maps. In comparison with Feed-forward:

- The receptive fields : These are the inputs of the neurons.
- The kernels : These are the weights of the synapses. They are the ones that will be trained.
- The feature maps: these are the outputs of the neurons. The size of the layer is equal to the number of features map C^l. It is also called the channels.

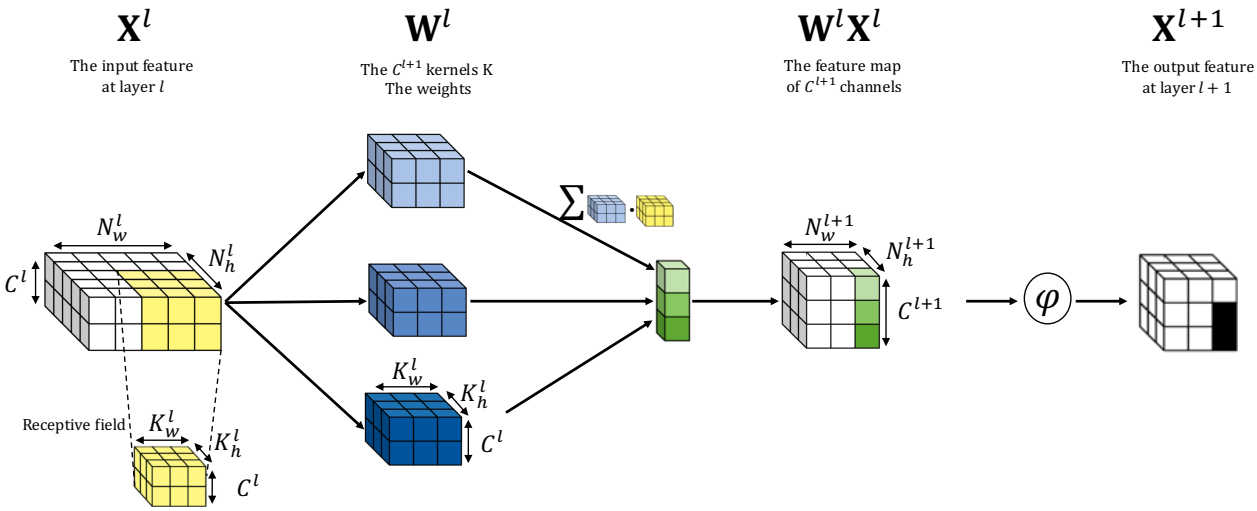

Figure 2.14: General representation of the convolutional neural network.

2.4.2 Pooling layer

To improve the efficiency of the training, one can subsample the features map. Indeed, it allows to extract, with very little loss of information, the essence of a zone while dividing the number of parameters to train.
For this, the so-called pooling is used. One possibility is to keep only the maximum of a neighborhood. This method is called maximum pooling. Figure 2.15 shows an application of a maxpool layer on a feature map.

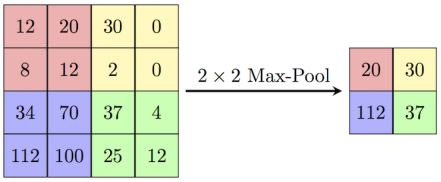

Figure 2.15: Example of a maxpool on a feature map.

2.5 In Practice : Implementation of a state of the art model

To benchmark the performance of the model, a state-of-the art spoken digit recognition model is implemented. The one chosen is the M5 model from [43]. Indeed, it is a lightweight model that claims to have the same performance as simple models using preprocessing such as MFCC. Its performance will serve as a benchmark for the final model Here is a description.

This model is a very deep neural network since it is composed of several layers:

- Input Layer : The inputs of this model are directly the raw data of the sounds. Being sampled at 16 kHz, their length is normalised to one second, which makes input vectors of 16000 elements.
- Deep Layer : The input layer is then followed by four pairs of convolutional and max pool layer. The convolutional layer doubles the number of kernels used each time ($C^{l+1} = 2C^l$) while the pooling layer reduces the size of the feature map by four ($N^{l+1} = 0.25 \cdot N^l$). Given the unidimensional nature of the inputs, the layers are also uni-dimensional.
- Global average pooling Layer : It averages each feature map along the temporal axes into a single value. The number of feature maps is N_d.
- Softmax layer : Based on the feature map values it gives a pseudo-probability for each class.

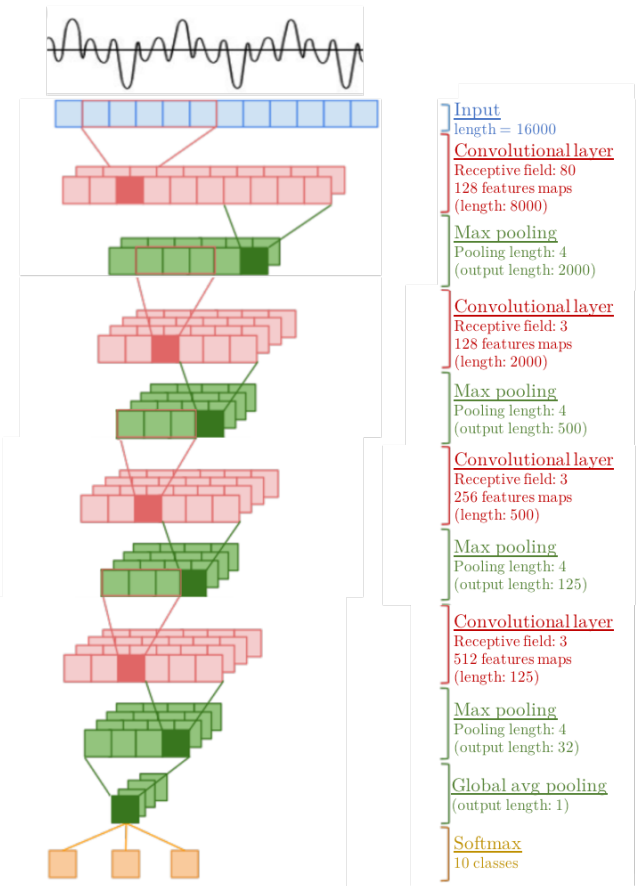

Figure 2.16: State of the art CNN for spoken digit recognition, adapted [43].

In total, this model has 25290 training weights. This makes it a rather lightweight model compared to more popular CNN models. For example the well known VGG model has over a million parameters to train [44].

Conclusion

This concludes the chapter on machine learning.

It was first seen how to process datasets in an efficient and methodical way. For this the K-fold corss-validation technique was studied. It was also shown how to extract important data from waveforms through feature extraction. This can be done with MFCC or Cochlear techniques.

Then the general concepts of machine learning were discussed. The simple and then the multilayer perceptron were described. Then the architecture of the convolutional neural network was presented.

Finally, a state of the art model was implemented to allow benchmarking of the final model.

Chapter 3

Reservoir computing

Reservoir computing is a bio-inspired neural network architecture [5, 6]. This architecture is critical to the development of the prototype because it allows the hardware integration [9, 27, 10].Also, it has many other advantages. It allows for fast and reliable learning, using simple linear algebra concepts. Its training is also deterministic and offers a global optimum.

The following chapter first provides a general description of reservoir computing. It discusses its training and algorithms. It shows how the training can be solved by a simple matrix inversion.

Next, it is described how this algorithm can be adapted to a neuromorphic device and what properties it should have. It also explains how the algorithms are adapted for this purpose.

Finally, it introduce the neuromorphic device studied in this work: the spin-torque nano-oscillator (STNO). First, a brief description of the device is given, followed by a study of the numerical resolution of its dynamics. Indeed, in order to ensure that the device could be integrated into the hardware, a preliminary simulation of its behaviour in the neural network was necessary.

Figure 3.1 shows the position of this chapter in the overall work-flow. Note that it is really at the border between software and hardware as it introduces its integration.

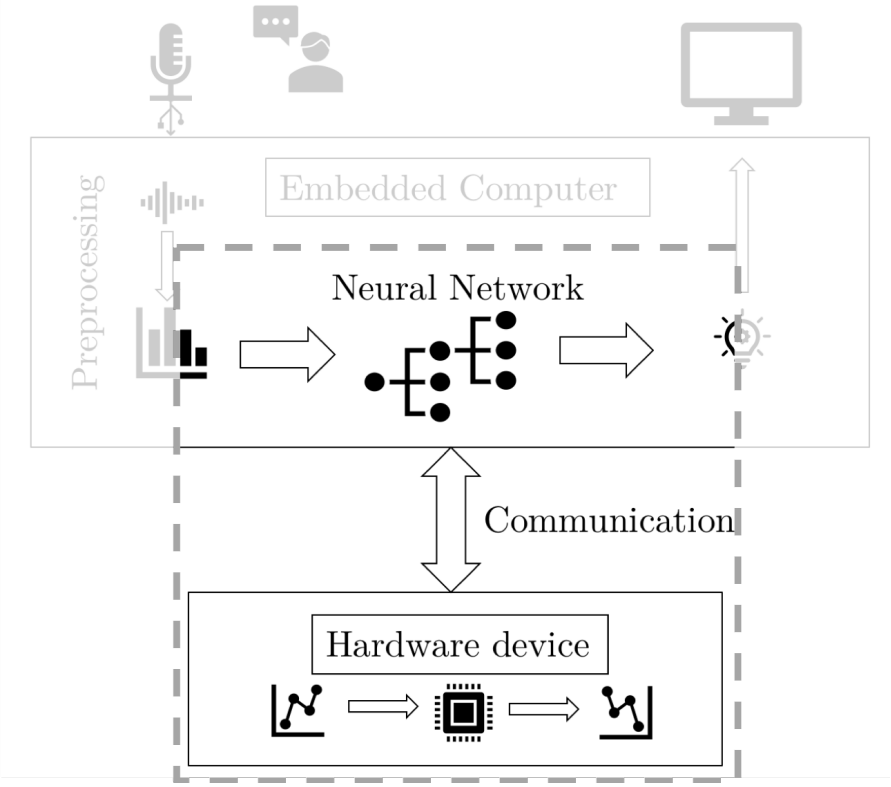

Figure 3.1: Hardware model adaptation in the global work-flow.

3.1 General description

Reservoir computing is an architecture belonging to the family of recurrent neural networks (RNN). The RNN are different from other architectures because of the cycles they present. These cycles offer a certain memory to the model that allows it to be better in tasks involving causality in the data. This is why RNNs are often used in speech recognition.

Definitions

The reservoir computing is made up of several elements.

- Input **X**: The input layer is linked to the reservoir by a set of randomly chosen synaptic weights. This set is called the mask **M** and is kept constant during the all process.
- Reservoir: The reservoir is a set of neurons (perceptron or other nonlinear function) and connected by synapses, also random and fixed. The organization of the neurons between them does not follow any rules.
- Output Layer : In the output layer is performed the readout of the reservoir. The readout consists in a weighted reading **W** of the outputs **V** of some neurons of the reservoir. The readout is performed after a certain duration τ of evolution.

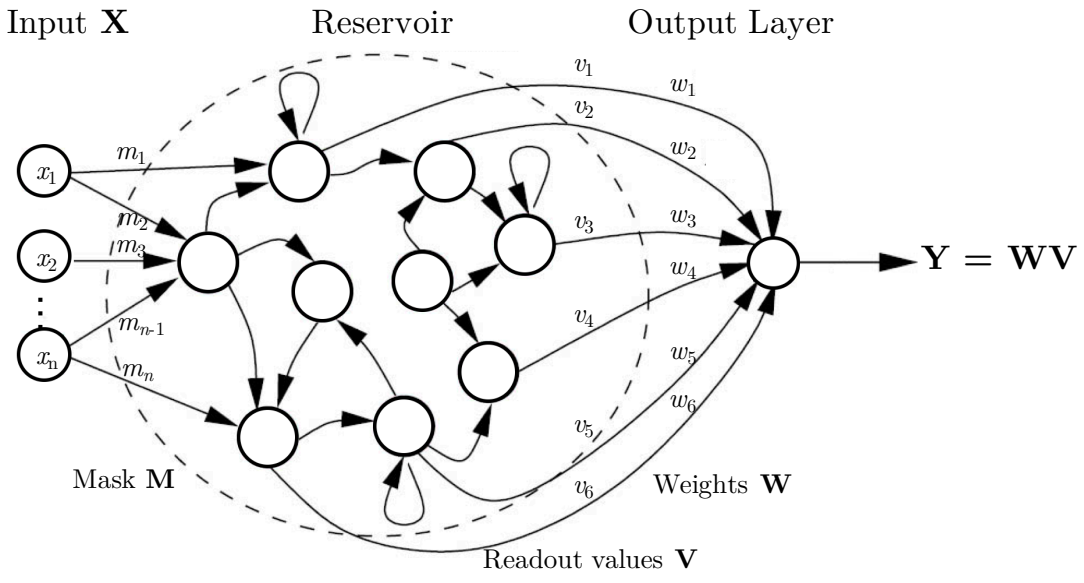

Figure 3.2: General representation of the reservoir computing, adapted from [45].

Training

The training of such an architecture is very fast [5]. Indeed, the output is summarized as :

$$\mathbf{Y} = \mathbf{WV} \qquad \text{with } \mathbf{V} = \text{Reservoir}(\mathbf{MX}) \tag{3.1}$$

If one imposes that the outputs tends to the targets such that $\mathbf{Y} \to \mathbf{T}$, one gets:

$$\mathbf{T} \approx \mathbf{Y} = \mathbf{WV} \tag{3.2}$$

Finally, in order for the model to be trained, the weight has to be adapted. For this, Moore-Penrose [7, 8] pseudo-inverse is used, minimizing the least square inverse approximation, it is equivalent to a simple linear regression :

$$\mathbf{W} = \mathbf{TV}^{-1} \tag{3.3}$$

This learning is very fast and allows a global solution for training.

Complete Algorithm

The complete algorithm for the use of reservoir computing is as follows.

1. Multiplication of inputs by a random mask $\mathbf{M}$.
2. Convergence of the reservoir for a period of time τ.
3. Weighted sum by $\mathbf{W}$ of the readout values $\mathbf{V}$.
4. Training of the model on the training set with Eq.3.3.
5. Inference of new data in the model according to Eq.3.1.

3.2 Hardware adaptation

An adaptation of the computing reservoir can be made for the use of a single neuron with memory [9]. First, the architecture of the reservoir is simplified to a simple loop. Moreover, the readout and the mask connects all the neurons of the reservoir.

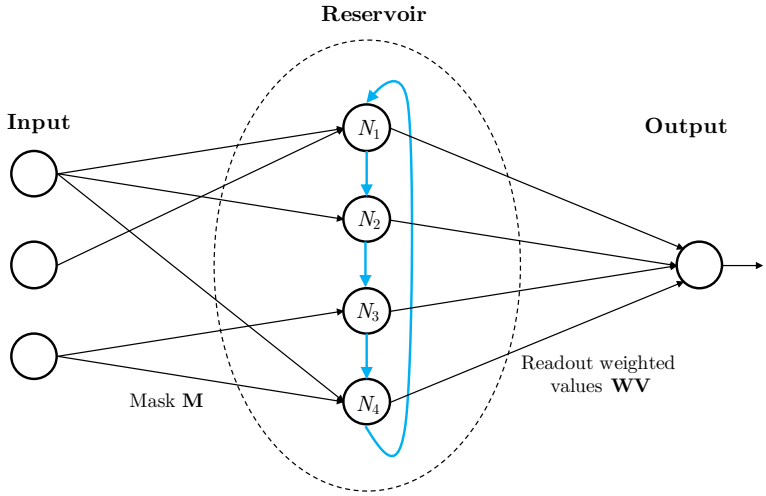

Figure 3.3: Hardware adaptation of reservoir computing, adapted from [9].

Then, it is important to understand that the memory of our hardware neurons will replace the connections of the neurons inside the reservoir (blue arrow $\rightarrow$ on the schematics). Thus, it will be possible to multiplex the neurons temporally. This means that the same unique neuron will always be used but at different times.

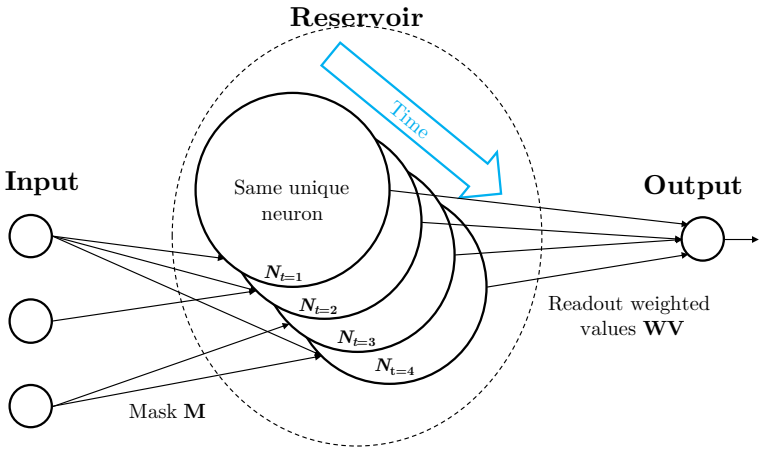

Figure 3.4: Time multiplexing of a single neuron for hardware adaptation.

Software adaptation

In practice, our non-linear function can only process one scalar at a time. The data stream will therefore be adapted to this purpose [28].

- Input data : These are the features to be processed for our model and are of the size $\mathbb{R}^{N_f \times N_\tau}$. In this case the inputs are the results of the application of a feature extraction technique. The N_f corresponds to the number of extracted features while the N_τ corresponds to the number of windows of size τ that the signal contains.

- Masked input : They are the result of the matrix product between the mask $\mathbf{M}$ of size $\mathbb{R}^{N_\theta \times N_f}$ and the inputs. The N_θ is the number of neurons present in the reservoir.

- Flatten input : as explained before, the neuron can only process one scalar at a time. This is why the inputs are flatten in order to be a continuous flow of data passing through the neuron. The flatten function transforms the masked input matrix of size $\mathbb{R}^{N_\theta \times N_\tau}$ into a data stream of size $\mathbb{R}^{1 \times N_\theta N_\tau}$.

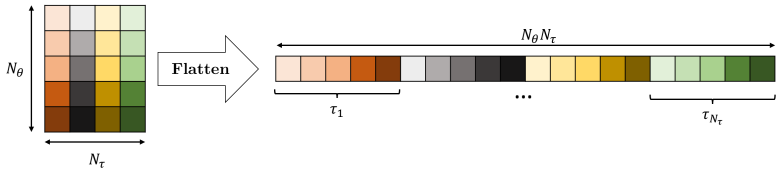

Figure 3.5: Representation of the function flatten.

- Flatten Readout : The data are transformed one by one in the non linear neuron. The result is a vector of the same size as the input vector $\mathbb{R}^{1 \times N_\theta N_f}$..

- Weighted Readout : The inverse function of the flatten is applied and the readout can be weighted with the weight matrix $\mathbf{W}$ of size $\mathbb{R}^{1 \times N_d N_\theta}$.The number N_d is the number of neurons in the output layer. That is to say the number of possible classes to which to assign the inputs. The weighed readout matrix is therefore of the size $\mathbb{R}^{1 \times N_d N_\tau}$.

- Mean readout : Finally the scores of each τ is averaged. Thus, one scalar per digit is obtained, so a result of size $\mathbb{R}^{N_d \times 1}$.To determine which digit is the recognized one, the Winner-takes-all strategy is applied.

  Note that in the case of real time processing, this step can be removed and a τ-wise recognition can then be performed. Of course, this recognition will be less accurate because it will not have the whole sound to classify.

The Fig. 3.6 summarises the complete data flow in the architecture.

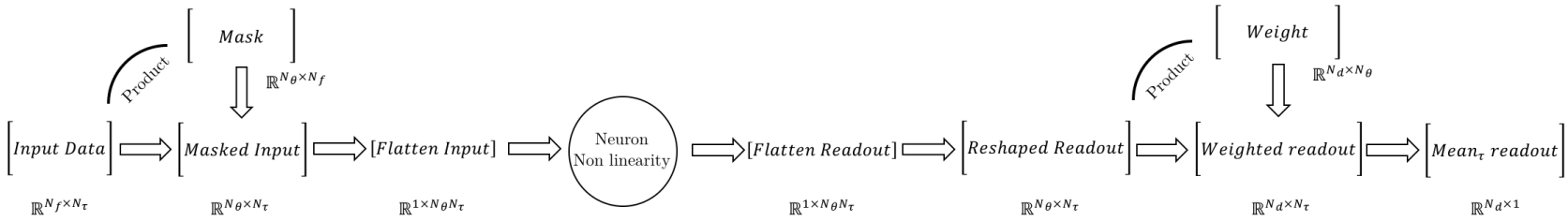

Figure 3.6: Complete data flow for the hardware adaptation of reservoir computing.

In practice

In practice, the variables have the following values:

- N_f : Depending on whether MFCC or Cochlear is used, this value is 13 or 78 respectively.
- N_τ : Depending on speech duration, $N_\tau^{Cochlear}$ ranges from 16 to 41 and N_τ^{MFCC} ranges from 31 to 83.
- N_θ : The number of neurons chosen is 400.
- N_d : The number of possible classes is 10.

3.3 The STNO : a neuromorphic devices

The previous section introduced reservoir computing. More specifically, an implementation of it allowing the use of a single device. It is precisely this device that this section is about.

It is thus necessary to incorporate a device having a memory component and making a nonlinear transformation. It must take as input a vector $\mathbf{X}$ of size $\mathbb{R}^{1\times N_\theta N_\tau}$ and output a vector $\mathbf{Y}$ of the same size.

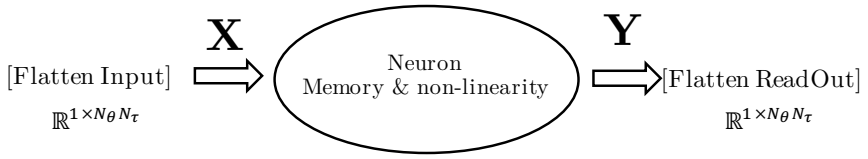

Figure 3.7: Neuron positioning and requirements for reservoir computing adaptation.

In this study, a spin-torque nano-oscillator (STNO) is used as the neuron. Indeed, in addition to providing the required property, it shares the same structure as current magnetic memory cells and is therefore compatible with CMOS technology. It also has high durability, operates at ambient temperature and can be manufactured in large numbers on a single chip [10].

This section start with a brief description of the STNO and is followed by details of its dynamics and how it has been simulated.

3.3.1 STNO : general description

First, a brief introduction to the spin-torque nano-oscillator (STNO)

As described in Fig. 3.8 a spin-torque nano-oscillators is a nanoscale stack of three layers: two ferromagnetic surrounding a non-magnetic one. When current flows through this pillar, it becomes spin-polarized which generates torques on the magnetizations. That implies a sustained magnetization precession. A voltage oscillation is derived from magnetization fluctuation through magneto-resistance.

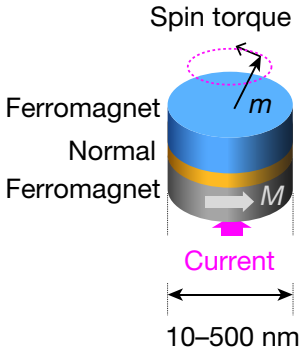

Figure 3.8: Schematic of a spin-torque nano oscillator.

In this system, the STNO is driven by the combination of a constant DC current I_0 and a varying voltage. This varying voltage is a proportional transformation (by a factor V) of the input $\mathbf{X}$, the same as the flatten input in Fig. 3.7. The output is the measured voltage $\tilde{V}$, so the $\mathbf{Y}$ in Fig. 3.7.

The complete dynamic of the STNO for an input x[i] returning an output voltage y[i] is such that :

$$STNO(x[i]) = \begin{cases} y[i] = O[i] \cdot (1 - e^{\frac{-dt}{\tau}}) + e^{\frac{-dt}{\tau}} \cdot y[i-1] & \text{if i} \neq 0 \\ y[0] = O[0] & \text{if i} = 0 \end{cases} \tag{3.4}$$

where : $O[i] = p_A \sqrt{\left|(I_0 - I_c) - \frac{V \cdot x[i]}{R}\right|}$
with the following parameter :

- $dt = 5\ ns$: the time step between two inputs.
- $\tau = 410\ ns$: Oscillator relaxation time.
- $R = 52.5\ \Omega$: Resistance of the oscillator
- $V = 250\ mV$: Amplitude of the voltage from 0 to peak.
- $pA = -2.3754$: The constant before the square root (offset).
- $Ic = 4.9\ mA$: The critical current
- $I_0 = 6.0\ mA$: applied DC current.

3.3.2 STNO Simulation

This section discusses the dynamics of STNO. First, it gives a rough approach to its dynamics. Then, to allow more accurate simulations, a refinement of the formula is undertaken.

Global description

The STNO can be described as follow :

$$\text{STNO}(x[i]) = \begin{cases} y[i] = O[i] \cdot (1 - e^{\frac{-dt}{\tau}}) + e^{\frac{-dt}{\tau}} \cdot y[i-1] & \text{if i} \neq 0 \\ y[0] = O[0] & \text{if i} = 0 \end{cases} \tag{3.5}$$

where : $O[i] = p_A \sqrt{\left|(I_0 - I_c) - \frac{V \cdot x[i]}{R}\right|}$

For clarity, one defines the following variables (note that p_A is negative):

$$p_\Delta = p_A^2 \cdot (I_0 - I_c) \quad \text{and} \quad p_I = p_A^2 \cdot \frac{V}{R} \quad \text{and} \quad E = e^{\frac{-dt}{\tau}} \tag{3.6}$$

The dynamics of the STNO is therefore described by :

$$STNO(x[i]) = \begin{cases} y[i] = O[i] \cdot (1 - E) + E \cdot y[i-1] & \text{if i} \neq 0 \\ y[0] = O[0] & \text{if i} = 0 \end{cases} \tag{3.7}$$

where : $O[i] = -\sqrt{|p_\Delta - p_I \cdot x[i]|}$

Interpolation

For more accuracy in the model, one interpolates two consecutive inputs into K points such that :

$$\text{Interp}_{x,i}[k] = (x[i] - x[i-1]) \cdot \frac{k}{K} + x[i-1] \tag{3.8}$$

And such that :

$$x[i] = \text{Interp}_{x,i}[K] \quad \text{and} \quad x[i-1] = \text{Interp}_{x,i}[0] \tag{3.9}$$

This gives :

$$\begin{aligned} O_{\text{Interp}_{x,i}}[k] &= -\sqrt{\left|p_\Delta - p_I \cdot \text{Interp}_{x,i}[k]\right|} \\ &= -\sqrt{\left|p_\Delta - p_I \cdot (x[i] - x[i-1]) \cdot \frac{k}{K} + x[i-1]\right|} \\ O_{\text{Interp}_{x,i}}[k] &= -\sqrt{\left|p_\Delta - p_I \cdot x[i-1] + (x[i-1] - x[i]) \cdot \frac{k \cdot p_I}{K}\right|} \end{aligned} \tag{3.10}$$

The STNO model can therefore be summarised as follows:

$$\mathrm{STNO}_{\mathrm{Interp}}(x[i]) = \begin{cases} y_{\mathrm{Interp}_{x,i}}[k] = O_{\mathrm{Interp}_{x,i}}[k] \cdot (1-E) + E \cdot y_{\mathrm{Interp}_{x,i}}[k-1] & \text{if i} \geq 0 \\ y_{\mathrm{Interp}_{x,0}}[0] = O_{\mathrm{Interp}_{x,0}}[1] = y[-1] & \text{if i = -1} \end{cases} \tag{3.11}$$

And such that :

$$y[i] = y_{\mathrm{Interp}_{x,i}}[K] \quad \text{and} \quad y[i-1] = y_{\mathrm{Interp}_{x,i}}[0] \tag{3.12}$$

Note that, as illustrated on Fig. 3.9, to find the value of $y[0]$ an interpolation of $\mathbf{X}$ is needed between $x[-1]$ and $x[0]$. It is therefore necessary to add an element to the matrix $\mathbf{X}$. It is decided to add it at the beginning, which leads to the use of the index -1. There is also a change in the initial condition which therefore starts at $y[-1]$. Obviously, this scalar is just useful for the calculation of its successors but will not be retained in the final vector $\mathbf{Y}$.

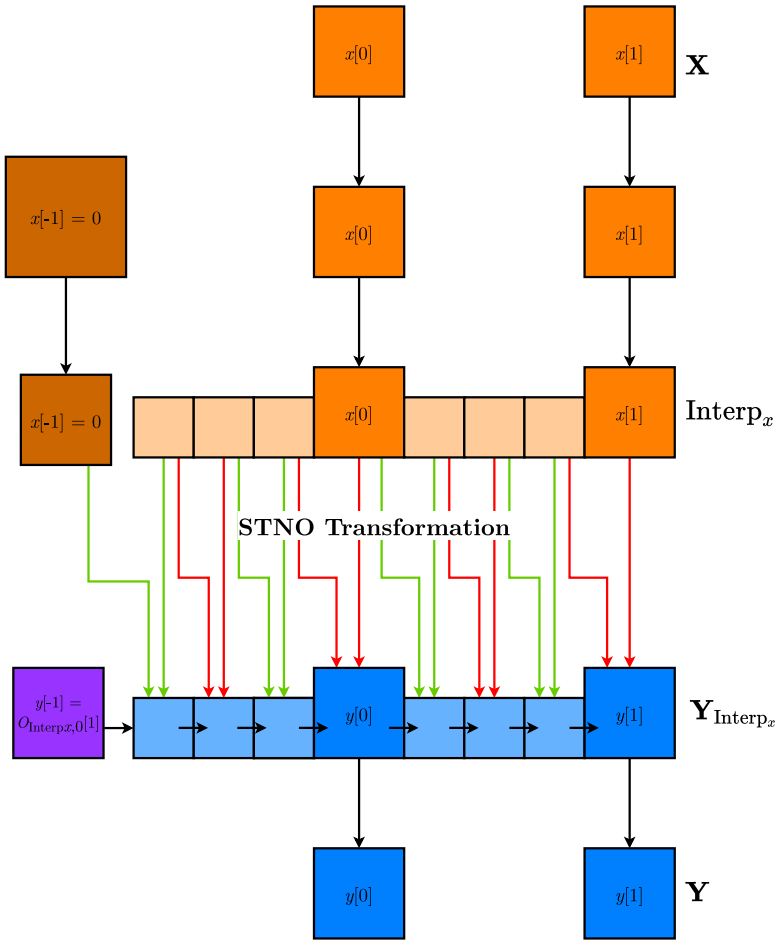

Figure 3.9: Element inserting to allow proper interpolation.

3.4 In Practice : Python implementation

In practice, this algorithm has been implemented in python in order to verify its proper working. The number of interpolation points is set to $K = 20$.

The results are successful since it recognises the data well. The training is done in a reasonable amount of time and the inference is done in a period of time acceptable for real time. The detailed results will be presented at the end of this work.

Conclusion

This concludes the first steps towards the hardware integration of a neuromorphic device, in this case a spin-torque nano-oscillator.

First, the description of the reservoir computing and its training were discussed. This section showed how it is possible to train such a network globally with simple linear algebra.

Then its adaptation for the integration of a single neuron with memory and non-linearity was tackled. A presentation of the adaptation for its implementation was also given.

Finally, the STNO acting as a neuron was described. A general and numerical description was presented. Finally, a complete software implementation of reservoir computing integrating the STNO models was implemented. The results are successful.

Chapter 4

Implementation of an ASIC prototype

In [10, 28], a reservoir computing has already been used to integrate STNO. However, it was built with very large devices around the nano-oscillator. The objective of this thesis is to make such a embedded version of that system and easily adaptable to other devices offering memory and a non-linear transformation.
In order to demonstrate the feasibility of the embedded use of a hardware device, the study focused on the design of an hardware accelerator simulating the behaviour of a STNO. The hardware accelerator is also called an application specific integrated circuit (ASIC).

The following chapter describes the implementation of a prototype of ASIC with all the steps that were necessary for its design :

1. First of all, a brief review of digital electronics concepts is given.
2. Next, the adaptation of the algorithm to make it implementable and efficient is discussed.
3. Then, the data communication between the software and the ASIC via the interfaces is detailed.
4. The final section is devoted to the actual implementation of the prototype of ASIC with all the sub-modules it contains.

Figure 4.1 presents the integration and the elaboration of the hardware device in the global work-flow. Note that the entire system has been designed so that any device with memory and non-linearity can be interchanged, as long as it can communicate with the software.

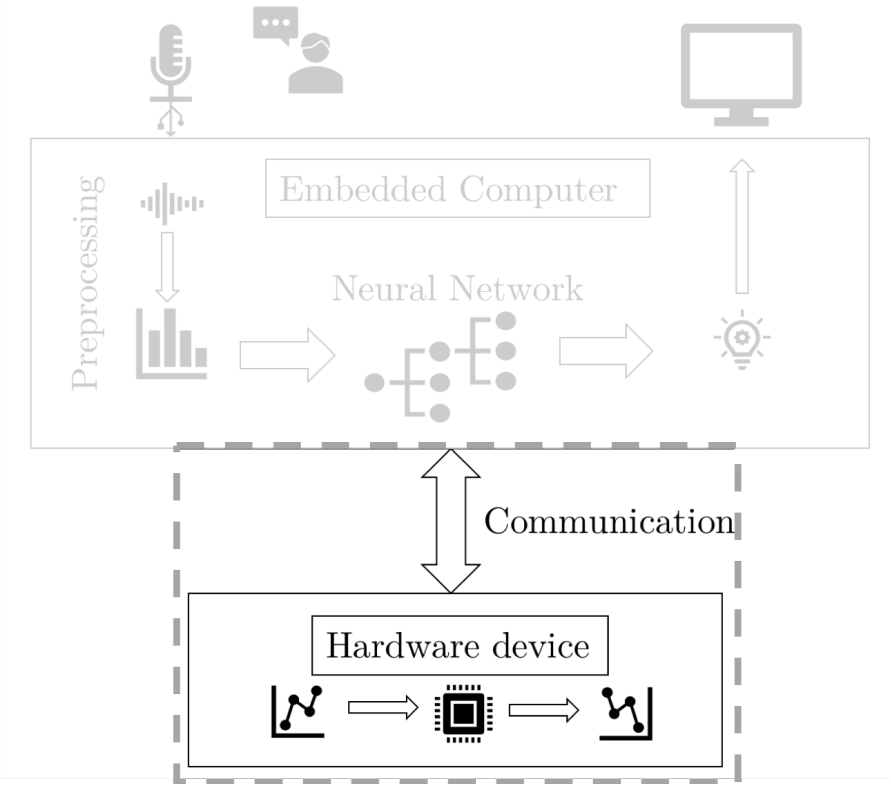

Figure 4.1: Hardware device in the global work-flow.

4.1 Digital electronics concepts

First of all, to enable the reader to understand the section properly, important concepts are required.

- Cloks and synchronicity : The clock is used to create a rhythm for a system. Indeed, all the operations in a system must be clocked in order to allow a synchronicity between them. The clock therefore generates a binary signal that oscillates at a fixed interval. This interval is called a clock cycle. When an operation is said to be executed in $\tau = n$, it means that it takes n clock cycle to be executed.

- Operation parallelism : Two operations are parallelizable if they are independent, i.e. no result of one is needed to compute the other. Therefore, no matter how they are processed, their result is the same. If both operations are executed at the same time, they are said to be executed in parallel.

- Pipelining : A module doing several operations is pipelined if it can process new data without having finished processing the previous ones.
  For example, let a module take the opposite of the inverse of a variable x ($f(x) = -1/x$ with $\tau_{\text{inv}} = \tau_{\text{opp}} = \tau$). Two variables a and b are submitted to it. Here are the pipelined execution steps:

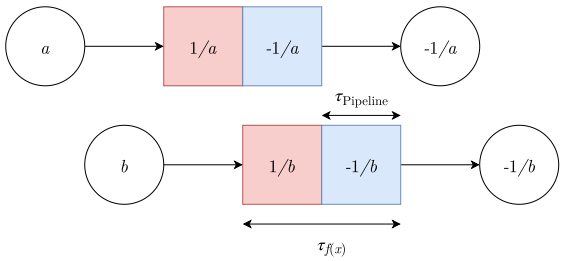

Figure 4.2: Example of pipelined operation.

1. Computes the inverse of a.
2. Computes the opposite of a and simultaneously the inverse of b.
3. Computes the opposite of b.

  Thus, the total execution took 3τ instead of 4τ if the operation had not been pipelined. The delay between two outputs is called τ_{Pipeline}. This delay is generally equal to 1.

- Data types and format : Data can have several types depending on how it is used. In the following sections, the main type used is `float32`. It is designed to store 32-bit floating point data.
  The format can also take many forms. In the following, the `float32` will generally be represented as 8 hexadecimal (each hexadecimal occupying 4 bits).

4.2 Adaptation of the algorithm

First of all, it is necessary to adapt the algorithm for the ASIC. Indeed, some modifications are necessary to allow a more efficient calculation, in particular the parallelization of the algorithm, presented in this part.

Motivation

The objective of the ASIC is to fast compute all $y[i]$ values. Let's define $\mathcal{I}$ the recursive function of the STNO, simulating the integration. Thus, the model becomes :

$$\mathrm{STNO}_{\mathrm{Interp}}(x[i]) = \begin{cases} y_{\mathrm{Interp}_{x,i}}[k] = \mathcal{I}_{\mathrm{Interp}}(y_{\mathrm{Interp}_{x,i}}[k-1], O_{\mathrm{Interp}_{x,i}}[k]) \\ \qquad\qquad = O_{\mathrm{Interp}_{x,i}}[k] \cdot (1-E) + E \cdot y_{\mathrm{Interp}_{x,i}}[k-1] & \text{if } i \geq 0 \\ y[-1] = y_{\mathrm{Interp}_{x,0}}[0] = O_{\mathrm{Interp}_{x,0}}[1] & \text{if } i = \text{-1} \end{cases} \tag{4.1}$$

One of the advantages of ASICs is their parallelisation capability, i.e. their ability to perform independent operations in parallel.

However, the calculations of $y_{\mathrm{Interp}_{x,i}}[k]$ are dependent on their predecessors. To calculate $y[i]$ from $y[i-1]$ one must sequentially calculate the intermediate values:

$$y[i] = y_{\mathrm{Interp}_{x,i}}[K] = \mathcal{I}_{\mathrm{Interp}}\left(y_{\mathrm{Interp}_{x,i}}[k-1] \ , \ O_{\mathrm{Interp}_{x,i}}[k] \text{ for } k \in [1,K]\right) \tag{4.2}$$

with $y_{\mathrm{Interp}_{x,i}}[0] = y[i-1]$

The calculations of $O_{\mathrm{Interp}_{x,i}}[k]$ are already independent. One would like to adapt the rest of the equation to make all the calculations independent and therefore parallelisable such that :

$$y[i] = y_{\mathrm{Interp}_{x,i}}[K] = \mathcal{I}'_{\mathrm{Interp}}\left(y[i-1] \ , \ O_{\mathrm{Interp}_{x,i}}[k] \text{ for } k \in [1,K]\right) \tag{4.3}$$

The basic case

As a first step, it would be interesting to modify the initial formula:

$$STNO(x[i]) = \begin{cases} y[i] = \mathcal{I}(y[i-1], O[i]) \\ \qquad = O[i] \cdot (1-E) + E \cdot y[i-1] & \text{if } i \neq 0 \\ y[0] = O[0] & \text{if } i = 0 \end{cases} \tag{4.4}$$

It would be desirable to deduce $y[i+N]$ from $y[i]$ without calculating the intermediate values. As for the Eq.4.2 and Eq.4.3 :

$$y[i+N] = \mathcal{I}\left(y[i+n-1] \ , \ O[i+n] \text{ for } n \in [1,N]\right) \tag{4.5}$$

that should be changed in :

$$y[i+N] = \mathcal{I}'\left(y[i] \ , \ O[i+n] \text{ for } n \in [1,N]\right) \tag{4.6}$$

Solving the reccursion

First, by expanding $y[i+1]$, one gets :

$$\begin{aligned} y[i+1] &= O[i+1] \cdot (1-E) + E \cdot y[i] \\ &= O[i+1] \cdot (1-E) + E \cdot (O[i] \cdot (1-E) + E \cdot y[i-1]) \\ &= O[i+1] \cdot (1-E) + O[i] \cdot E \cdot (1-E) + E^2 \cdot y[i-1] \end{aligned} \tag{4.7}$$

The following formula is obtained recursively:

$$\begin{aligned} y[i+N] &= y[i-1] \cdot E^{N+1} + (1-E) \cdot \sum_{n=0}^{N} O[i+n] \cdot E^{N-n} \\ y[i+N] &= y[i] \cdot E^{N} + (1-E) \cdot \sum_{n=0}^{N-1} O[i+1+n] \cdot E^{N-1-n} \\ y[i+N] &= y[i] \cdot E^{N} + (1-E) \cdot \sum_{n=1}^{N} O[i+n] \cdot E^{N-n} \end{aligned} \tag{4.8}$$

Therefore, it is possible to deduce $y[i+N]$ from $y[i]$ without calculating the intermediate values:

$$y[i+N] = \mathcal{I}'\left(y[i]\ ,\ O[i+n] \text{ for } n \in [1,N]\right) \tag{4.9}$$

The formula can therefore be parallelized.

Extending to interpolation

For interpolation, it is a matter of constructing a function such that :

$$y[i] = y_{\text{Interp}_{x,i}}[K] = \mathcal{I}'_{\text{Interp}}\left(y[i-1]\ ,\ O_{\text{Interp}_{x,i}}[k] \text{ for } k \in [1,K]\right) \tag{4.10}$$

Two interesting properties can be derived from Eq.3.12:

$$y[i] = y_{\text{Interp}_{x,i+1}}[0] \quad \text{and} \quad y[i+1] = y_{\text{Interp}_{x,i+1}}[K] \tag{4.11}$$

Thus, it is possible to calculate the interpolation of y such that:

$$\begin{aligned} y[i+1] &= y_{\text{Interp}_{x,i+1}}[0+K] \\ y[i+1] &= y_{\text{Interp}_{x,i+1}}[0] \cdot E^{K} + (1-E) \cdot \sum_{k=1}^{K} O_{\text{Interp}_{x,i+1}}[0+k] \cdot E^{K-k} \\ y[i+1] &= y[i] \cdot E^{K} + (1-E) \cdot \sum_{k=1}^{K} O_{\text{Interp}_{x,i+1}}[k] \cdot E^{K-k} \end{aligned} \tag{4.12}$$

By a change of index, it is therefore possible to deduce :

$$y[i] = y[i-1] \cdot E^{K} + (1-E) \cdot \sum_{k=1}^{K} O_{\text{Interp}_{x,i}}[k] \cdot E^{K-k} \tag{4.13}$$

This formulation meets the condition 4.10

Putting all together

Finally, the ASIC will implement the :

$$\mathrm{STNO}_{\mathrm{ASIC}}(x[i]) = \begin{cases} y[i] = \mathcal{I}_{\mathrm{ASIC}}[i] & \text{if } i \geq 0 \\ y[-1] = O_{\mathrm{Interp}_{x,0}}[1] & \text{if } i = \text{-1} \end{cases} \tag{4.14}$$

with :

$$\mathcal{I}_{\mathrm{ASIC}}[i] = y[i-1] \cdot E^K + (1-E) \cdot \sum_{k=1}^{K} O_{\mathrm{Interp}_{x,i}}[k] \cdot E^{K-k} \tag{4.15}$$

and with the computation of $O_{\mathrm{Interp}_{x,i}}[k]$ independent and thus parallelisable:

$$O_{\mathrm{Interp}_{x,i}}[k] = -\sqrt{\left| p_\Delta - p_I \cdot x[i-1] + (x[i-1] - x[i]) \cdot \frac{k \cdot p_I}{K} \right|} \tag{4.16}$$

4.3 The Interfaces

The objective of interfacing is to communicate data to the ASIC. Indeed, the reservoir computing gives an input vector $\mathbf{X}$ of size $\mathbb{R}^{1\times N_\theta N_\tau}$ and must receive an output $\mathbf{Y}$ of the same size.
To ensure this communication, several levels of interfacing are necessary between the software and the hardware:

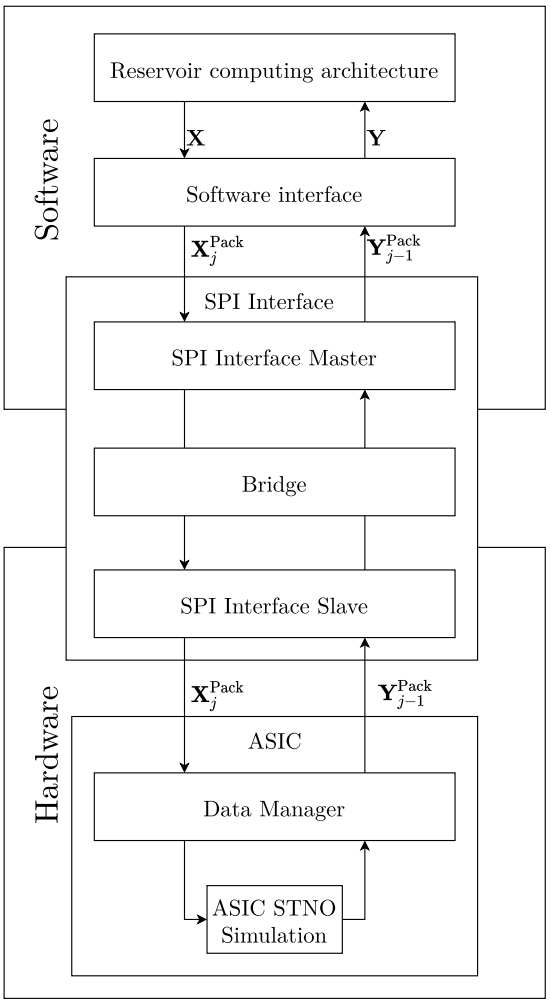

Figure 4.3: Global description of the data flow through the interfaces.

- Software Interface: It splits the $\mathbf{X}$ input into subsets. Indeed, it is not possible for the ASIC to process too large quantity of data.
- SPI Interface : It ensures communication at the material level. It is composed of three elements:
  - The master instance : It gives rhythm to communication.
  - The bridge : It ensures the physical communication of data.
  - The slave instance : It answers the request of the master.

  Each of the instances is composed of a sender and a receiver.
- ASIC : It processes the packets and simulates the STNO on each scalar of the packets.

Once the output value has been calculated, the whole communication path is then carried out in reverse.

Note that simultaneously, packets are received and sent. However, as the calculation of $\mathbf{Y}_j^{\text{Pack}}$ is not instantaneous for an input $\mathbf{X}_j^{\text{Pack}}$, its predecessor $\mathbf{Y}_{j-1}^{\text{Pack}}$ is sent.

4.3.1 The software interface

This interface, illustrated in Fig. 4.4 ensures that the data is sent in a packet format that can be handled by the ASIC.

Indeed, on the one hand the data has to be converted to ByteArray format to be readable by the ASIC.

On the other hand, since the ASIC memory is limited, it is not possible to send all the data at once. Sending the scalars one by one is not practical because there is a fixed time delay for each communication. Thus, data packets are created of the largest size P that can be stored in the ASIC. They are denoted $\mathbf{X}_j^{\text{Pack}}$ (respectively $\mathbf{Y}_j^{\text{Pack}}$) such that :

$$\mathbf{X}_j^{\text{Pack}} = \{x[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \text{ and } \mathbf{Y}_j^{\text{Pack}} = \{y[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \tag{4.17}$$

The complete matrix $\mathbf{X}$ (respectively $\mathbf{Y}$) is then :

$$\mathbf{X} = \left\{ \mathbf{X}_j^{\text{Pack}} \text{ for } j \in \left[0, \frac{N_\theta N_\tau}{P}\right] \right\} \text{ and } \mathbf{Y} = \left\{ \mathbf{Y}_j^{\text{Pack}} \text{ for } j \in \left[0, \frac{N_\theta N_\tau}{P}\right] \right\} \tag{4.18}$$

To indicate that a new waveform $\mathbf{X}$ must be processed, the variable RST_X is created and is active when $j = 0$.

Working

To ensure all this processing, the software interface is decomposed into :

- Sender : It converts the data into packets that can be sent directly to the ASIC. To do this, two processes are performed on the data :
  - Packet Folding : It ensures the construction of $\mathbf{X}^{\text{Pack}}$ packets and the creation of the RST_X signal
  - `float32` to ByteArray converter : It converts the data from vector of `float32` to byteArray.
- Receiver : This part ensures the complementary operations to the sender. It is therefore composed of a "Byte to `float32` converter" and a "Packet recontructor".

In practice

In practice the length of $\mathbf{X}$ is $N_\theta N_\tau$. It is known that $N_\theta = 400$ and $N_\tau \in [27; 67]$, so $\mathbf{X}$ is of length $[10400; 26800]$. In order to send packets that are always the same size, they must be of size that divides 400. The size chosen is P=100, the largest packet length that can be stored in the FPGA Cyclone IV.

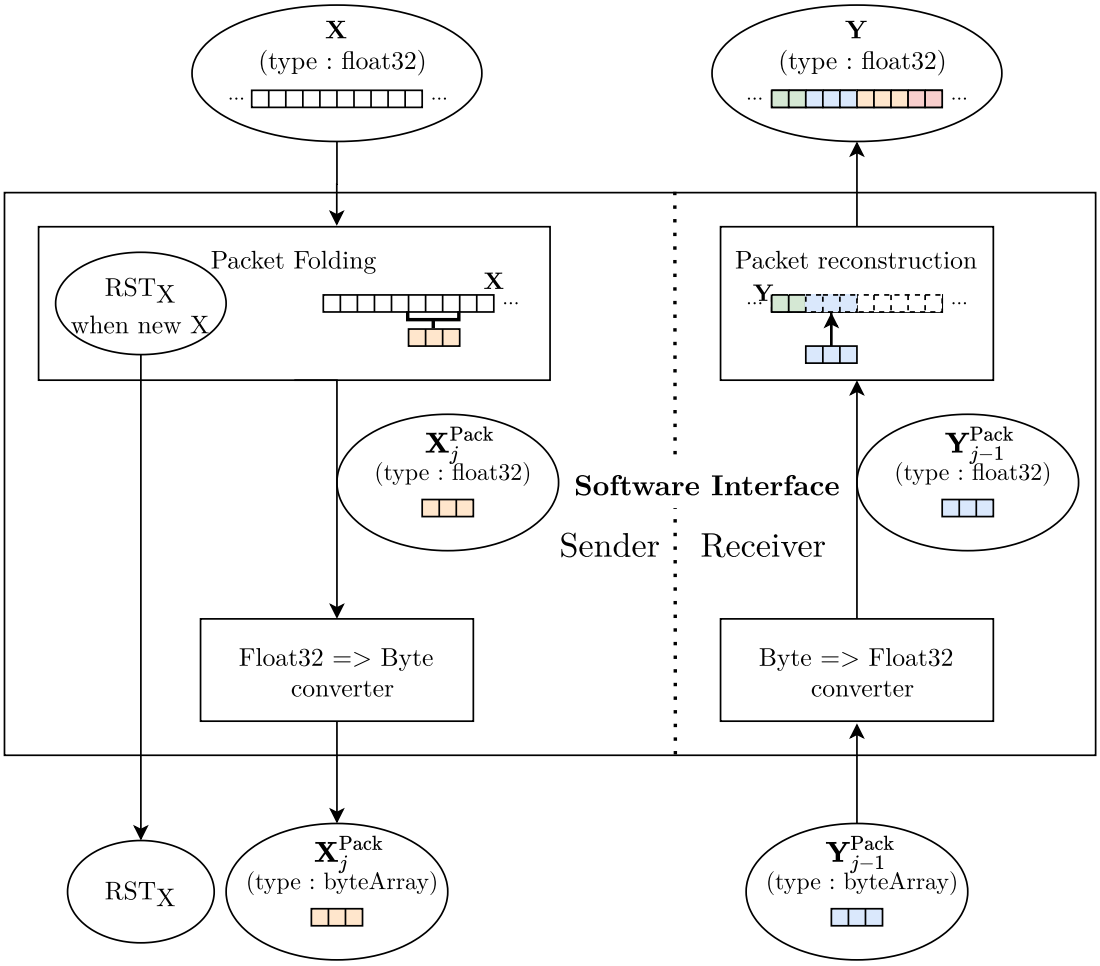

Figure 4.4: Work-flow of the data in the software interface.

4.3.2 The serial peripheral interface (SPI)

The serial peripheral interface ensures the material-level communication between the software and the hardware.

Working

This interface, illustrated in Fig. 4.5, is composed of two modules :

- The master instance: It sets a rhythm for communication for two reasons :
  - Synchronous clock : The two devices (embedded computer and the hardware accelerator) are assynchronous. This means that they are not cadenced by the same clock. The master therefore imposes its clock to the slave through SCLK.
  - Master requests : To avoid the master listening to the slave all the time, the master only listens when it sends a request. To signify a request it sets chip select to 0 (active-low).
- The slave instance : It is rythmed on the master and responds to its requests.

Each of these modules consists of a sender and a receiver.

- Sender : The sender transforms the bytearray to be sent into a serial bit stream. The sender of the master and slave sends this stream via the MOSI (Master Out Slave In) and MISO (Master In Slave Out) respectively.
- Receiver : It receives the data stream and reassembles it into a bytearrray.

Note that based on the CS, it is possible to determine the start and end of the stream. Thus, the slave's receiver generates the P_E signal when the stream is complete and the data ready for use.

In practice

In practice, packets of 100 `float32` scalars are communicated. To this must be added RST_X encoded on 8 bits. There are thus 3208 bits to transmit in total. The SPI communication is limited to 9.25 MHz. So the transmission takes $3208 \text{ bits}/9,25 \text{ MHz} = 350 \ \mu\text{s}$.

The $\mathbf{Y}_j^{\text{Pack}}$ must thus be compute during the transfer of $\mathbf{Y}_{j-1}^{\text{Pack}}$ and has to be ready for the next transfer. As so, its processing must be faster than a communication. As it will be seen later, the $\mathbf{Y}_j^{\text{Pack}}$ calculation is done 82.2 μs. The packet is therefore ready well before the next send.

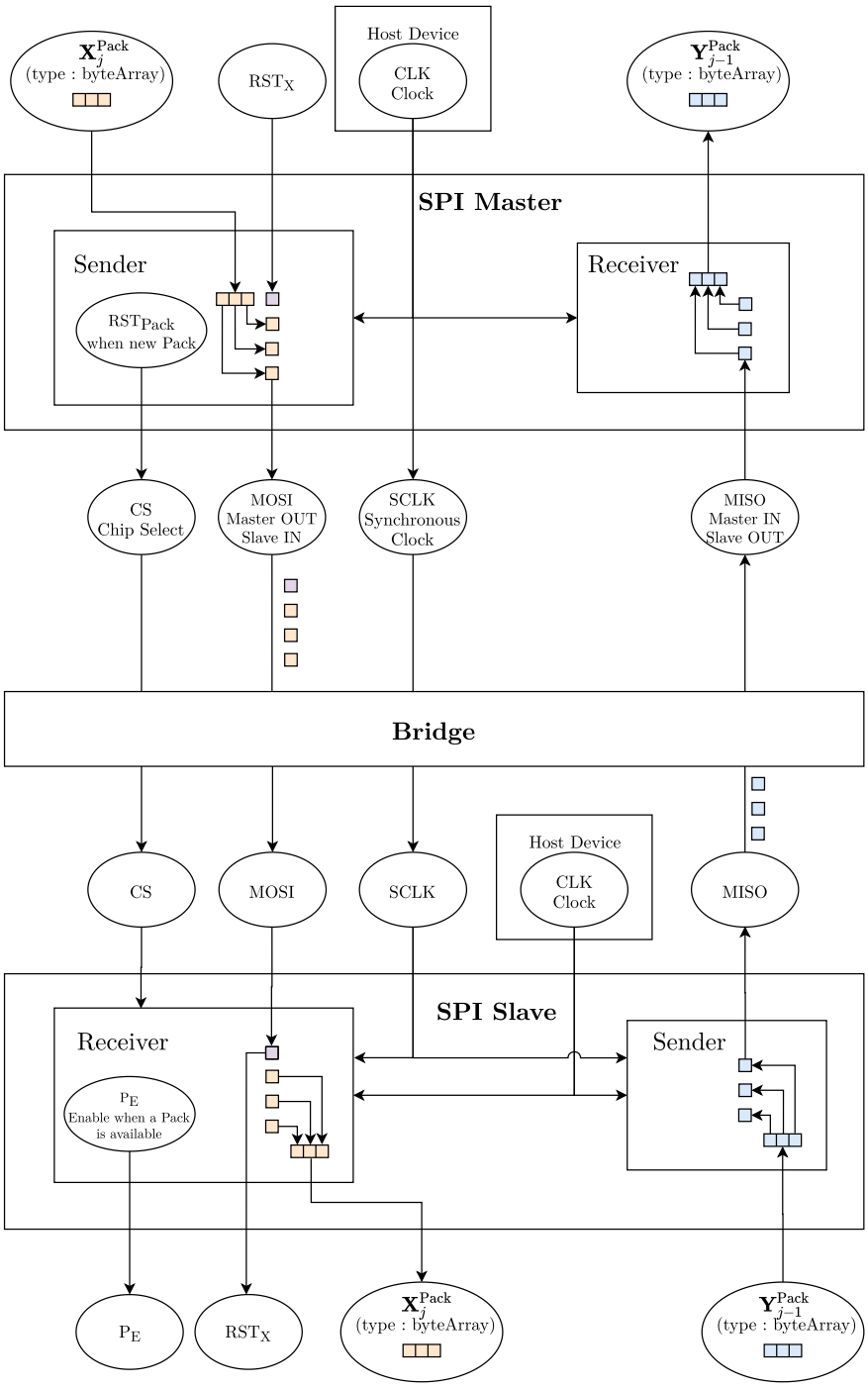

Figure 4.5: Work-flow of the data in the SPI interface.

4.4 Data Manager

Once the packets have arrived, they must be processed. In order to do this, the data must first be prepared. This section discusses the modules that have been implemented for this purpose.

The data to be output are:

$$\begin{aligned}\mathbf{Y}_j^{\text{Pack}} &= \{y[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \\ &= \{\text{STNO}_{\text{ASIC}}(x[i]) \text{ for } i \in [j \cdot P, (j+1) \cdot P[\}\end{aligned} \tag{4.19}$$

Note that $\text{STNO}_{\text{ASIC}}(x[i])$ actually needs other arguments that are not displayed for readability reasons. Its true form is :

$$\text{STNO}_{\text{ASIC}}(x[i], x[i-1], y[i-1], (i=0)) \tag{4.20}$$

With (i=0) the variable describing whether i is null or not. Thus, the Eq.4.19 can be rewritten :

$$\begin{aligned}\mathbf{Y}_j^{\text{Pack}} &= \{y[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \\ &= \{\text{STNO}_{\text{ASIC}}(x[i], x[i-1], y[i-1], (i=0)) \text{ for } i \in [j \cdot P, (j+1) \cdot P[\}\end{aligned} \tag{4.21}$$

If one defines the function STNO_{Mat} able to apply the function $\text{STNO}_{\text{ASIC}}$ to all the elements of a vector such that :

$$\text{STNO}_{\text{Mat}}(\mathbf{X}) = \{\text{STNO}_{\text{ASIC}}(x) \text{ for } x \in \mathbf{X}\} \tag{4.22}$$

This can be summarised as follows:

$$\mathbf{Y}_j^{\text{Pack}} = \text{STNO}_{\text{Mat}}(\mathbf{X}_j^{\text{Pack}}, x[j \cdot P - 1], y[j \cdot P - 1], (j=0)) \tag{4.23}$$

This section covers the implementation of :

- The Global Manager: It ensures the preparation of data useful for STNO_{Mat}.
- The FirstOut Manager: It provides the calculation of $y[-1]$ to calculate $y[0]$.
- The Computation Manager: it ensures the calculation of STNO_{Mat}.

In practice

Each of the modules is implemented on a DE0-Nano with a Cyclone IV FPGA. An analysis of the signals is discussed for each module. These `float32` signals are displayed in hexadecimal format.
A simulation of a packet of length $P = 2$ has been simulated for the sake of clarity in the explanations.

4.4.1 Global Manager

The global Manager, presented in Fig. 4.6, is in charge of managing the incoming data from the SPI interface. It also has to prepare the data useful for STNO_{Mat}.

I/O

The following are the inputs and outputs of the Global Manager:

- Input : This module has the following signals as input:
  - $\mathbf{X}_j^{\text{Pack}}$: The data packet to be processed.
  - P_E: The signal indicating when the $\mathbf{X}_j^{\text{Pack}}$ packet is ready to be used.
  - RST_X: It indicates when j is null. It is equivalent to the variable (j = 0)
- Output : It has as output $\mathbf{Y}_j^{\text{Pack}}$..

Working

The Global Manager ensures the preparation of data useful to STNO_{Mat} in the following calculation:

$$\mathbf{Y}_j^{\text{Pack}} = \text{STNO}_{\text{Mat}}(\mathbf{X}_j^{\text{Pack}}, x[j \cdot P - 1], y[j \cdot P - 1], (j = 0)) \tag{4.24}$$

The module therefore prepares the following data:

- $\mathbf{X}_j^{\text{Pack}}$: It simply forwards the signal
- $x[j * P - 1]$: This signal is not included in $\mathbf{X}_j^{\text{Pack}}$. It is equivalent to :

$$x[j \cdot P - 1] = \begin{cases} \mathbf{X}_{j-1}^{\text{Pack}}[P] & \text{if } j \neq 0 \\ x[-1] = 0 & \text{if } j = 0 \end{cases} \tag{4.25}$$

  This signal is therefore either generated or retained from the previous $\mathbf{X}_{j-1}^{\text{Pack}}$.

- $y[j \cdot P - 1]$: This signal is equivalent to :

$$y[j \cdot P - 1] = \begin{cases} \mathbf{Y}_{j-1}^{\text{Pack}}[P] & \text{if } j \neq 0 \\ y[-1] = O_{\text{Interp}_{x,0}}[1] & \text{if } j = 0 \end{cases} \tag{4.26}$$

  This signal is therefore either generated or retained from the previous $\mathbf{Y}_{j-1}^{\text{Pack}}$.

- $(j = 0)$: This signal is simply forwarded from RST_X

Submodules

The Global Manager calls and controls the following two modules:

- First Out Manager launched by $\mathrm{RST_{FO}}$ (Reset First Out): This is triggered when RST_X and P_E are active. Indeed it means respectively that $(j = 0)$ and that this information is true. This signal allows to launch the calculation of $y[-1]$ on the basis of the first input of the first module $\mathbf{X}_0^{\mathrm{Pack}}[0] = x[0]$.
- Computation Manager: launched by $\mathrm{RST_{CM}}$ (Reset Computation Manager): This signal means the start of the calculation of $\mathrm{STNO_{Mat}}$ ensured by the Computation Manager. This calculation can only be launched if one of these two conditions is met:
  - $(t - t_{\mathrm{RST_{FO}}}) > \tau_{FO}$: This condition means that the module can only be executed when the firstOut module has finished processing $y[-1]$
  - P_E : This means that it can only be launched when new data is available

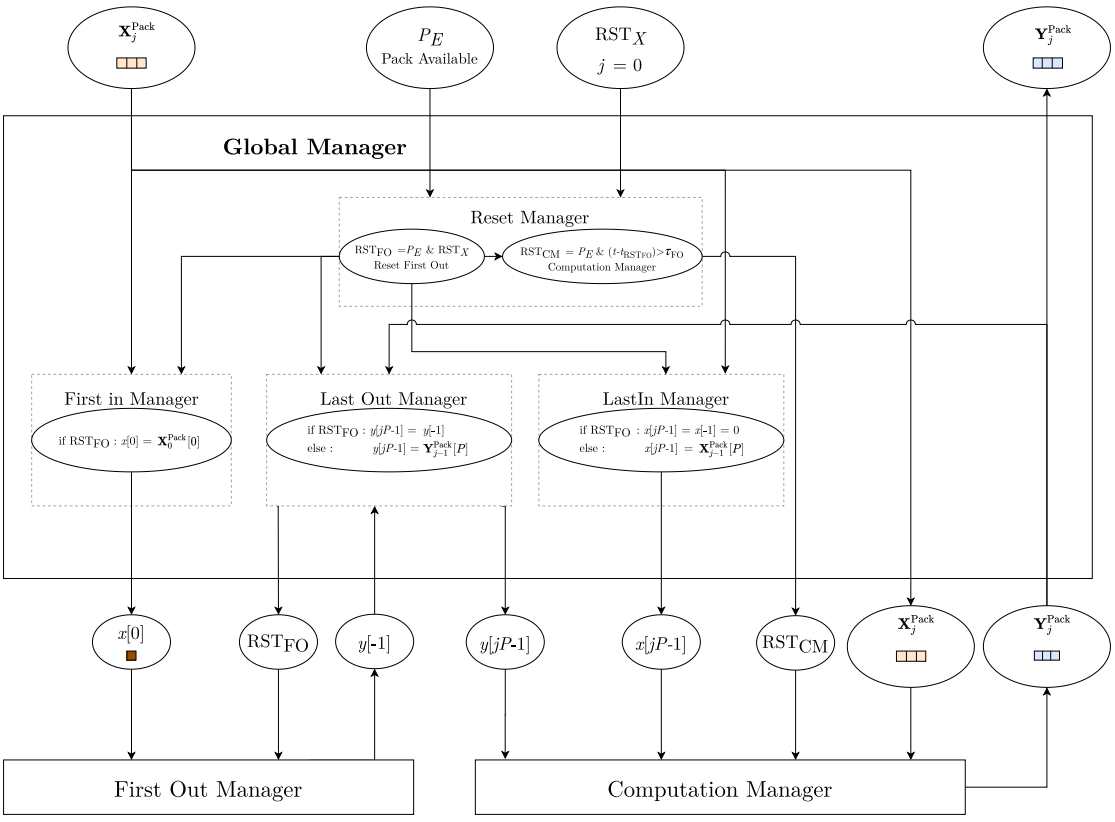

Figure 4.6: Work-flow of the data in the Global Manager.

In practice

The following signals interact with the Global Manager

- Figure 4.7 shows the case where the first packet arrives (e.g. $\mathbf{X}_j^{\text{Pack}} =$ `c0f9a4ecbe4d81a6`). The RST_X is thus activated and the RST_{FO} is triggered. $x[-1]$ and $y[-1]$ are sent to the First Out Manager. It can be seen that RST_{CM} is only triggered after τ_{FO}.

Figure 4.7: Signal analysis for the Global Manager in the case where the first packet arrives.

- Figure 4.8 shows the same case as Fig. 4.7 but with a larger time scale. It can be seen that after the activation of RST_{CM} the computation manager takes some time before returning a partial and then a complete version of $\mathbf{Y}^{\text{Pack}}$.

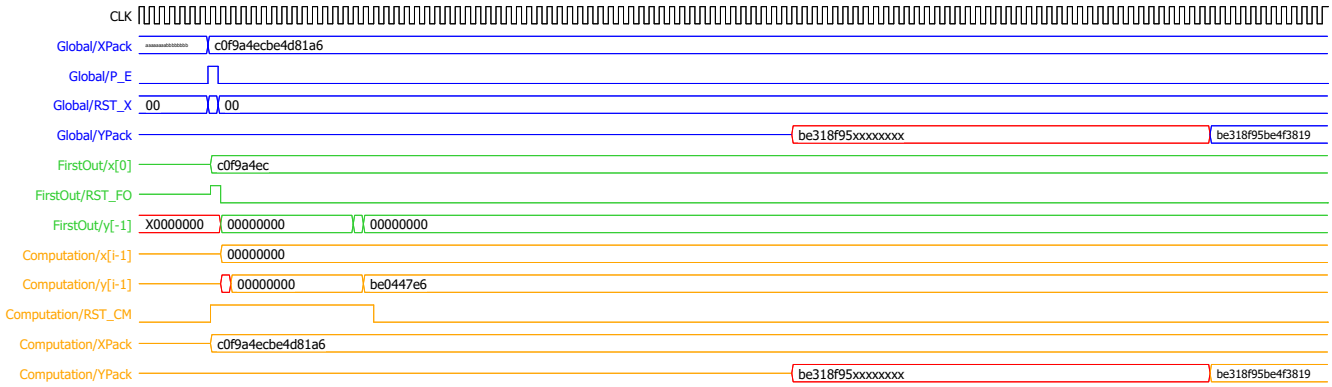

Figure 4.8: Signal analysis for the Global Manager in the case where the first packet arrives in a larger time-scale.

- The Fig. 4.9 shows the case where it is not the first $\mathbf{X}^{\text{Pack}}$ that arrives but a following one. It can be seen that RST_X and therefore RST_{FO} are not activated but that RST_{CM} is directly activated because P_E is on. Also, it can be observed that $\mathbf{Y}^{\text{Pack}}$ is first partially and then completely changed.

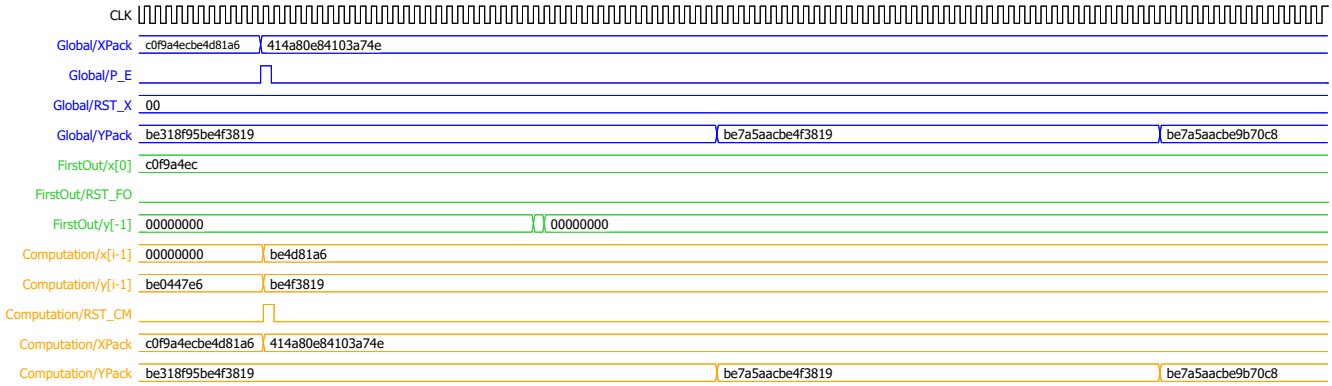

Figure 4.9: Signal analysis for the Global Manager in the case where the next packet arrives.

4.4.2 First Out Manager

The First Out Manager, shown in Fig. 4.10, is in charge of compute $y[-1]$.

I/O

Here are the Inputs/Outputs of the FirstOut Manager :

- Input : This module has as input the following signals :
  - $x[0]$: This is the first entry in the first $x[0] = \mathbf{X}_0^{\text{Pack}}[0]$
  - RST_{FO} : This is the signal indicating when this module can be launched.
- Output : It has as output $y[-1]$ after a certain processing time τ_{FO}.

Working

This module has to calculate $y[-1]$ such that:

$$y[-1] = O_{\text{Interp}_{x,0}}[1] \tag{4.27}$$

To this end, it prepares the data $O_{\text{Interp}_{x,0}}[1]$ needs, i.e.:

- $x[-1]$: As this signal is generated for interpolation, it is null.
- $x[0]$: This signal is directly forwarded from the inputs.

Submodules

It interacts with the Interpolate module launched with the RST_I (Reset Interpolate) signal forwarded by RST_{FO}. Interpolate gives the results of the interpolation one after the other:

$$\{O_{\text{Interp}_{x,i}}[k] \text{ for } k \in [1, K]\} \tag{4.28}$$

The role of the FirstOut Manager is to isolate $y[-1] = O_{\text{Interp}_{x,0}}[1]$. Based on RST_I, it waits for the processing time of Interpolate τ_I and isolates the first output.

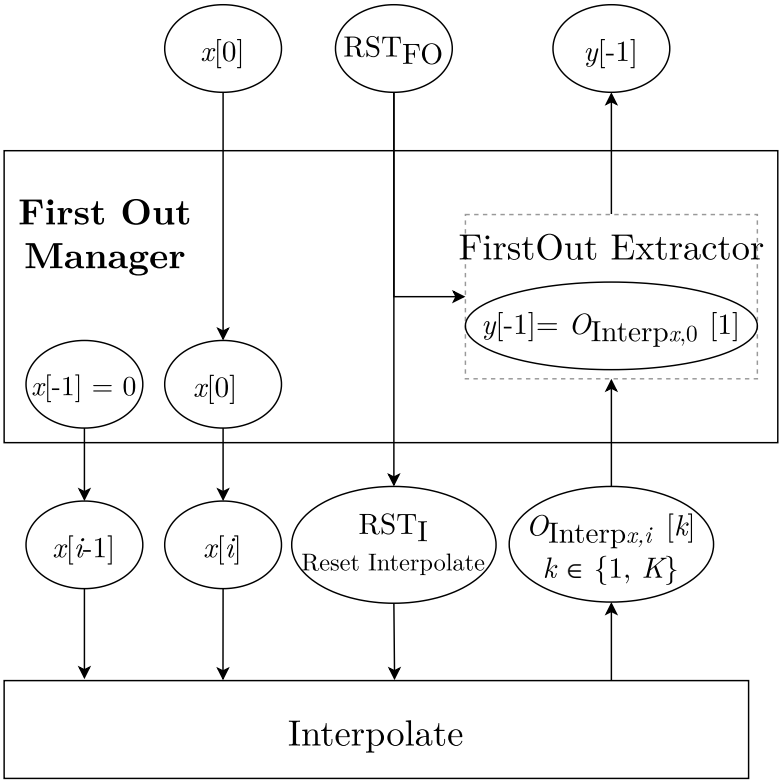

Figure 4.10: Work-flow of the data in the FirstOut Manager.

In practice

The following signals interact with the First Out Manager.

It can be seen that the $\text{RST}_{\text{FO}} = \text{RST}_I$. Also, the interpolate produces its outputs after a certain delay and FirstOut only isolates the first one (e.g. $y[-1] =$ `be0447e6`).

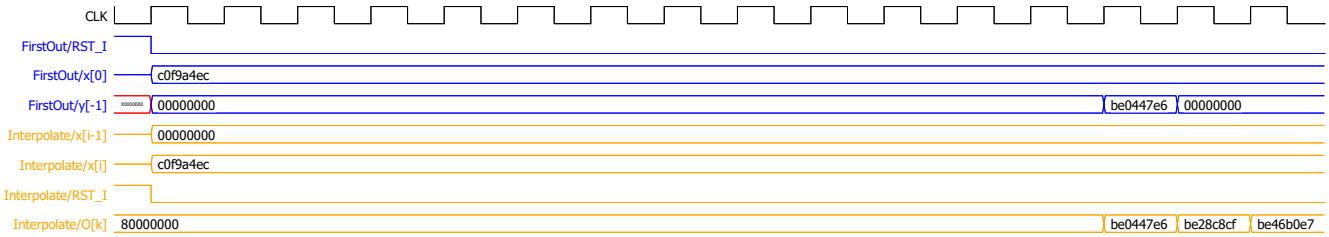

Figure 4.11: Signal analysis for the FirstOut Manager.

4.4.3 Computation Manager

The Computation Manager, shown in Fig. 4.12, manage the computation of $\mathbf{Y}^{\text{Pack}}$ such that :

$$\mathbf{Y}_j^{\text{Pack}} = \text{STNO}_{\text{Mat}}(\mathbf{X}_j^{\text{Pack}}, x[j \cdot P - 1], y[j \cdot P - 1], (j = 0)) \tag{4.29}$$

I/O

Here are the Inputs/Outputs of the Computation Manager :

- Input : This module has as input the following signals :
  - $y[j \cdot P - 1]$: The last output of the previous calculation.
  - $x[j \cdot P - 1]$: The last entry in the previous calculation.
  - $\mathbf{X}_j^{\text{Pack}}$: The data packet to be processed.
  - RST_{CM} : The signal indicating when this module can start.
- Output : It has as output the packet $\mathbf{Y}_j^{\text{Pack}}$.

Working

This module must calculate $\mathbf{Y}^{\text{Pack}}$ such that

$$\begin{aligned}\mathbf{Y}_j^{\text{Pack}} &= \{y[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \\ &= \{\text{STNO}_{\text{ASIC}}(x[i], x[i-1], y[i-1], (i = 0)) \text{ for } i \in [j \cdot P, (j+1) \cdot P[\}\end{aligned} \tag{4.30}$$

Thus, it proceeds in several steps:

1. $\mathbf{X}_j^{\text{Pack}}$ padding : The previous entry is pad in the beginning. Indeed, $x[j \cdot P - 1]$ is not included in $\mathbf{X}_j^{\text{Pack}}$ but is necessary for the calculation of $\mathbf{Y}_j^{\text{Pack}}$. :

$$\begin{aligned}\mathbf{X}_{j,Pad}^{\text{Pack}} &= \{x[j \cdot P - 1], \mathbf{X}_j^{\text{Pack}}\} \\ &= \{x[i] \text{ for } i \in [j \cdot P - 1, (j+1) \cdot P[\}\end{aligned} \tag{4.31}$$

2. $\mathbf{X}_{j,Pad}^{\text{Pack}}$ unfolding : it extracts the elements $x[i]$ one by one from $\mathbf{X}_{j,Pad}^{\text{Pack}}$ according to the Eq.4.31.

3. Transformation : It extracts for each input $x[i]$ its output $y[i]$ such that :

$$y[i] = \text{STNO}_{\text{ASIC}}(x[i], x[i-1], y[i-1], (i = 0)) \tag{4.32}$$

   Note that (i=0) is derived from RST_{CM} which means that a new $\mathbf{X}^{\text{Pack}}$ packet must be processed. In other words, that i is null.

4. $\mathbf{Y}_j^{\text{Pack}}$ folding : It reconstructs the output packet $\mathbf{Y}_j^{\text{Pack}}$ such that :

$$\mathbf{Y}_j^{\text{Pack}} = \{y[i] \text{ for } i \in [j \cdot P, (j+1) \cdot P[\} \tag{4.33}$$

Submodules

The Computation Manager uses the Integrate module which calculates :

$$y[i] = \text{STNO}_{\text{ASIC}}(x[i], x[i-1], y[i-1], (i == 0)) \tag{4.34}$$

To start Integrate a RST_{Int} signal is generated. It ensures that Integrate only starts calculating when all the data is ready. Also RST_{Int} is activated so that a calculation is not started until the previous one is finished. To this end, it is active under at least one of these conditions:

- RST_{CM} : This means that the Integrate will have to process the first entry in a packet. It has not yet calculated so there is no risk of stopping an operation in progress.
- $(t - t_{\text{RST}_{\text{CM}}})\%\tau_{\text{Int}} = 0$: The Integrate module can only start when the previous calculation has been executed and is finished. (Note that % stands for the operation modulo.)

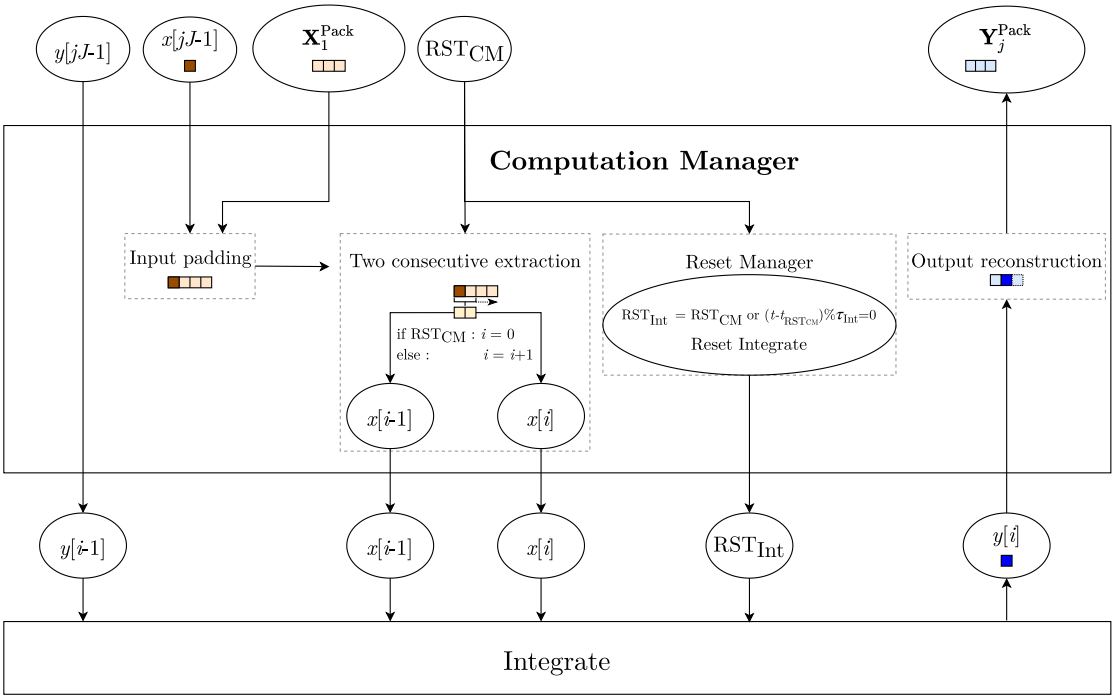

Figure 4.12: Work-flow of the data in the Computation Manager.

In practice

The following signals interact with the Computation Manager :

- Figure 4.13 shows the interaction between the Computation Manager module and the Integrate module. It illustrates the case where $\mathrm{RST}_{\mathrm{CM}}$ is activated due to $(j = 0)$, i.e. when a new $\mathbf{X}$ must be processed. It is structured in several steps.

  1. $\mathbf{X}_j^{\mathrm{Pack}}$ padding : First, one can see that the vector $\mathbf{X}_{0,Pad}^{\mathrm{Pack}}$ is indeed padded with $x[jP-1] = x[-1] = 0$.
  2. $\mathbf{X}_{j,Pad}^{\mathrm{Pack}}$ unfolding : The vector is decomposed into $x[i]$ and the rest of the data is also prepared. As soon as all the data is ready, the calculation is started for Integrate via the signal $\mathrm{RST}_{\mathrm{Int}}$.
  3. Transformation : Integrate takes some time τ_{Int} to process and then outputs a value $y[i]$ (e.g. $y[i] =$ `be318f95`).
  4. $\mathbf{Y}_j^{\mathrm{Pack}}$ folding : The vector is partially filled. As the integrate process is finished (i.e. $(t - t_{\mathrm{RST}_{\mathrm{CM}}})\%\tau_{\mathrm{Int}} = 0$), a $\mathrm{RST}_{\mathrm{Int}}$ is reactivated and the loop is repeated until $\mathbf{Y}_0^{\mathrm{Pack}}$ is completed.

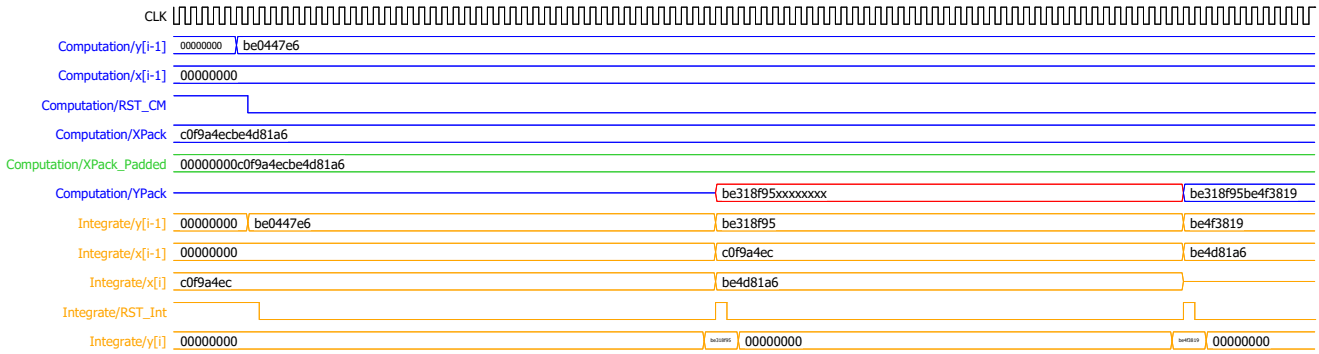

Figure 4.13: Signal analysis for the Computation Manager in the case where the first packet arrives.

- Figure 4.14 also shows the interaction between the Computation Manager and Integrate modules. However, it illustrates a case where $j \neq 0$. Thus, it can be seen for step 4. that $\mathbf{Y}_j^{\text{Pack}}$ is first partially and then completely changed.

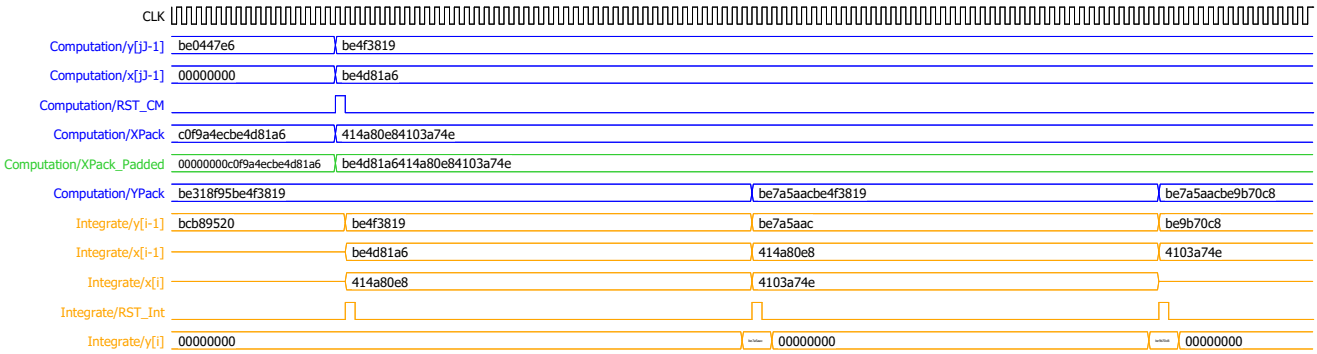

Figure 4.14: Signal analysis for the Computation Manager in the case where the next packet arrives.

4.5 Computation Modules

As a reminder, the $\text{STNO}_{\text{ASIC}}$ function is described as follows:

$$\text{STNO}_{\text{ASIC}}(x[i]) = \begin{cases} y[i] = \mathcal{I}_{\text{ASIC}}[i] & \text{if } i \geq 0 \\ y[-1] = O_{\text{Interp}_{x,0}}[1] & \text{if } i = \text{-1} \end{cases} \tag{4.35}$$

While all data and case management is carried out by the Data Manager, it remains to implement the functions $\mathcal{I}_{\text{ASIC}}$ and $O_{\text{Interp}_{x,i}}[k]$. It is the implementation of these functions that will be discussed in this section:

- The Integrate module covers the implementation of $\mathcal{I}_{\text{ASIC}}$.
- The Interpolate module covers the implementation of $O_{\text{Interp}_{x,i}}[k]$.

In practice

For these two functions, simple operations are required.Their implementation is achieved through the floating point IP Core from Intel:

- `altfp_abs` : The absolute value of a `float32` in $\tau_{\text{abs}} = 0$.
- `altfp_add_sub` : The addition/subtraction of two floats in $\tau_{\text{add/sub}} = 2$.
- `altfp_mult` : The multiplication of two `float32` in $\tau_{\text{mult}} = 3$.
- `altfp_sqrt` : The square root of a `float32` in $\tau_{\text{sqrt}} = 6$.
- `fp_acc_custum` : The accumulation of `float32` after $\tau_{+\text{acc}} = 7$ and $\tau_{\text{Pipeline}} = 1$ no matter how many floats to accumulate.
- `fp_opp` : The opposite of a `float32` in $\tau_{\text{opp}} = 0$.

4.5.1 Integrate module

The Integration module, shown in Fig. 4.15, is the module calculating the function:

$$y[i] = \mathcal{I}_{\text{ASIC}}[i] = \mathcal{I}_{\text{ASIC}}(y[i-1], x[i-1], x[i]) \tag{4.36}$$

I/O

Here are the Inputs/Outputs of the Integrate module:

- Input : This module has the following signals as input:
  - $y[i-1]$, $x[i-1]$, $x[i]$: Scalars representing the previous output and inputs respectively.
  - RST_{Int} : This signal indicates when the module can start.
- Output : It has as output the value $y[i]$.

Working

This module has to calculate $y[i]$ such that:

$$y[i] = \mathcal{I}_{\text{ASIC}}[i] = y[i-1] \cdot E^K + \sum_{k=1}^{K} (1-E) \cdot O_{\text{Interp}_{x,i}}[k] \cdot E^{K-k} \tag{4.37}$$

To do this, it proceeds in several steps:

1. $O_{\text{Interp}_{x,i}}[k]$: First, the Interpolation module has to be launched. This returns after a certain delay τ_{Interp} outputs at each clock cycle.
2. $y[i-1] * E^K$: While the interpolation module is processing, the calculation of this term can already be started. Note that $y[i-1]$ is an input while E^K is a fixed value which is therefore hardcoded.
3. $\text{FactE}[k] = (1-E) * E^{K-k}$: The process of $\text{factE}_{\text{generator}}$ is also launched, which is a module generating one of the FactEs at each clock cycle.
4. AccTerm : The terms are summed in an accumulator outAcc. such that :

$$\text{OutAcc}[k] = y_{\text{Interp}_{x,i}}[k] = \sum_{k'=0}^{k} \text{AccTerm}[k'] \tag{4.38}$$

   with :

$$\text{AccTerm}_i[k'] = \begin{cases} y[i-1] * E^K & \text{if } k' = 0 \\ \text{FactE}[k] * O_{\text{Interp}_{x,i}}[k'] & \text{else} \end{cases} \tag{4.39}$$

5. Out : $y[i] = y_{\text{Interp}_{x,i}}[K] = \text{OutAcc}[K]$: The output of the module is ready when $k = K$.

Submodules

Two modules are required for Integrate to work :

- Interpolate: This module is in charge of calculating $O_{\text{Interp}_{x,i}}[k]$. Being the module with the longest delay, it is first launched via RST_I.
- $\text{factE}_{\text{generator}}$: This module generates the FactE$[k]$. Since these values are always the same regardless of the inputs, $\text{factE}_{\text{generator}}$ is simply a module that sends hard-coded values to each cycle. It is launched at the same time as the arrival of the first Interpolate value, so that the k indices match.

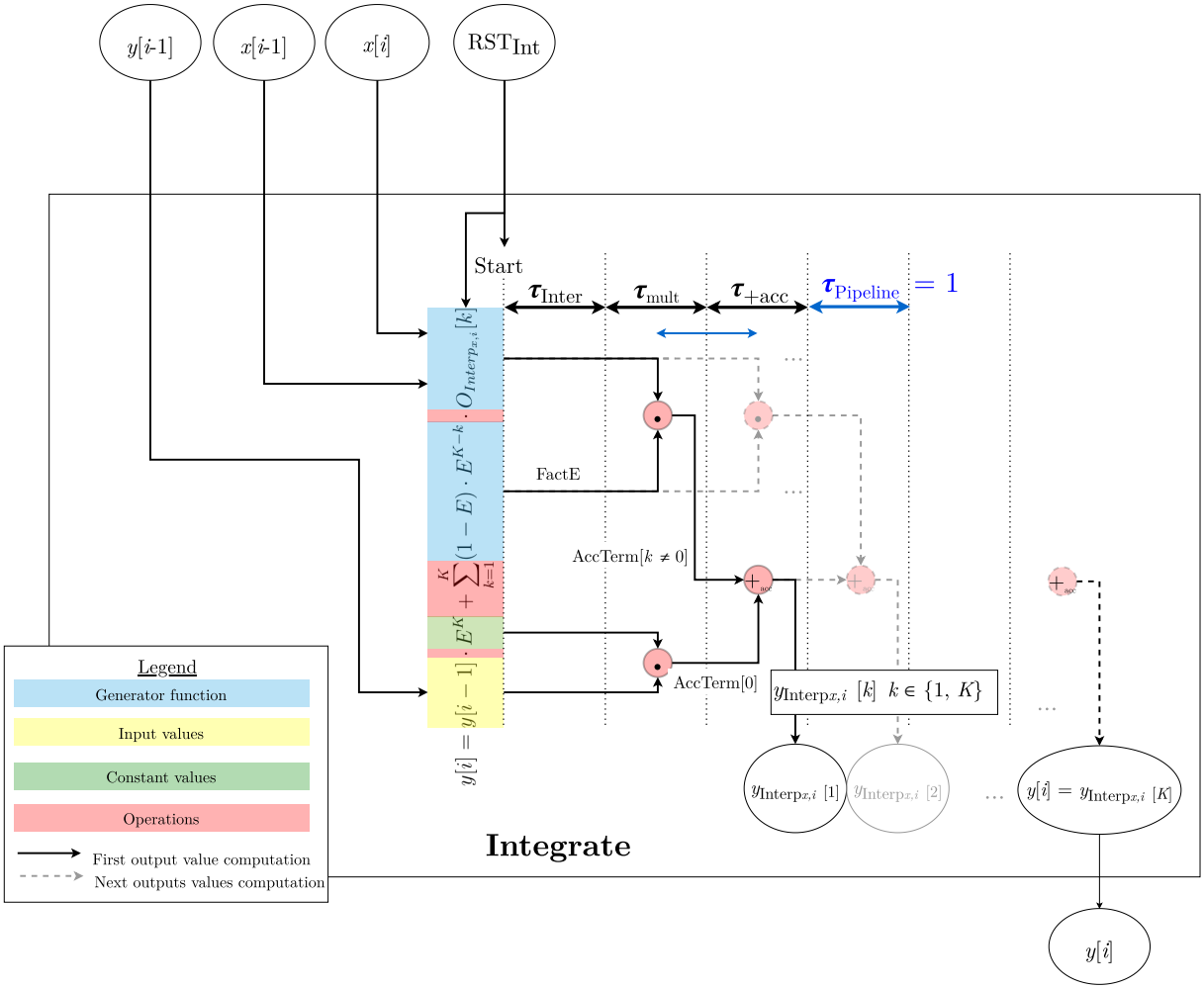

Figure 4.15: Work-flow of the data in the Integrate module.

In practice

Here are the signals interacting within the Integrate module. As a reminder, the operations "multiplication" and "accumulation" have respectively delays of $\tau_{\text{mult}} = 3$ and $\tau_{+\text{acc}} = 7$ clock cycle.

- Figure 4.16 shows the triggering after the activation of RST_{Int}. The following steps can be observed:

  1. $O_{\text{Interp}_{x,i}}[k]$: First the Interpolate module is launched. It starts to return output after a delay of $\tau_{\text{Interp}} = 13$ values, then at each clock cycle.
  2. $y[i-1] * E^K$: While the process interpolation module is running, the calculation of this term has already been started and is ready after $\tau_{mult} = 3$.
  3. $\text{FactE}[k] = (1-E) * E^{K-k}$: As soon as the values of $O_{\text{Interp}_{x,i}}[k]$ arrive, the $\text{factE}_{\text{generator}}$ is also lanched.
  4. AccTerm : The values are accumulated via the accTerm. Their accumulation is stored outAcc$[k]$ after $\tau_{+acc} = 7$ and is updated at each clock cycle.
  5. Out : $y[i] = y_{\text{Interp}_{x,i}}[K]$: Finally, as soon as $k = K$, the output is ready and returned.

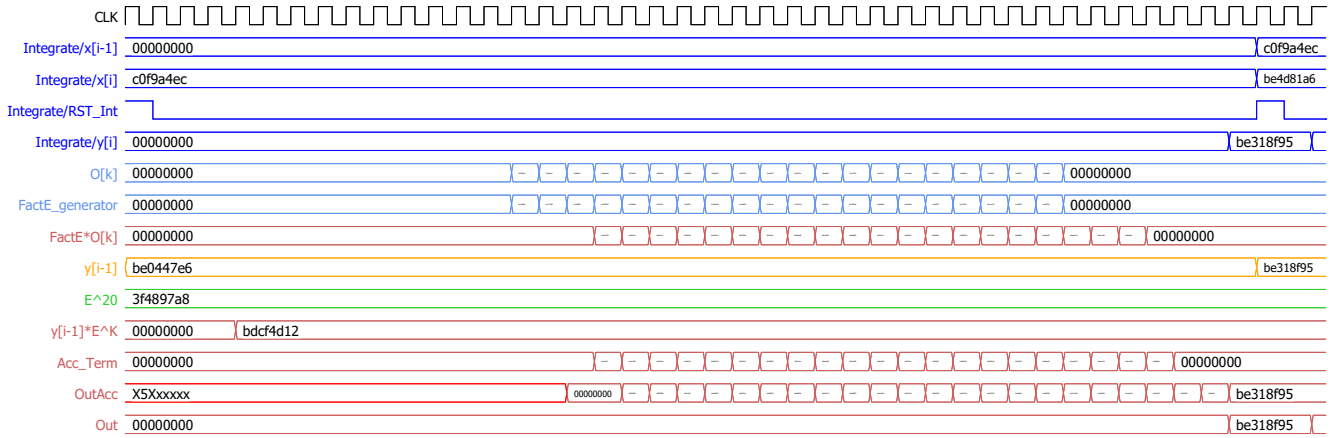

Figure 4.16: Signal analysis for the Integrate module showing one single integration.

- Figure 4.17 shows, as in Fig. 4.16 the dynamics of the signals but at a larger scale.

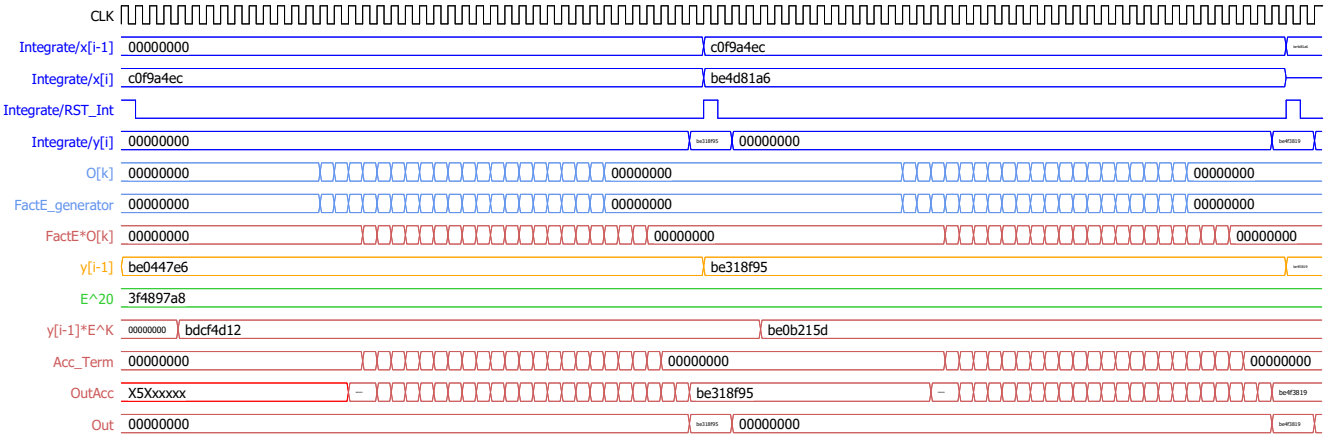

Figure 4.17: Signal analysis for the Integrate module at a larger time-scale showing multiple integration.

4.5.2 Interpolate module

The Interpolate module, shown in Fig. 4.18, is the module calculating the terms:

$$\{O_{\text{Interp}_{x,i}}(x[i-1], x[i], k) \text{ for } k \in [1, K]\} \tag{4.40}$$

I/O

Here are the Inputs/Outputs of the Interpolate module:

- Input: This module has the following signals as input:
  - $x[i-1]$, $x[i]$: Scalar inputs.
  - RST_I : This signal indicates when the module can start.
- Output : It has as output the packet. $\{O_{\text{Interp}_{x,i}}(x[i-1], x[i], k)\}$.

Working

This module has to calculate all the $O_{\text{Interp}_{x,i}}[k]$ such as:

$$O_{\text{Interp}_{x,i}}[k] = -\sqrt{\left| p_\Delta - p_I \cdot x[i-1] + (x[i-1] - x[i]) \cdot \frac{k \cdot p_I}{K} \right|} \tag{4.41}$$

To do this, it proceeds in several steps:

1. $\text{diffIn} = x[i-1] - x[i]$: First the difference of the inputs is calculated.
2. $\text{ramp}[k] = \frac{k \cdot p_I}{K}$: The ramp is generated by $\text{ramp}_{\text{generator}}$ which is a module that generates one of the values every clock cycle.
3. $ax = \text{diffIn} * \text{ramp}[k]$: The multiplication is calculated and stored.
4. $\text{pIx} = p_I * x[i-1]$: The hardcoded constant p_I and the input $x[i-1]$ are multiplied.
5. $b = p_\Delta - \text{pIx}$: The difference between p_Δ and pIx is calculated.
6. $c = ax + b$: ax et b are added together. Note that ax and b are calculated in parallel as they are not dependent on each other.
7. $-\sqrt{|c|}$: Finally the calculation is concluded with this last operation.

Submodules

The sub-module $\text{ramp}_{\text{generator}}$ is required. This module generates the ramp$[k]$. As these values are always the same, $\text{ramp}_{\text{generator}}$ is simply a module sending hardcoded values at each cycle. It is launched at the same time as the arrival of the diffIn values.

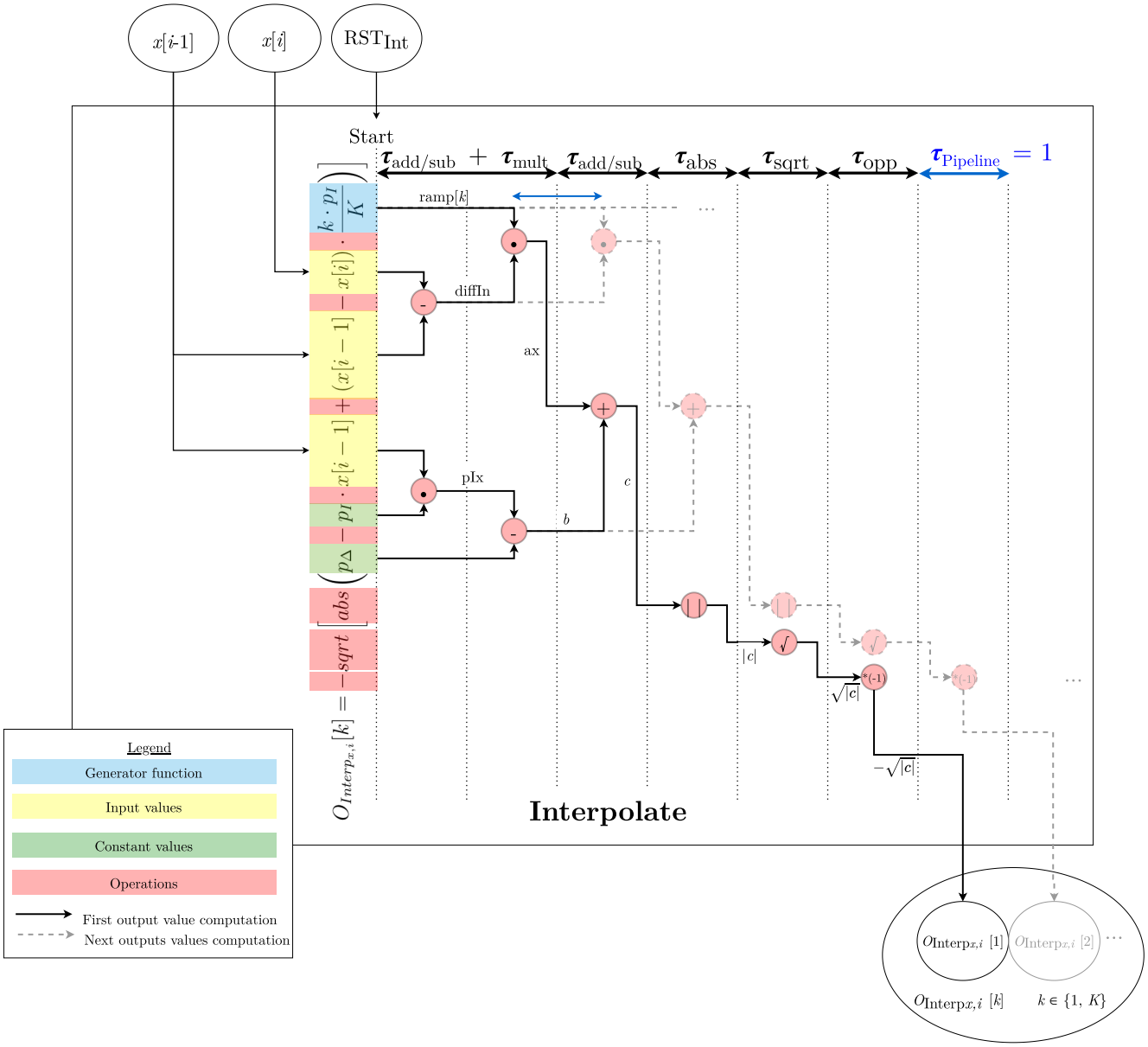

Figure 4.18: Work-flow of the data in the Interpolate module.

In practice

Here are the signals that interact within the Interpolate module. As a reminder, the operations "multiplication", "subtraction/addition", "absolute value", "opposite" and "square root" have respective delays of $\tau_{\text{mult}} = 3$, $\tau_{\text{add/sub}} = 2$, $\tau_{\text{abs}} = 0$, $\tau_{\text{opp}} = 0$ and $\tau_{\text{sqrt}} = 6$ clock cycle.

- Figure 4.16 shows the triggering after RST_I and the output of the first values. The following steps can be observed:

  1. $\text{diffIn} = x[i-1] - x[i]$: First the input difference is calculated after 2 cycles
  2. $\text{ramp}[k] = \frac{k \cdot p_I}{K}$: The ramp is generated by $\text{ramp}_{\text{generator}}$ as soon as diffIn is ready.
  3. $ax = \text{diffIn} * \text{ramp}[k]$: The multiplication is then performed and the results are given successively 3 clock cycles later.
  4. $\text{pIx} = p_I * x[i-1]$: During this time, the multiplication between the constant pI and the input $x[i-1]$ is processed and the result arrives 3 cycles later.
  5. $b = p_\Delta - \text{pIx}$: The difference between p_Δ and pIx is calculated in 2 cycles.
  6. $c = ax + b$: ax and b are added in 2 cycles. Note that ax and b are calculated in parallel and arrive exactly at the same time because their process takes both: $\tau_{\text{mult}} + \tau_{\text{add/sub}} = 5$
  7. $-\sqrt{|c|}$: Finally the calculation concludes with this last operation which takes $\tau_{\text{sqrt}} + \tau_{\text{abs}} + \tau_{\text{opp}} = 6$

  Then the other k outputs arrive successively one after the other at each clock cycle. i.e. $\tau_{\text{Pipeline}} = 1$. This clearly show that the module is pipelined properly.

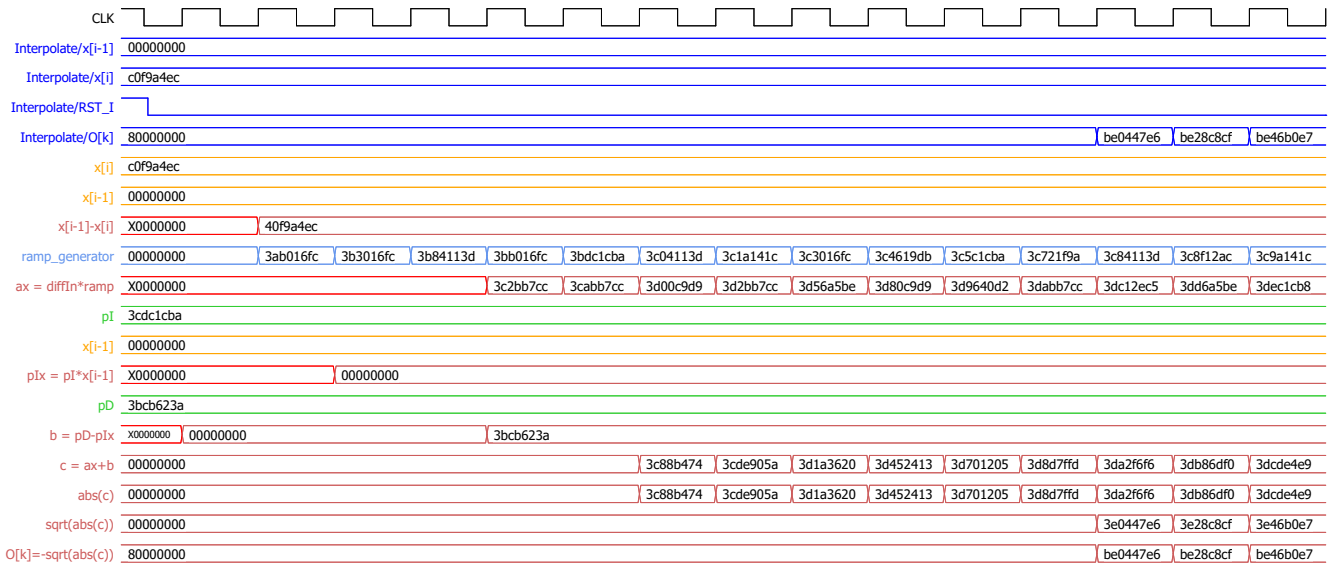

Figure 4.19: Signal analysis for the Interpolate module showing the first output processing.

- Figure 4.20 shows as the Fig. 4.19 the dynamics of the signals but on a larger scale. One notices that the $O_{\text{Interp}_{x,i}}$ are well given successively during the K=20 cycles.

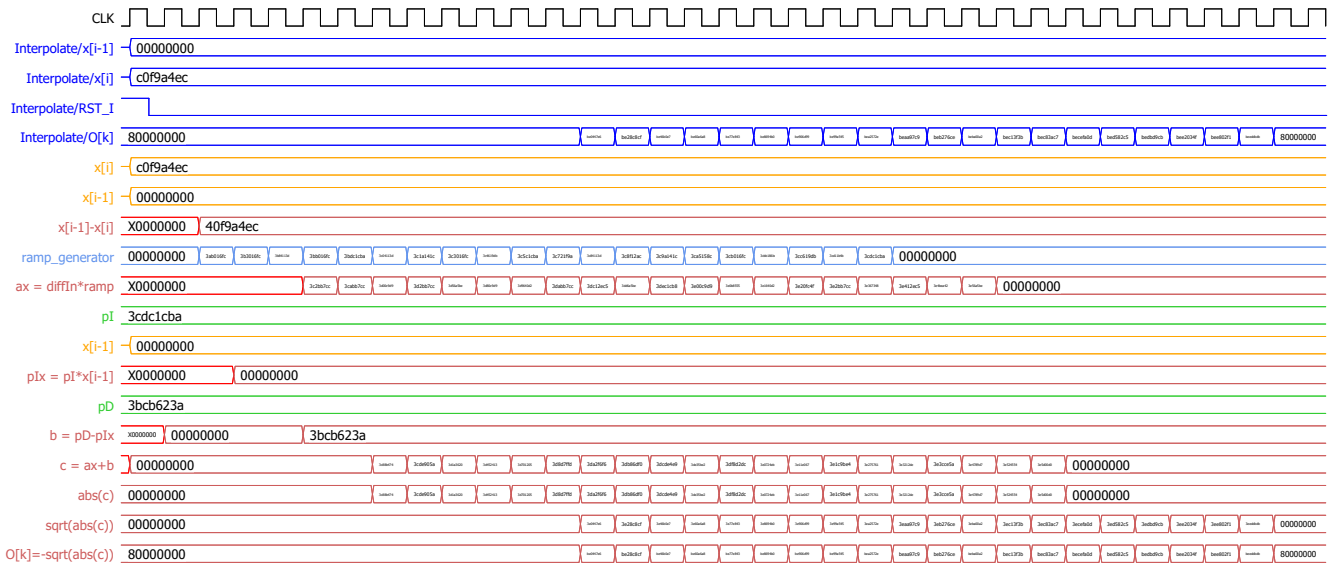

Figure 4.20: Signal analysis for the Interpolate module showing the 20 output processing.

4.6 In practice : Implementation

The complete assembly, shown in the Fig. 4.21, consists of three elements.

- The software including preprocessing and machine learning (i.e. Reservoir computing) is implemented in Python on a Raspberry Pi 4. [11]. It provides the following features:
  - CPU : Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz.
  - RAM : 8GB of LPDDR4-3200 SDRAM
- For the hardware part, it was prototyped on a DE0-Nano including a Cyclone IV FPGA [12]. It offers the following resources :
  - Clock frequency : 50 MHz
  - Total logic elements : 22320 (in use : 20705 (93%))
- The two devices are communicating through a bridge and using Serial Peripheral Interface (SPI). This interface is in practice working at 9,25 MHz, thus allowing the transfer of a package of 100 items in 350 μs. On average, a complete waveform can therefore be processed in 80 ms (using Cochlear transform, MFCC being twice of this time) which is reasonable for real time application (in the order of ten milliseconds).

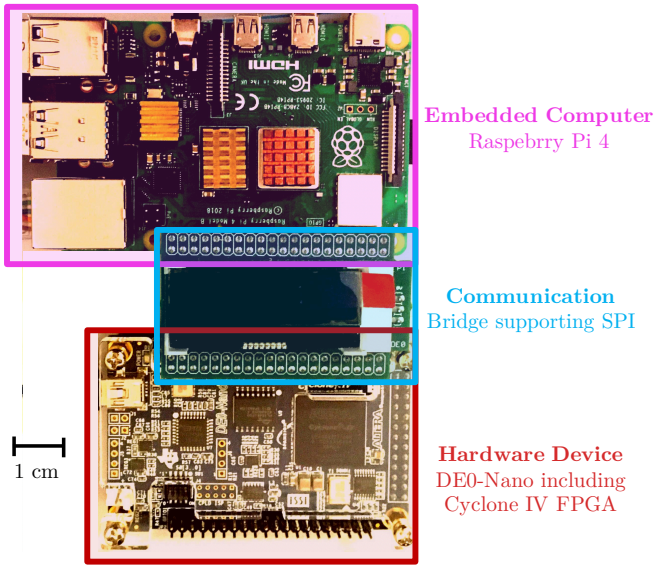

Figure 4.21: Picture of the real prototyping platform.

The total process delay for one input is 41 clock cycles long. Thus, the total calculation for a packet of $P = 100$ inputs is 4100 clock cycles. Added to this the processing time of the packet on arrival that is taking 1 clock cycle. The processing time of a packet is therefore 4101 clock cycles in total. Clocked at 50 MHz, this gives 82.2 μs.

Remarkably, the process time of is much less than the SPI communication time which is the limiting factor of this system (i.e. 82.2 μs $<$ 350 μs.). However, if the algorithm would not have been parallelised, the computation of a packet would have taken K times longer. In this case the packets would have been processed in 1.64 ms becoming the limiting factor of the process (i.e. 1.64 ms $>$ 350 μs). On average, this means that for the complete processing of a waveform, it would have taken 400 ms which is at the limit of what is acceptable for real time applications.

Conclusion

This concludes this chapter.

The hardware implementation of the STNO simulation was discussed, proving the good communication between an embedded software of model of machine learning and a neuromorphic hardware device. To carry out this part, it was necessary to proceed in several steps.

First, the algorithm was optimised so that it could be faster on the ASIC. It turns out that without this optimisation, the model would have been 5 times slower and the system would have experienced borderline delays for real time applications.

Then, the information had to be communicated to the hardware. For this purpose different interface layers were detailed, namely the software interface and the SPI communication. If another device is to be integrated, it is likely that similar interfaces will have to be implemented.

Finally, the implementation of the dynamics simulation was examined. For this, the formula was decomposed and each of the modules implementing a part of the algorithm were detailed. There were the data managers which prepare and manage the conditions and data. The computation modules took care of the actual calculation of the dynamics.

In conclusion, the hardware implementation gives the same results as its software equivalent. This means that the work is a success.

Chapter 5

Results and perspectives

Although this system works well, it is interesting to benchmark the first version of such a system with the state of the art model described in chapter 2.

This chapter first recontextualises the models and dataset studied. Then, it describes the results obtained for the training and inference of each model. It discusses the accuracy and speed performance. Finally, it addresses the ways in which the model can be improved.

Figure 5.1 shows the conceptual positioning of the chapter in the work flow. The final test is a very high level task. This is why it is shown at the top of the figure, as if the models were black boxes.

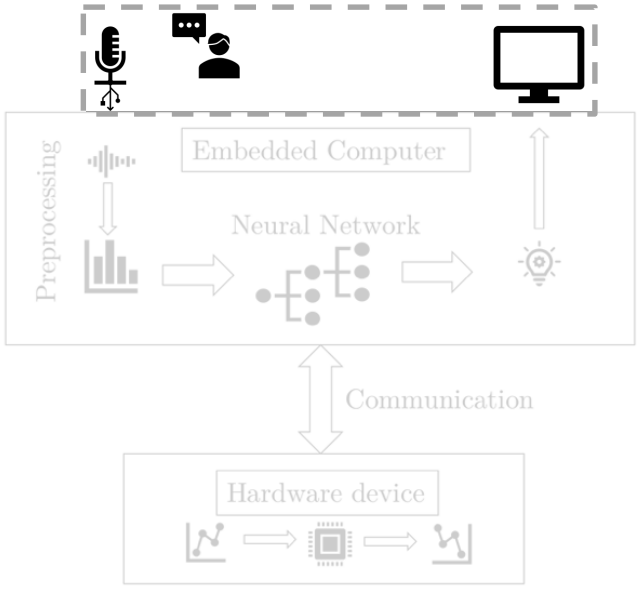

Figure 5.1: Testing the model in the global work-flow.

5.1 Foreword on datasets and compared models

In order to ensure relevant and consistent results, it is necessary to put the elements influencing training performance into context: the model and the dataset.

Models

First of all it is important to recontextualise the models in terms of their implementation, their number of parameters, the device on which they run,...

On the one hand, there is the M5 model. This one comes from a very fashionable architecture: the convolutional neural network. Although it is described as a lightweight model, it has 25290 parameters to train. Moreover its implementation has been made with `pyTorch`, a very optimized python library.
Please note that the results described below are the result of training as described in the paper, without personal tuning. The training is stopped when the training losses have less than one percent variation for five periods in a row.

On the other hand, there is the model described in this report. It comes from reservoir computing, a pioneering architecture in machine learning. It has only 400 parameters to train, which leaves it much less room for adjustment. Its implementation is homemade and is not as optimized as `pyTorch`. Moreover the hardware version runs on a device clocked at 50 MHz against 1.5 GHz for the M5 model running on Raspberry Pi.

However, it has the advantage of being a fast learning system that finds a global optimum for its weights.

It also allows tau wise inference which means that it has the ability to train itself to recognize words over time windows, even if the word is not pronounced in its entirety.

Finally, it should be noted that both the hardware and software versions of reservoir computing give the same classification results and therefore have the same accuracy.

Note that to enable a consistent time performance comparison, all models are tested directly on the Raspberry Pi.

Dataset

On the one hand there is the TI-46 with 500 elements and widely used in the reservoir computing hardware community.

On the other hand, there is the SPEECHCOMMANDS database with more than thirty thousand elements. This one is very much used by the development community working on unique word speech recognition models.

To compare the performance of models on datasets it is necessary for the sets to be of the same size. Indeed, a model is more likely to generalise well if it has a large database to train on. Thus, it was necessary to first truncate the SPEECHCOMMANDS database to 500 elements to have a reference point. Then, it was possible to evaluate the models on a larger dataset, they were then tested on 1000 SPEECHCOMMANDS elements.

5.2 Performances

		Train						Inference
Database		TI-46 (500 samples)		SpeechCommands (500 samples)		SpeechCommands (1000 samples)		
Model		Test accuracy in [%]	Time in [s]	Test accuracy in [%]	Time in [s]	Test accuracy in [%]	Time in [s]	Time in [ms/s]
M5		99,8	420	70,6	280	81,2	490	1,2
RC - MFCC	Pure software	99,8 (76,3)	313	69,8 (25,8)	250	79,8 (27,7)	335	180
	Hardware		296		232		343	160
RC - Cochlear	Pure software	99,6 (88,5)	170	42,2 (30)	65	50,2 (33,3)	109	88
	Hardware		157		53		109	80

Figure 5.2: Performance table comparing the two models according to their accuracy on the test set and time of training and inference. τ-wise performance are in parentheses.

5.2.1 Training performance

Here are the important points observed on the results table 5.2. They are divided into a precision and timing analysis.

Accuracy analysis

- Although the performances are very close, the M5 model is the most accurate, followed by the reservoir computing using MFCC and then cochlear. This can be explained by the much higher number of parameters that the M5 can adjust.
- The SPEECHCOMMANDS dataset seems more complicated than the TI-46. Indeed, the results are much lower for the models training on the 500 SPEECHCOMMANDS data.
- The models generalise better on the large dataset. Indeed, there is more data to train on.
- The tau-wise training is obviously less efficient because it is only based on window of sizes $\tau_{\text{MFCC}} = 25$ ms and $\tau_{\text{Cochlear}} = 50$ ms. These windows contain only incomplete fragments of digit pronunciation waveform.
- Cochlear offers poorer overall performance but better tau-wise. This is probably due to its larger window size. Indeed, it is easier to recognise a part of a word when it is longer.

Timing analysis

- Reservoir computing is faster in training.
- The hardware model is faster than its software counterpart except when the dataset becomes larger. Since all parts of the algorithm are common except for the transformation in the STNO, it can be assumed that the software has a sub-linear complexity for this part. On the other hand, the hardware has a linear complexity.
- The training time is shorter on the SPEECHCOMMANDS database items. Their pronunciation waveforms are probably shorter.

5.2.2 Inference performance

- The inference of the MFCC hardware model is twice as slow as the Cochlear. Indeed, the data sent to the STNO is proportional to the N_τ, so the computation time too.
- Hardware model inference is faster than software inference.
- The inference time of the M5 is much less than that of reservoir computing. Indeed, the M5 is implemented with a `Pytorch` being very optimized while the reservoir computing is a home-made implementation. Moreover, the hardware version is clocked at 50 MHz while the Raspberry Pi on which the M5 runs is clocked at 1.5 GHz.

5.3 Perspectives

Several improvements to the model are possible. This section deals with hardware and algorithm improvements for reservoir computing :

Algorithm improvements

Several improvements are possible to improve the reservoir computing algorithm integrating a hardware neuron:

- Optimise the number of neurons according to the number and size of the dataset.
- Optimize the mask $\mathbf{M}$ so that it is no longer random but deterministic while keeping data transformation patterns suitable for exploring the whole transient dynamics of the STNO.
- Improve the training algorithm so that it can work with batch data for training.

Hardware improvements

Various hardware improvements are also possible and have already been considered:

- Improve communication between software and hardware. For example, increase the SPI exchange frequency or put several channels in parallel.
- Group modules to save clock cycles. For example, it would be possible to directly combine the $\text{factE}_{\text{generator}}$ with the interpolate module and thus gain τ_{mult} cycle.
- Increase the packet size while using FPGA memory bits to be able to store more information on it (Memory space being the limiting factor to increase the packet size). The bigger the packets, the faster the communication.
- Increase the clock frequency of the ASIC.

The ultimate goal would be to integrate an STNO directly on a System on Chip and clock the communication at the same rate as the processor. If such work is done, it would be potentially possible to achieve inference speeds in the ms range if the clock frequency is in the order of GHz. This is entirely compatible with the STNO, which can be sampled at a frequency of the order of MSample/s.

Conclusion

This thesis has met the objectives set.
It has been demonstrated that the reservoir computing architecture integrating a hardware neuron is possible in an embedded way. In order to prove that it works, a speech recognition task of the 10 spoken digits has been proposed and performed. A comparison with a state-of-the art embedded model using known architecture was made. The results are of the same order in terms of accuracy and are promising in terms of timing but also in terms of energy consumption.

In order to carry out this project, it was first necessary to understand the challenges of speech recognition. The first part of the project focused on the processing of the sound waveform signal. Cleaning and voice detection methods were discussed.

Next, the classification was discussed. Different machine learning methods were considered for this purpose. General concepts for dealing with datasets were discussed. A description of known models such as the single and multilayer perceptron were explained. Finally, the convolutional neural network was described and an example of such a neural network was implemented for future bench-marking.

Afterwards, it was necessary to understand the models allowing hardware integration. The one chosen was the reservoir computing model because of its numerous advantages. An adaptation of this was explored in order to integrate a single hardware neuron. This neuron has to be a neuromorphic device with memory and a non-linear response. A proof of concept was first tested in software.

Once all the understanding was established, the hardware implementation of an STNO simulation could be addressed. An adaptation of the algorithm was first made to allow a better efficiency of the ASIC prototype. Then, the interfaces were designed and implemented. Then, condition and data management modules were described and developed. Finally, the computational units were presented.

To conclude this thesis, benchmarks were carried out in order to situate the implemented model against a classical architecture. The results were very conclusive. However, several improvements have been proposed.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[2] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 818–833, Springer International Publishing, 2014.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[4] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," tech. rep., Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[5] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *Science*, vol. 304, no. 5667, pp. 78–80, 2004.

[6] W. Maass, T. Natschläger, and H. Markram, "Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations," *Neural Computation*, vol. 14, pp. 2531–2560, 11 2002.

[7] E. H. Moore, "On the reciprocal of the general algebraic matrix," *Bull. Am. Math. Soc.*, vol. 26, pp. 394–395, 1920.

[8] R. Penrose, "A generalized inverse for matrices," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 51, no. 3, p. 406–413, 1955.

[9] L. Appeltant, M. Soriano, G. Van der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C. Mirasso, and I. Fischer, "Information processing using a single dynamical node as complex system," *Nature communications*, vol. 2, p. 468, 09 2011.

[10] J. Torrejon, M. Riou, F. A. Araujo, S. Tsunegi, G. Khalsa, D. Querlioz, P. Bortolotti, V. Cros, K. Yakushiji, A. Fukushima, H. Kubota, S. Yuasa, M. D. Stiles, and J. Grollier, "Neuromorphic computing with nanoscale spintronic oscillators," *Nature*, vol. 547, no. 7664, pp. 428–431, 2017.

[11] E. Upton and G. Halfacree, *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[12] Terasic, "De0-nano soc."

[13] C. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, pp. 10–21, jan 1949.

[14] H. Pham, "Pyaudio."

[15] E. W. Fleischman, "WAVE and AVI Codec Registries." RFC 2361, June 1998.

[16] R. V. Cox, S. F. De Campos Neto, C. Lamblin, and M. H. Sherif, "Itu-t coders for wideband, superwideband, and fullband speech communication [series editorial]," *IEEE Communications Magazine*, vol. 47, no. 10, pp. 106–109, 2009.

[17] S. Butterworth, "On the theory of filter amplifiers," *Experimental Wireless and the Wireless Engineer*, vol. 7, pp. 536–541, 1930.

[18] Wikipedia, "Filtre de butterworth."

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[20] S. Mallat, *A wavelet tour of signal processing*. Elsevier, 1999.

[21] A. Haar, "Zur theorie der orthogonalen funktionensysteme," *Mathematische Annalen*, vol. 69, no. 3, pp. 331–371, 1910.

[22] J. Laurent, "LELEC2885 : Lecture notes about The Wavelet Transform," September 2020.

[23] D. Valencia, D. Orejuela, J. Salazar, and J. Valencia, "Comparison analysis between rigrsure, sqtwolog, heursure and minimaxi techniques using hard and soft thresholding methods," in *2016 XXI Symposium on Signal Processing, Images and Artificial Vision (STSIVA)*, pp. 1–5, 2016.

[24] D. L. Donoho and I. M. Johnstone, "Adapting to unknown smoothness via wavelet shrinkage," *Journal of the american statistical association*, pp. 1200–1224, 1995.

[25] P. Mei, R. Liu, I. Zhi, J. Huang, D. HuiQian, and H. Nahrstaedt, "Pyyawt," 2009.

[26] M. Liberman et al., "Ti 46-word ldc93s9. web download," 1993.

[27] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar, "Optoelectronic reservoir computing," *Scientific Reports*, vol. 2, Feb 2012.

[28] F. Abreu Araujo, M. Riou, J. Torrejon, S. Tsunegi, D. Querlioz, K. Yakushiji, A. Fukushima, H. Kubota, S. Yuasa, M. D. Stiles, and J. Grollier, "Role of non-linear data processing on speech recognition task in the framework of reservoir computing," *Scientific Reports*, vol. 10, no. 1, p. 328, 2020.

[29] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018.

[30] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 357–366, 1980.

[31] R. Lyon, "A computational model of filtering, detection, and compression in the cochlea," in *ICASSP '82. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 7, pp. 1282–1285, 1982.

[32] M. Slaney, "Lyon's cochlear model," *Apple Computer, Inc., Cupertino, CA, apple computer technical report 013 edn*, 1988.

[33] D. Tkanov, "Lyon's auditory model for python."

[34] M. Slaney, "Auditory toolbox," *Apple Computer, Inc., Cupertino, CA, apple computer technical report 010 edn*, 1988.

[35] F. Mosteller and J. Tukey, "Data analysis, including statistics.," *Handbook of Social Psychology. Addison-Wesley, Reading, MA,*, 1968.

[36] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[38] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[39] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[40] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[41] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[42] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[43] W. Dai, C. Dai, S. Qu, J. Li, and S. Das, "Very deep convolutional neural networks for raw waveforms," 2016.

[44] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[45] G. V. der Sande, D. Brunner, and M. C. Soriano, "Advances in photonic reservoir computing," *Nanophotonics*, vol. 6, no. 3, pp. 561–576, 2017.

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