

A Low Power and Real-Time Hardware Recurrent Neural Network for Time Series Analysis on Wearable Devices

Emanuele Torti*, Cristina D'Amato, Giovanni Danese, Francesco Leparati

Department of Electrical, Computer and Biomedical Engineering, University of Pavia, Pavia, Italy

*Corresponding author: emanuele.torti@unipv.it

Abstract

The research presented in this paper addresses the exploitation of Deep Learning methods on wearable devices. We propose a hardware architecture capable of analyzing time series signals through a Recurrent Neural Network implemented on FPGA technology. This architecture has been validated using a real dataset, which includes three-axial accelerometer data acquired by a wearable device used for fall detection. The experiments have been conducted considering different devices and demonstrates that the proposed hardware architecture outperforms the state of the art solutions both in terms of processing time and power consumption. Indeed, the proposed architecture is real-time compliant in the elaboration of the fall detection dataset adopted for the validation. The power consumption is in the order of dozens μ W. Finally, further functionalities could be added in the same chip since the resource usage is low.

Keywords—*Embedded systems, Deep Learning, Hardware architectures, FPGA, Wearable devices.*

1. Introduction

Several healthcare applications are based on time series signals acquired by portable or wearable devices. Examples are given by sleep apnea study [1], gait analysis [2], heart rate monitoring [3], and fall detection [4]. Typically, these applications monitor parameters such as heart rate, blood pressure, and temperature through wearable and intelligent sensors, performing the so-called *personal monitoring*. Recently, automatic classification of daily activities attracted the scientific community, to point out uncommon and dangerous events such as unintentional falls [5]. Essentially, a wearable device acquires the signals from one or many sensors such as accelerometers, gyroscopes, and/or barometers; the data are then elaborated to identify if an unintentional fall happened.

In the literature, the automatic classification of time series signals acquired by wearable devices has been addressed by proposing different techniques from threshold-based to Deep Learning methods. The threshold-based algorithms are among the first proposed methods since they are simple and feature a low computational complexity. This last characteristic is of crucial importance and allows to directly implement these algorithms on the wearable device. Indeed, the processing unit performs simple operations with a low impact on battery life. However, the threshold-based approach suffers from low detection accuracy in many situations, as an example when different types of falls are considered.

To overcome this limit, the researchers adopted machine learning methods, such as Naïve Bayes [6], Support Vector Machines (SVM) [7], and Deep Learning [4, 8]. Initially, these algorithms were executed on a remote host rather than on the wearable device, since their computational complexity is high and the performed operations would fast drain the battery charge. Thus, many systems feature two main modules: a wearable device and a remote host. The former acquires and transmits the data to the latter, which is a processing unit charge with the classification. It is important to highlight that this scheme is based on *off-board* processing, which requires continuous data transmission between the wearable device and the remote host. This represents the main limitation of the previously mentioned approach since

communication is the most power-consuming part of the system. Thus, continuous data transmission negatively affects negatively battery life.

The implementation of Deep Learning methods on wearable devices has emerged as a feasible technique only in recent years. TensorFlow, the most used framework for Deep learning, has been ported to smartphones and embedded devices with the release of TensorFlow Lite¹. At the moment, Tensorflow Lite includes only a subset of the original TensorFlow methods, limiting the exploitation of this framework for embedded applications. Another limit is given by the supported devices: only a few microcontrollers based on the Cortex M3 architecture can use TensorFlow Lite. Finally, among the unsupported methods in TensorFlow Lite, the Recurrent Neural Networks (RNNs) are the state of the art for many time series analysis applications.

This paper proposes a novel hardware architecture based on RNNs capable of meeting the constraints given by a wearable device. The architecture is validated on different FPGA devices with three-axial accelerometer data acquired by a wearable device for automatic fall detection. During the experiments, the processing time and power consumption have been measured or estimated. The obtained results show that the proposed architecture correctly classifies the time series signals. Moreover, it outperforms other state-of-the-art solutions both in terms of processing time and power consumption (see Results and Discussion section). The proposed architecture has been developed in order to provide a very low-power LSTM hardware implementation, real-time compliant and without the need for an external and power-consuming HPC elaboration.

This activity prolongs the tradition of the authors' laboratory, where, in the past, embedded and wearable devices were developed [9, 10].

This paper extends the work presented in [11] and is organized as follows. Section 2 describes the Recurrent Neural Networks focusing on Long Short-Term Memory (LSTM), which is the state-of-the-art for time series analysis. Section 3 details the developed hardware architecture, while Section 4 contains the experimental results, together with their discussion. Section 5 concludes the paper and addresses future research work.

The main contributions of this work are:

- the description of a real-time and low power LSTM network hardware architecture;
- the validation of the proposed architecture with a real dataset;
- the evaluation of the proposed architecture on different FPGA chips;
- the comparison with state-of-the-art solutions.

2. Time Series Analysis Thorough Deep Learning

Deep Learning algorithms are the state-of-the-art solution for in different fields ranging from signal processing [12, 13] to imaging [14, 15]. The literature demonstrates that RNNs are the best Deep Learning solution to classify time series signals [16].

2.1. Recurrent Neural Networks

RNNs are a particular artificial neural network where part of the output is fed back as input. In general, the expression of an RNN is given by (1):

$$y^{(t)} = wg(Wx^{(t)} + Uh^{(t-1)} + b) + c \quad (1)$$

¹ <https://www.tensorflow.org/lite> last access November 2020

where $x^{(t)}$ and $y^{(t)}$ represent the input and the output at time t , while W, U, w, b and c are the network parameters. The term g denotes a suitable non-linear function. The term $h^{(t)}$ represents the hidden state, given by:

$$h^{(t)} = g(Wx^{(t)} + Uh^{(t-1)} + b) \quad (2).$$

Figure 1 shows the typical structure of an RNN.

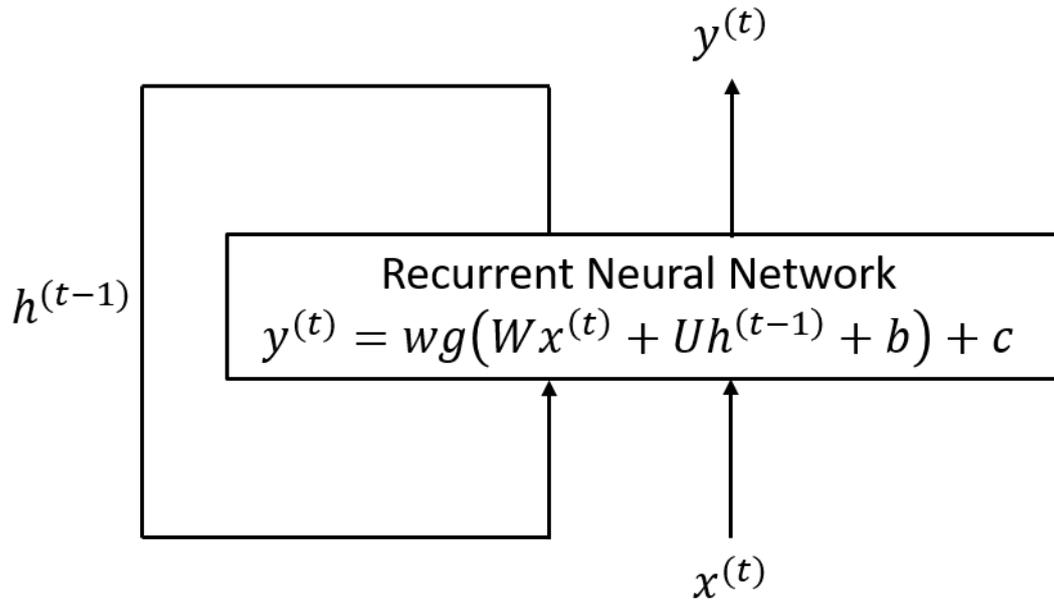


Figure 1 – The typical structure of an RNN, where part of the output is fed back as input.

RNNs adopt a *sliding window* approach to scan the input stream. Therefore, the input stream is split into chunks of equal size, according to the dimension of the sliding window. The RNN analyses a single chunk and, then, the internal state is re-initialized before starting to elaborate the next chunk. This process is called *inference* and it is capable of recognizing patterns in the input stream. The computational complexity of the inference is reduced by sliding the input window w_i at constant time intervals, called *strides*. These concepts are summarized in Figure 2.

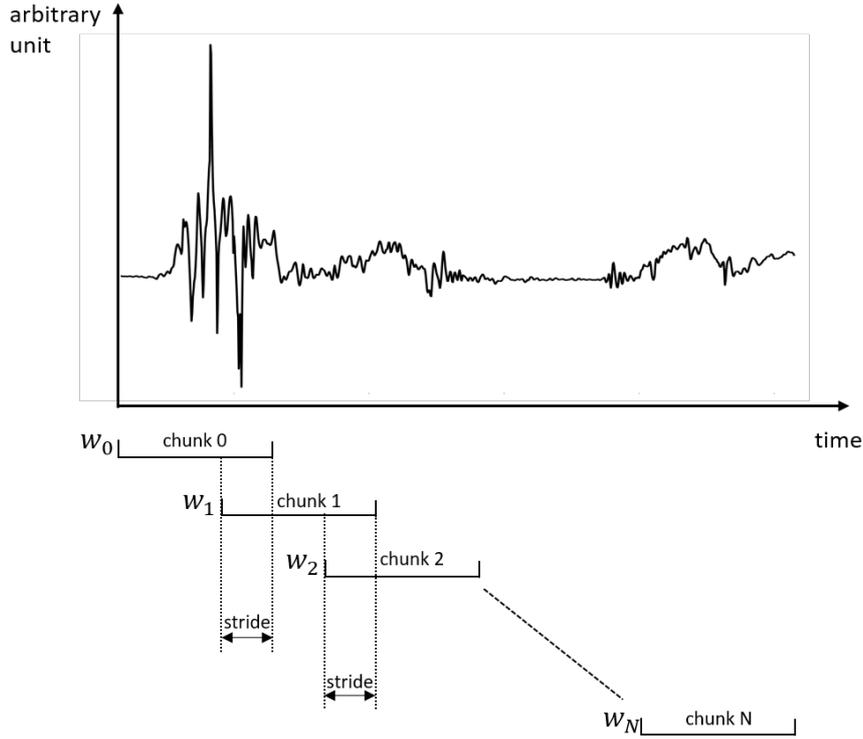


Figure 2 – Sliding window, chunk, and stride.

2.2. Long Short-Term Memory Networks (LSTM)

Among the different kinds of RNNs, the LSTM are capable of learning long-term temporal dependencies. The main difference with a standard RNN is that the input is distributed over four gates. Each of these gates is described by an expression similar to (1). The four gates are called the input gate ($i^{(t)}$), the output gate ($o^{(t)}$), the cell gate ($c_{in}^{(t)}$), and the forget gate ($f^{(t)}$). A single LSTM is described by (3)-(8):

$$i^{(t)} = \text{sigmoid}(W_i x^{(t)} + U_i h^{(t-1)} + b_i) \quad (3)$$

$$o^{(t)} = \text{sigmoid}(W_o x^{(t)} + U_o h^{(t-1)} + b_o + b_{forget}) \quad (4)$$

$$c_{in}^{(t)} = \tanh(W_c x^{(t)} + U_c h^{(t-1)} + b_c) \quad (5)$$

$$f^{(t)} = \text{sigmoid}(W_f x^{(t)} + U_f h^{(t-1)} + b_f) \quad (6)$$

$$c^{(t)} = f^{(t)} * c^{(t-1)} + i^{(t)} * c_{in}^{(t)} \quad (7)$$

$$h^{(t)} = o^{(t)} * \tanh(c^{(t)}) \quad (8)$$

where $W, U \in \mathbb{R}^{LS \times LS}$, $i^{(t)}, o^{(t)}, c_{in}^{(t)}, f^{(t)}, c^{(t)}, h^{(t)}, x^{(t)}, b \in \mathbb{R}^{LS}$ and the operator $*$ denotes the element-wise product. The term b_{forget} in eq. (4) is the *forget bias* and it is typically set equal to 1 [17]. The term LS is an hyperparameter called *inner size* or *LSTM size*.

This model can be extended by connecting multiple LSTM cells in a cascade, obtaining the so-called *stacked LSTM network*. Finally, the stacked LSTM network can be extended by introducing fully connected layers and/or activation functions. These operations increase the network depth, ensuring a high level of abstraction [18].

2.3. Description of the Adopted Network

This paper considers a stacked LSTM network designed for automatic fall detection [4]. The network includes a fully connected layer to process the inputs, two stacked LSTM cells, and another fully connected layer to elaborate the outputs. The inputs processed by the fully connected layer are given as input to the ReLU function, whose output is the input of the first LSTM cell. The output of the second fully connected layer is given as input to the *softmax* functions, providing the final system output. The work in [4] describes a data organization that allows computing expressions in the form of (3)-(6) as a matrix-vector multiplication followed by a vector-vector sum. In this data organization, the LSTM cell weights and biases are packaged into a single matrix W and b , defined as follows:

$$W = \begin{bmatrix} W_i & W_c & W_f & W_o \\ U_i & U_c & U_f & U_o \end{bmatrix} \quad (9)$$

$$b = [b_i \quad b_c \quad b_f \quad b_o] \quad (10)$$

to compute the vector v :

$$v = [i^{(t)} \quad c_{in}^{(t)} \quad f^{(t)} \quad o^{(t)}] \quad (11)$$

using a single expression like as:

$$v = rW + b \quad (12)$$

with r defined as:

$$r = [x^{(t)} \quad h^{(t)}] \quad (13).$$

The W matrix has $2LS$ rows and $4LS$ columns, the b and v vectors have a size of $4LS$ while the r vector includes $2LS$ elements.

The time series given as input to the network has a size of $NF \times ww$, where NF is the number of features and ww is the window width. The network's output is a vector with NC elements, where NC indicates the number of classes considered in the inference process. The $i - th$ element of the vector indicates the probability of belonging to the $i - th$ class.

The architecture of the adopted stacked LSTM network is reported in Figure 3, together with the dimensionality of each stage.

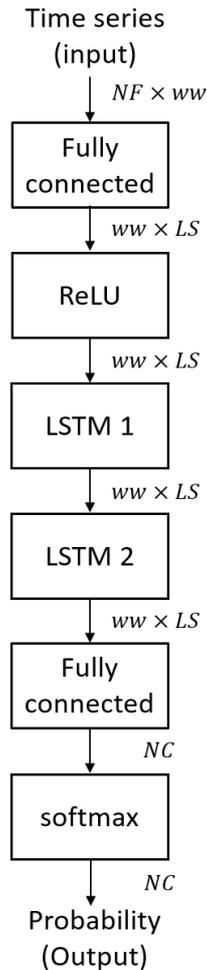


Figure 3 – The structure of the adopted network. The dimensionality is reported close to the arrows.

The network described in [4] targets automatic fall detection performing inference on data acquired by a three-axial accelerometer. The network works on a time interval of 1 second, while the accelerometer acquires data at 100 Hz frequency. Thus, NF and ww are equal to 3 and 100, respectively. The time series are classified into three classes, namely: *background*, *alert* and *fall*, that is the number of classes NC is 3.

The class *background* represents all the situations of the daily life that are not related to an unintentional fall. The class *alert* is related to events of interest, which may or may not directly lead to a fall. Finally, the class *fall* contains the time series signals related to an unintentional fall. Concerning the LS parameter, the optimal value has been estimated during the network training by testing different values. The tests revealed that the optimal inner dimension of each LSTM cell is equal to 32.

3. The Hardware Architecture

This section describes the hardware architecture, focusing on the solutions developed for the critical parts of the system.

First, the system acquires data from the accelerometer. Most of the commercially available accelerometers acquire data and store them into FIFO memories. Then, the sensor notifies the elaboration unit with a digital signal when the desired number of samples has been acquired. This mechanism is known as *watermark*. In the considered case (an accelerometer with 100 Hz sampling frequency) to acquire acceleration tracks of 1 second, the watermark level must be set to 100. After the samples acquisition, the

elaboration is performed by applying the fully connected layers, the LSTM cells, and the activation functions according to Figure 3.

3.1 Fixed point format

The most used software frameworks for Deep Learning perform inference mainly adopting the single precision floating-point representation, which ensures the best accuracy. However, when considering a hardware architecture, floating-point arithmetic is not a suitable choice, since the resource usage of the computational blocks is high. This is a critical issue because high resource usage causes an increment both in processing times and in power consumption. To efficiently develop an architecture suitable for wearable devices, the authors conceived a fixed-point representation that ensures a suitable precision for the computations. Moreover, the adoption of the fixed-point format keeps low resource usage.

The fixed-point format for the stacked LSTM network has been developed by exploiting the Fixed-Point Toolbox included in Matlab. This toolbox is designed to quickly test different fixed-point configurations and to evaluate the error between the fixed and floating-point algorithms. The authors performed several experiments to identify the number of the bits which gives a suitable precision. During these experiments, the inference is performed by a Matlab script containing a fixed-point version of the stacked LSTM network described in [4]. The script compares the result of the fixed and floating-point stacked LSTM networks and computes the Mean Squared Error (MSE) between the results. The experiments highlighted that the best solution is given by a 22 bits representation. In particular, 10 bits are devoted to the integer part, while the remaining 12 to the decimal part. This format adopts the two's complement notation to consider signed numbers. Figure 4 shows the relation between the MSE and the number of bits used for the fractional part.

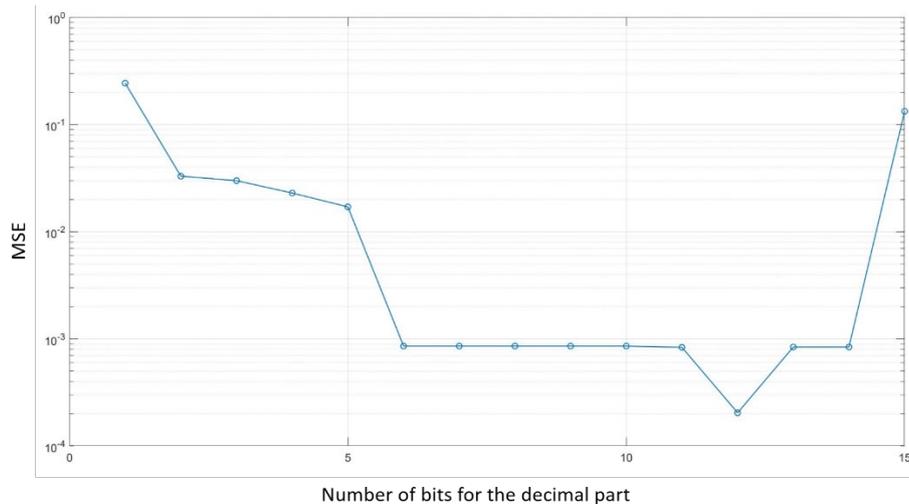


Figure 4 – MSE for the number of bits for the decimal part. The y-axis is on a logarithmic scale.

The analysis of the chart reported in Fig. 4 shows that the MSE between the floating-point algorithm and the fixed point one is about 10^{-4} , for the considered format.

3.2 Overview of The Proposed Architecture

The proposed architecture can be divided into three main blocks:

1. *pre-processing*;
2. *LSTM cells*;
3. *classification*.

These three functional blocks are connected in *cascade*. The data exchange between the blocks is managed by a distributed control logic. Figure 4 shows a diagram of the architecture and highlights the main functional blocks for each system part.

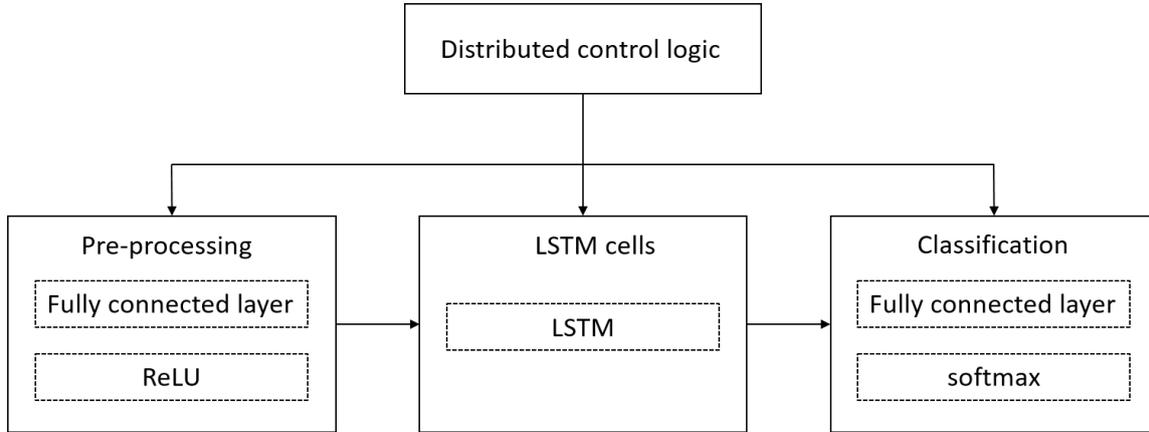


Figure 5 – Diagram of the system architecture. The functional blocks of each system part are highlighted by the dashed lines.

As shown by Figure 5, the pre-processing includes two main functional blocks: a fully connected layer and the ReLU function. The output of the pre-processing is given as input to the LSTM cells block. This part is the core of the system and is made up of only one functional block, called LSTM. It is worth noticing that only one LSTM block is used to design this part of the system. As detailed in the following, the block contains all the logic circuits needed to evaluate a single LSTM cell. This design choice has been taken to keep low resource usage. Again, the output of this part of the system is given as input to the next one. The last part, the Classification, includes two main functional blocks: a fully connected layer and the softmax function. It is worth noticing that the fully connected layer block is logically equivalent to the one in the pre-processing. The main difference is in the data dimensionality on which the blocks operate.

3.3 The Fully Connected Layer Functional Block

In general, the operation performed by a fully connected layer is described by (14):

$$x_{new} = x_{in}W_{FC} + b_{FC} \quad (14)$$

where x_{in} is the input, x_{new} is the output, W_{FC} is the weight matrix and b_{FC} is the bias vector.

The values of W_{FC} and b_{FC} are estimated during the training phase of the network and they do not change during the inference. Thus, these values are stored in two different ROM memories, one for the weights and one for the bias. The input x_{in} is stored into a FIFO memory since it must be read and written by other components. Concerning the pre-processing, the input is acquired by an accelerometer and stored in a FIFO memory. The content of this memory is the input of the fully connected layer functional block of the pre-processing. On the other hand, when considering the classification part, the input of the fully connected layer is the output of the LSTM cells part. In this case, the second LSTM cell output is written into the FIFO memory.

Summarizing, the fully connected layer functional block receives as input the content of two ROMs (weights and bias) and of a FIFO (input values) and produces an output which is stored in another FIFO memory.

Figure 6 shows the fully connected layer architecture. In this figure, the control signals are highlighted by dashed arrows.

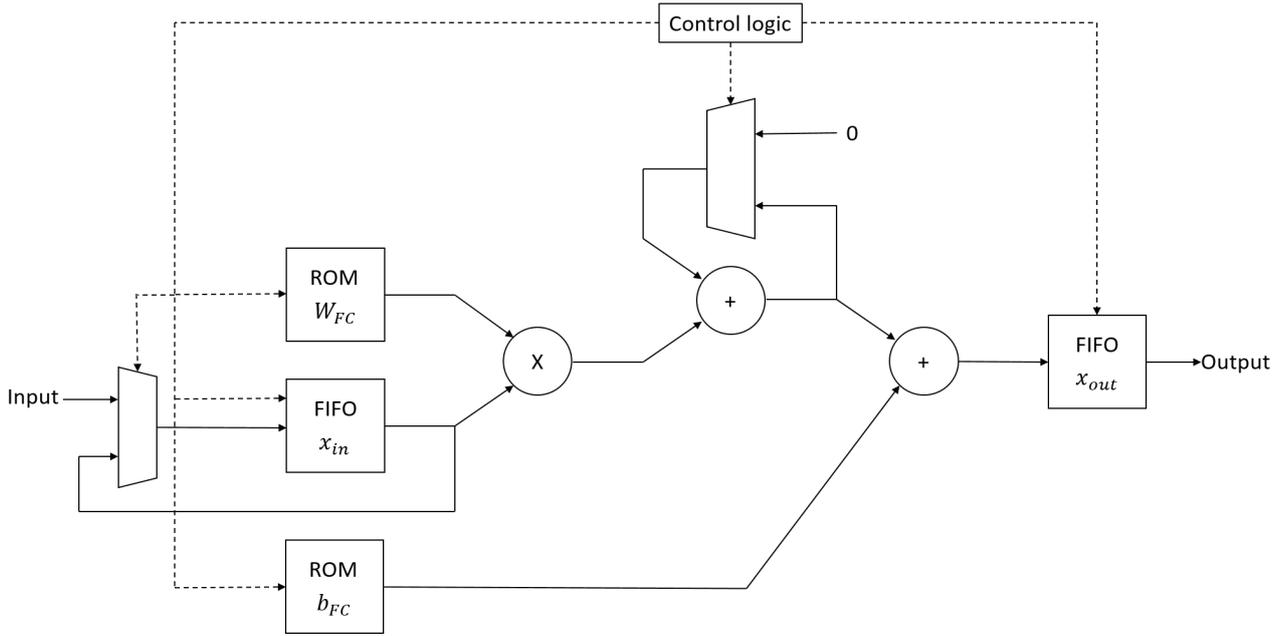


Figure 6 – The architecture of the fully connected layer. The control signals are highlighted by the dashed arrows.

Based on which fully connected block is considered, the term input in Figure 6 indicates the data acquired by the accelerometer or the output of the second LSTM cell.

The FIFO x_{in} has a feedback loop since the data, after being read, must be stored again in the same FIFO. In fact the same data must be multiplied by each row of the W_{FC} matrix. The feedback loop is carried out through a multiplexer, which transfers to the FIFO the input to the block or the datum read from the same FIFO. This feedback loop can be avoided by instantiating multiplication and sum blocks equal to the number of rows of the W_{FC} matrix. However, due to the data dimensionality, this will lead to high resource usage.

The addresses generation to read the memories and the enabling signals are managed by the control logic. The pairs of values read by the ROM W_{FC} and the FIFO x_{in} are sent to a multiplier. The result of this multiplication is the first input of an adder. The second input of the adder is 0 when the product is the first of each W_{FC} matrix row, while it is the previous sum otherwise. Again, this selection is managed by the control logic through a multiplexer.

Finally, the output of this adder is summed to the content of the ROM b_{FC} and the result is stored in the FIFO x_{out} . In this case, also the FIFO is written only when the results are ready and the FIFO write signal is managed by the control logic.

3.4 The non-linear functions

The proposed architecture relies on three non-linear functions: ReLU, hyperbolic tangent (*tanh*), and *sigmoid*. The first one is used in the ore-processing, while the others are in the LSTM cells.

The ReLU function is defined as:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

As described before, the numerical representation is based on the two's complement notation; therefore, this function can be efficiently implemented using a single multiplexer. Indeed, the Most Significant Bit (MSB) of a value indicates the sign of the number. In particular, if the MSB is equal to one the number is

negative; otherwise, the value is positive. Thus, the MSB can be used to control the multiplexer. The circuit is shown in Figure 7.

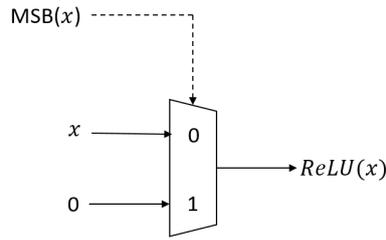


Figure 7 – The architecture of the ReLU function. The control signal is highlighted by the dashed arrow.

When the MSB is equal to 0, the multiplexer transfers to the output port the value of x ; otherwise, it transfers the constant value 0.

The \tanh function is defined as:

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \quad (16)$$

Therefore, the \tanh function can be implemented using the exponential function, an adder (to sum or subtract), and a divider. The exponential function is typically implemented exploiting the Taylor series. Experiments conducted with the Fixed Point Toolbox showed that a high order of the Taylor series should be taken into account to obtain a suitable precision. However, the Taylor series of a high order is onerous from a computational point of view; moreover, the resource usage is not negligible. Therefore, the authors developed a strategy to limit the order of the Taylor series to employ. This strategy is based on empirical observations on the chart of the function, which leads to a piecewise definition. In particular, the authors noticed that the function output saturates to the value 1 when the input value is greater than 3.3 and to the value -1 when the input is less than -3.3. Between the input values -3.3 and 3.3, two other behaviors can be defined: a linear and a non-linear one. The non-linear part includes input values from -3.3 to -1.1 and from 1.1 to 3.3. The output values of these ranges have been tabulated, to avoid a high order Taylor series. Considering the remaining input range, i.e. values included between -1.1 and 1.1, the Taylor series of the ninth order is adopted. This function is implemented using three multiplexers, connected as shown in Figure 8. The multiplexer on the right side of Figure 8 selects among the LUT output and the other approximations computed by the circuit. Again, the approximated value is chosen through a set of multiplexers.

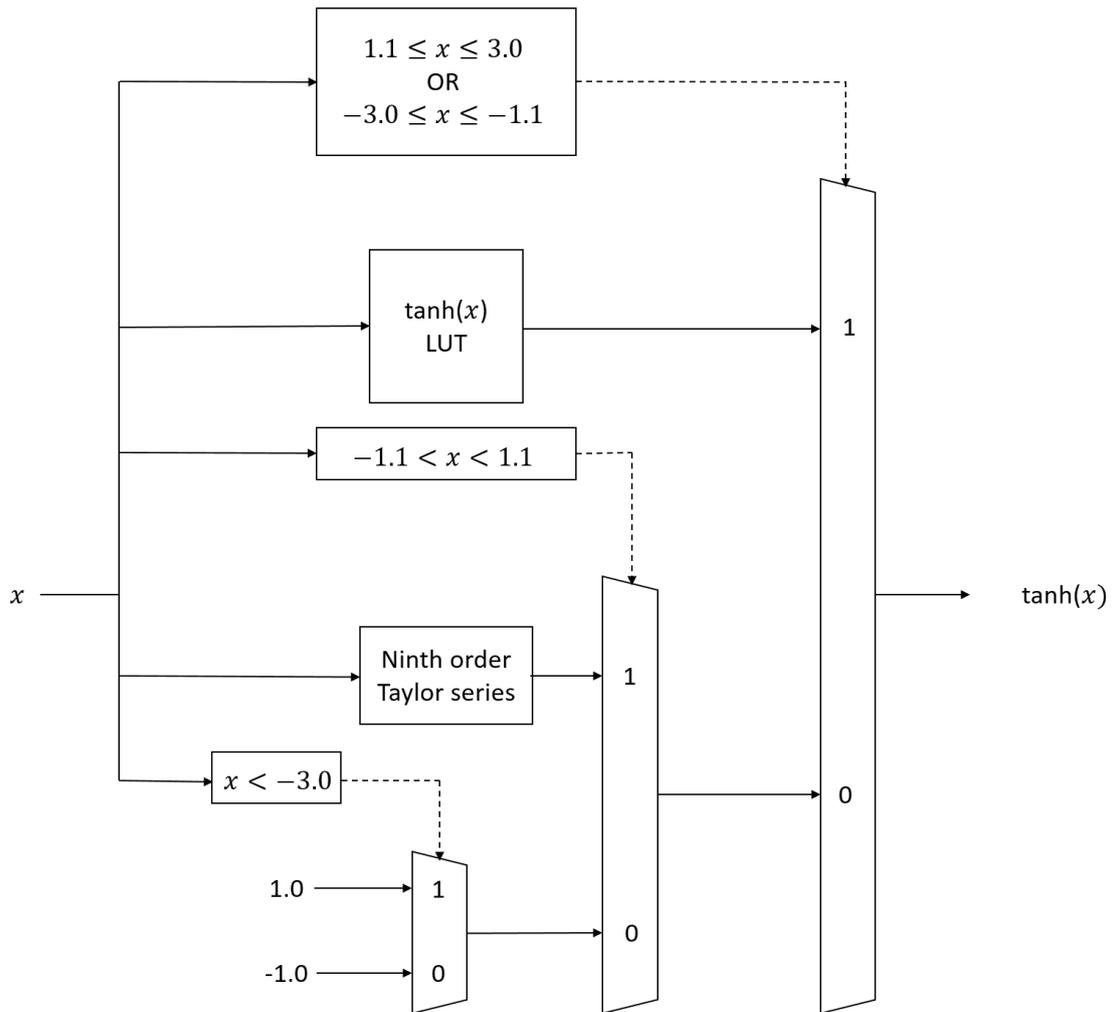


Figure 8 – Approximation of the tanh function.

Figure 9 shows the comparison between the proposed fixed-point tanh function and the floating-point version, which has been computed with the Matlab routine. In this figure, the red continuous line represents the floating-point version of the tanh function, while the fixed-point values are indicated by the blue markers. The chart clearly shows that the piecewise definition is a suitable approximation of the real tanh function. The MSE between the fixed and floating point tanh function is about 10^{-7} .

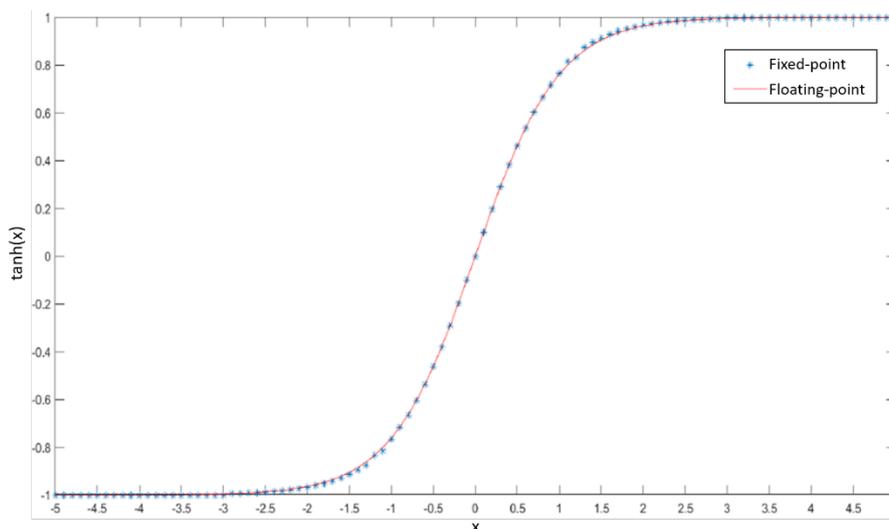


Figure 9 – Comparison between the fixed-point and the floating-point versions of the tanh function.

Finally, the sigmoid function can be defined as:

$$\text{sigmoid}(x) = \frac{\tanh(x/2) + 1}{2} \quad (17).$$

The sigmoid development is based on the tanh implementation, described above. The input of the sigmoid function is halved and given as input to the tanh routine. The halving operation can be easily performed by a logical right shift of the input value. The tanh output is then decremented by 1 through a suitable circuit and halved again by another logical right shift. Figure 10 shows the implementation of this function. Also in this case, the difference between the fixed-point and the floating-point versions of the sigmoid function is negligible, as shown in Figure 11.

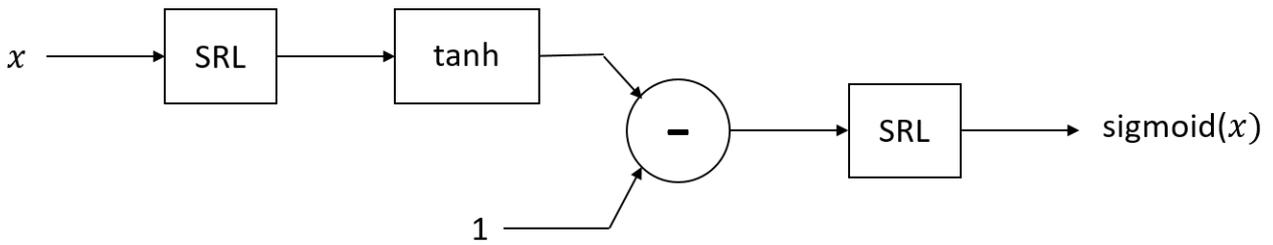


Figure 10 – Evaluation of the function.

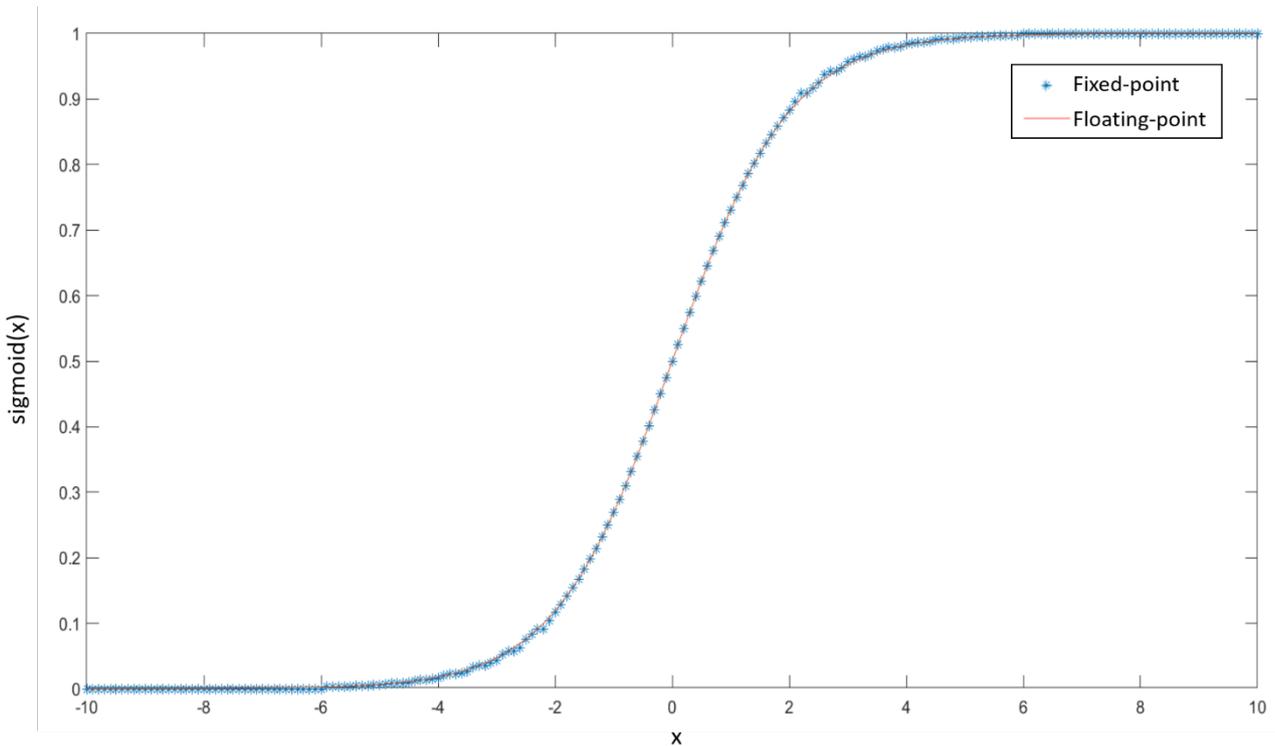


Figure 11 – Comparison between the fixed-point and the floating-point versions of the sigmoid function.

3.5 The LSTM cells

The LSTM cells block is the core of the system because it implements the main computations related to the LSTM. The network considered in this work consists of two different LSTM cells. Thus, two different choices can be made to design this part of the system. The first one is the simplest and requires to design a block with all the logic to perform the computation related to the LSTM cell. Then, this block is instantiated two times with suitable connections. Surely, this choice is easy to develop, but the resource usage increases

with the number of LSTM cells to instantiate. For this reason, the authors choose to instantiate the logic resources to compute a single LSTM cell and to properly manage the inputs to correctly evaluate the operations in cascade.

As discussed in Section 2.3, the computation of the argument of the non-linear functions of equations (3)-(6) can be performed through a simple vector-matrix multiplication followed by a vector-vector sum. This operation is the same for the two LSTM cells, but with different inputs. The W matrix and the b vector of each cell are stored in suitable ROM memories because the values are fixed in the training phase and must not change during inference. In the following, the matrix and vector related to the first LSTM cell will be denoted as W_1 and b_1 , respectively, while the notation W_2 and b_2 is used for the second LSTM cell.

The first operation is the product between the W matrix and the vector containing the input to the cell and state. Considering the first LSTM cell, the input is given by the output of the pre-processing, while, for the second LSTM cell, the input is the state of the first LSTM cell. Therefore, the authors decided to use three different FIFO memories. The first FIFO memory stores the output of the pre-processing, while the second FIFO stores the state of the first LSTM cell, which will be denoted with h_1 . The last FIFO stores the state of the second FIFO memory, which will be indicated with h_2 .

Figure 12 shows how the LSTM cells manage the inputs.

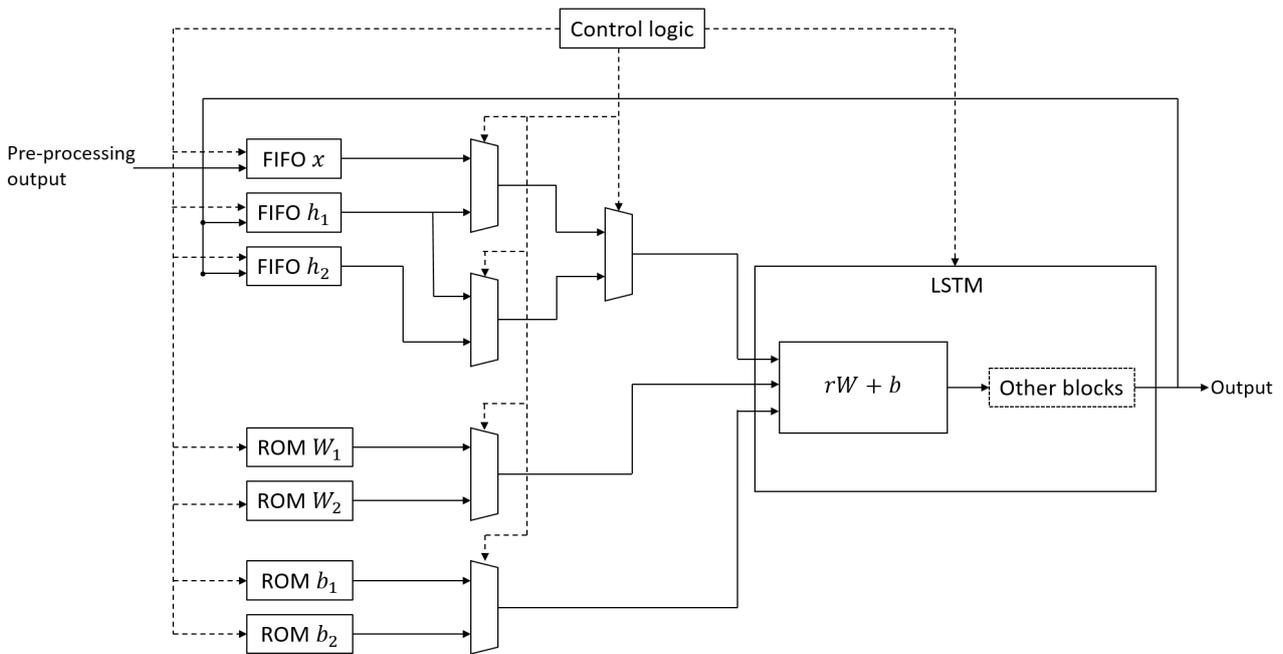


Figure 12 – The inputs management to the LSTM cells. The dashed arrows highlight the control signals.

The FIFOs are written and read by the LSTM block, together with the W matrices and the b vectors, as shown in Figure 12, where the control signals are highlighted with dashed arrows. The control logic enables the FIFO reading and writing based on the LSTM cell to evaluate. In the case of the first LSTM cell, the control logic enables the reading of the FIFO x for acquiring LS values and, then, enables the reading of the FIFO h_1 for the next LS values. It is worth noticing that the values read from each FIFO are fed back to the same FIFO to re-store them in the same memory. This is mandatory since the values must be read several times to correctly compute the vector-matrix product, as explained for the fully connected layer.

The control logic also manages the data reading from the ROM memories, ensuring the correct weights and bias transfer to the LSTM block. In particular, in particular, concerning the reading signals and the address given as input to the ROMs. Finally, it drives the multiplexers control to correctly transfer the data to the LSTM block.

After eq. (12) evaluation, the expressions (3)-(8) of the LSTM cell can be computed. The evaluation of these equations exploits the sigmoid and tanh functions, already described in Section 3.4. Figure 13 shows the architecture developed for these computations, which corresponds to the dashed block (“Other blocks”) in Figure 12, together with the block for the $rW + b$ computation.

As can be seen from Figure 13, the output of the $rW + b$ computation follows four different paths, each one corresponding to one equation from (3) to (6). The control logic enables the writing to the suitable FIFO on the basis of which result is ready to be stored. A counter performs the generation of the enable signals. The first LS values generated by the $rW + b$ block should be stored in the FIFO i memory since they are related to eq. (3). The same schema is applied to the eqs. (4)-(6). It is worth noticing that only one FIFO at a time is enabled to store LS values. Once these four FIFOs are full, i.e. each one contains LS values, the architecture is ready to evaluate eq. (7)-(8). The computation of $c^{(t)}$, instead, depends on $i^{(t)}$, $c_{in}^{(t)}$, $f^{(t)}$ and the previous values denoted as $c^{(t-1)}$. For this reason, another FIFO memory is used to store the values computed for $c^{(t)}$, which are fed back to the computation to correctly evaluate the next value of $c^{(t)}$.

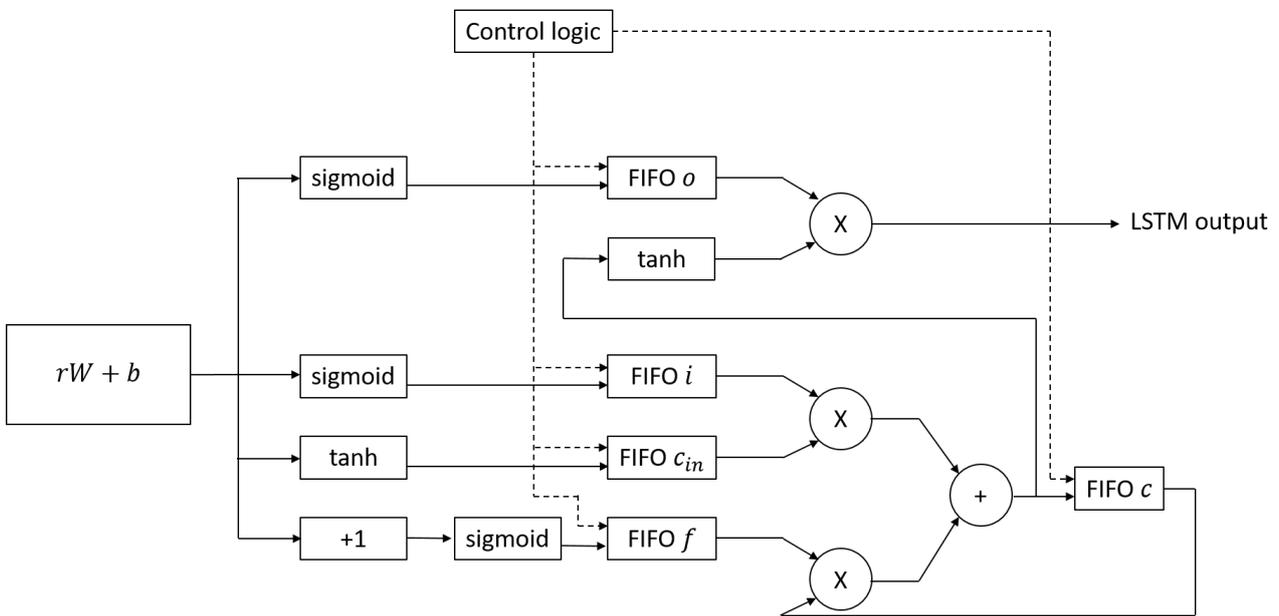


Figure 13 – The architecture to evaluate the LSTM equations. The dashed arrows highlight the control signals.

Each $c^{(t)}$ value is also sent to a tanh block, and the results are multiplied by the content of the FIFO o memory to compute the state of the LSTM cell, as described by eq. (8). The state values are then stored in the FIFO h_1 or in the FIFO h_2 shown in Figure 12, and then the LSTM cell is evaluated.

3.6 The Classification

The Classification phase is mainly based on the fully connected layer and the softmax function. This paragraph details only the softmax function since the fully connected layer description is contained in Section 3.3.

The definition of the softmax function is given by (18):

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{NC} e^{x_k}} \quad (18)$$

where $j = 1, \dots, NC$. Therefore, for the considered case, the softmax function is evaluated three times ($NC = 3$). Thus, the output of this block is an array of three values, each one representing the probability of the time series to belong to a class. As happened for the tanh function, the author did not develop the

softmax function exploiting the Taylor series of the exponential function. Again, the main reason is related to the resource usage for a high order Taylor series implementation to ensure a suitable accuracy.

The authors performed several experiments to efficiently develop the softmax function. The data acquired by an accelerometer as described in [4] are given as input to the network to identify the range of values that reach the softmax functional block. The experiments showed that the exponential input is always in the range $[-6;3]$. Further analysis revealed that input values lower than -3 give as output a number that can be approximated with 0. For this reason, only the results for input values in the range $[-3;3]$ have been stored in a ROM memory. These observations led to a piecewise definition of the exponential function. If the input value is lower than -3 the output is equal to 0, otherwise eq. (18) is evaluated after reading the exponential values from the ROM memory.

The softmax functional block receives as input the three values computed by the previous fully connected layer. These values are stored in a FIFO memory. For each value, the LSTM block evaluates the exponential function and, then, the three results are summed to compute the denominator of eq. (18).

Another critical issue is the quotient evaluation. A hardware divider implies a high resource usage; therefore, to avoid its usage, the authors decided to perform the multiplication between the numerator and the reciprocal of the denominator. This last value can be efficiently computed by exploiting the Newton method, an iterative algorithm that solves the eq. (19):

$$f(x) = x - \frac{1}{d} = 0 \quad (19)$$

finding:

$$x_{i+1} = 2x_i - x_i^2 d \quad (20)$$

The initial value x_0 should be chosen in the range $(0;2/d)$ that is the domain of convergence for the function. Experimental results showed that the optimal initial value is $x_0 = 0.1$. This value has been chosen by comparing the Newton method results with the one computed by Matlab. Several values have been tested and the one with the minimum error is then adopted for the Newton method implementation. Thus, two constant values are defined for the first iteration of the Newton method: $2x_0 = 0.1$ and $x_0^2 = 0.01$. The reciprocal is computed by evaluating ten iterations of the Newton method since experimental results showed that this is the optimal number for the considered range of values.

Finally, the reciprocal value is multiplied by each exponential value to obtain the three output values of the softmax function. Notice that the outputs of the softmax functions are probability values ranging from 0 to 1. The performed experiments showed that the MSE between the fixed and floating point softmax function is about 10^{-4} . Therefore, the introduced error is negligible with respect to the floating point computation.

4. Experimental Results and Discussion

The architecture described in this paper has been developed using the Intel Quartus II environment. To give an exhaustive characterization of the performance, the authors implemented the design into three different FPGAs. These FPGAs had been chosen to cover different families of devices. The first FPGA is a high-end Altera Stratix V 5SGXEA7N2F45C2, the second one is a mid-range Altera Cyclone V 5CEBA7F23C7, while the third is a low-end Altera Cyclone V 5CEFA2F23C6. Table 1 summarizes their main features.

Table 1 – The main features of the three FPGAs.

FPGA	Logic elements	DSP blocks	20 Kbits memory blocks	Pins
Stratix V 5SGXEA7N2F45C2	234,720	256	52 M	1,064

Cyclone V 5CEBA7F23C7	56,480	156	7 M	240
Cyclone V 5CEFA2F23C6	9,430	25	1.8 M	224

To validate the architecture, the authors exploited the Intel DSP Builder, a software to interface the boards with the Matlab environment. It allows importing data from the workspace and transmitting them to the device and, after the elaboration, the tool retrieves the results. In this way, analysis and validation can be performed offline. Testing and validation are performed exploiting the same dataset used in [4]. Data have been converted from the floating-point format to the fixed-point one adopted in the proposed architecture. Matlab performs this conversion, which can be easily implemented by the accelerometer in the real system. The converted data are then transferred to the FPGA hosting the proposed architecture. The transferred data have the size of a single time window, which is 1 second of signals acquisition at the sampling frequency of 100 Hz. The architecture elaborates the time windows and the results are sent back to the Matlab workspace. It is worth noticing that only the results are the probability of the time window to belong to each class. Thus, only three probability values are sent back to Matlab. The fixed-point results are then compared with the floating-point golden reference computed by the original implementation. As already described in Section 3.1, the MSE between the fixed and floating-point results is about 10^{-4} , which is negligible considering that the probability values are always in the range [0;1].

The performance of the three FPGA devices has been evaluated in terms of resource usage, processing time, and power consumption. Moreover, apart from the original architecture, a modified version has also been developed to reduce resource usage. This modified version is developed by removing the softmax functional block from the original architecture. This modification does not affect the correctness of the results, because the softmax layer is used to normalize the results in form of probability values. The classification is performed by the Fully Connected Layer of the Classification part of the system. Therefore, removing the softmax layer has only the effect of not providing normalized results, but the inference, i. e. the classification, is still correct.

Table 2 compares the resource usage of the two versions of the architecture.

Table 2 – Resource usage comparison between the two versions of the architecture for the three FPGA devices.

FPGA	Architecture with softmax				Architecture without softmax			
	Logic Elements	DSP blocks	20 Kbits memory blocks	Pins	Logic Elements	DSP blocks	20 Kbits memory blocks	Pins
Stratix V 5SGXEA7N2F45C2	3262/234720 (1%)	23/256 (9%)	1M/52M (2%)	78/1024 (7%)	3053/234720 (1%)	21/256 (8%)	845K/52M (2%)	78/1024 (7%)
Cyclone V 5CEBA7F23C7	3262/56480 (6%)	26/156 (17%)	1M/7M (15%)	78/240 (33%)	3041/56480 (5%)	24/156 (15%)	845K/7M (12%)	78/240 (33%)
Cyclone V 5CEFA2F23C6	3266/9430 (35%)	25/25 (100%)	1M/1.8M (56%)	78/224 (35%)	3054/9430 (32%)	24/25 (96%)	845K/1.8M (47%)	78/224 (35%)

Table 2 clearly shows that all three devices can host the proposed architectures. As expected, the Stratix V FPGA features the lowest resource usage, while the Cyclone V 5CEFA2F23C6 has the highest one. On the other hand, the resource-saving of the modified architecture has a greater impact on the Cyclone V devices rather than on the Stratix V one. In all the cases, the most used resources are the DSP blocks. This is because the Fully Connected Layers and the LSTM cells are based on multiplications. Finally, the impact of

the ROMs and the FIFOs on the total memory is not negligible only when considering the Cyclone V 5CEFA2F23C6 device.

The performance has also been measured in terms of processing time. For each device and each architecture, the maximum clock frequency was considered. These values are reported in Table 3.

Table 3 – Clock frequency and processing time comparison between the two versions of the architecture for the three FPGA devices.

FPGA	Architecture with softmax		Architecture without softmax	
	Clock frequency [MHz]	Processing times [ms]	Clock frequency [MHz]	Processing times [ms]
Stratix V 5SGXEA7N2F45C2	36	50	52	32
Cyclone V 5CEBA7F23C7	30	55	26	40
Cyclone V 5CEFA2F23C6	24	70	30	55

Considering that the time series lasts 1 second, all the three FPGA devices are real-time compliant. Again, the Stratix V device gives the best performance. In this case, the modification of the architecture gives an interest gain in terms of processing time. Indeed, for the Stratix V device, the gain is about 36%, while for the Cyclone V 5CEBA7F23C7 and the Cyclone V 5CEFA2F23C6 chips it is about 25% and 21%, respectively.

The evaluation of the power consumption required to exploit the PowerPLAY Power Analyzer tool included in Quartus II. The power consumption evaluation is based on the results of a functional simulation, which is needed to estimate the toggle rate. The functional simulation is performed through the Altera Modelsim software. The inputs provided to this tool were waveforms that the authors generated to simulate the data acquired by the accelerometer, together with the clock signal. The outputs produced by this tool are stored in a value change dump (vcd) file, which serves as input for the PowerPlay Power Analyzer Tool. The results in terms of power consumption and dissipated energy are reported in Table 4.

Table 4 – Power consumption and dissipated energy comparison between the two versions of the architecture for the three FPGA devices.

FPGA	Architecture with softmax		Architecture without softmax	
	Power consumption [mW]	Dissipated energy [μ J]	Power consumption [mW]	Dissipated energy [μ J]
Stratix V 5SGXEA7N2F45C2	0.04	2.00	0.06	1.92
Cyclone V 5CEBA7F23C7	0.06	3.30	0.05	2.00
Cyclone V 5CEFA2F23C6	0.05	3.50	0.05	2.75

The Stratix V has the lowest power consumption when considering the architecture with the softmax and the highest one without the softmax functional block. This is probably due to how the logic synthesizer elaborates the design and how the Place and Route tool exploits the FPGA. However, in terms of dissipated energy when considering the Stratix V device, there are no substantial differences between the two versions of the implemented architecture. The energy dissipation analysis for the Cyclone V chips revealed that the architecture without the softmax functional block ensures the best performance. Indeed, the dissipated energy is reduced by 40% for the Cyclone V 5CEBA7F23C7 device and by 20% for the Cyclone V 5CEFA2F23C6 chip.

The analysis of Tables 2, 3 and 4 highlights that all the devices and architectures considered in this work are real-time compliant and meet the power constraint of a wearable device.

Concerning the related works, a direct comparison can be performed between the proposed architecture and the research in [4]. This work exploited the SensorTile device produced by STMicroelectronics. This device is real-time compliant and targets wearable applications. In [4] the same stacked LSTM network of this paper is considered. The implemented inference module takes about 300 ms, with a power consumption of 16.5 mW and dissipated energy of 6290 μ J. Therefore, all the solutions proposed in this paper outperform that work, both in terms of processing time and power consumption. It is interesting to notice that a direct comparison can be carried out only considering the architecture with the softmax layer. Considering the worst case, i.e. the Cyclone V 5CEFA2F23C6 FPGA, the processing is about 4 times faster than in [4] and the dissipated energy is 3 orders of magnitude lower than in the SensorTile implementation.

The development of LSTM networks on FPGA devices has been recently addressed by the literature by the works in [19–25]. In particular, [19] describes a network with 3 LSTM cells and an inner dimension of 256, which is implemented on a Xilinx Virtex 7 FPGA. The authors adopted floating-point arithmetic and declare a processing time of about 390 ms at a working frequency of 150 MHz, with a power consumption of 19.63 W. The work described in [20] implemented 2 LSTM layers with an inner dimension of 1024 on a Xilinx Zedboard Zynq ZC7020 board. The authors reported results only in terms of performance per Watt, which is about 446 Mops/W in the best case. Moreover, the data are represented in a fixed-point format with 8 bits for the integer part and 8 bits for the decimal part. Another interesting work [21] reported the implementation of two LSTM cells with an inner dimension of 1024 on a Xilinx XCZU6EG FPGA. The data are represented in fixed-point with 16 bits, of which 10 are used for the integer part. The authors obtained a working frequency of 238 MHz. The power consumption is about 885 mW. Another interesting work, [22] described the implementation of a single LSTM cell with an inner dimension of 64 on different Xilinx FPGA. The highest working frequency is achieved by the Xilinx XCZU6EG FPGA and is about 385 MHz, while the lowest power consumption is given by the Xilinx XC7Z020 FPGA and is about 280 mW. The work in [23] is focused on providing a parallel and power efficient LSTM network on FPGA. In this case, a single LSTM cell with an inner size of 128 has been developed. The best implementation worked at 118 MHz, taking about 18 ms to elaborate 50 samples and consuming 1.86 mW. Author of [24] developed a system on chip (SoC) which includes a Xilinx FPGA and an ARM CPU. The work considered five LSTM cells with an inner dimension of 64. This network took 3.5 ms and consumes 3.6 W to classify the input signals. Finally, [25] presented an FPGA implementation of a single LSTM cell with an inner dimension of 32. The processing time was 0.11 ms at the working frequency of 200 MHz and with a power consumption of 1.52 W.

It is worth noticing that a direct comparison between the work proposed in this paper and the researches in [19–25] is not fair, since the literature mainly focuses on the parallelization of the LSTM operations. The novelty of the proposed solution is in the fact that despite the working frequency is lower than the ones reported in the literature, it is real-time compliant for the considered case study. Of course this brings a significantly lower power consumption than in [19–25].

Table 5 summarizes the comparison between the proposed work and the research described in [4] and in [19–25].

Table 5 – Comparison between the proposed solution and the state of the art.

Work	Device	LSTM network features		Processing time [ms]	Power consumption [mW]
		Number of layers	Inner dimension		
[4]	STM SensorTile	2	32	300	16.5

[19]	a Xilinx Virtex 7 FPGA	3	256	390	19630
[20]	Xilinx Zedboard Zynq ZC7020 SoC	2	1024	N.A.	N.A.
[21]	Xilinx XCZU6EG FPGA	2	1024	N.A.	885
[22]	Xilinx XCZU6EG FPGA	1	64	N.A.	280
[23]	Xilinx Zynq-7020 FPGA	1	128	18	1.86
[24]	Xilinx ZCU104 SoC	5	64	3.50	3600
[25]	Xilinx Virtex 7 FPGA	1	32	0.11	1520
Proposed work	Stratix V 5SGXEA7N2F45C2	2	32	32	0.06

5. Conclusion

This paper describes a hardware architecture based on the stacked LSTM network described in [4]. The network includes a pre-processing phase, two LSTM cells and a classification stage. The main functional blocks developed to implement the network are the fully connected layer and the ReLU, softmax and tanh non-linear functions.

The architecture was designed for wearable devices; therefore, special attention was given to power consumption and resource usage. Moreover, processing time constraints were taken into account.

This paper presents two different architecture versions implemented on three FPGA devices. All the versions have been validated on all the devices exploiting the dataset described in [4]. The conducted experiments showed that the two architectures are real-time compliant on all the devices. Moreover, they featured a low power consumption, which is suitable for wearable devices.

The authors compared the proposed work with the solution in [4], which exploited the SensorTile device. The proposed solution features a processing time which is about one order of magnitude lower than the one reported in [4]. Concerning the power consumption, it is reduced by about three orders of magnitude.

The other works exploiting LSTM networks on FPGA devices [19–25] focused on parallelization to reduce the processing time. Therefore, direct comparisons between these works and the proposed one are not completely fair. Nevertheless, it is worth noticing that the power consumption featured by the proposed architecture is significantly lower than the ones reported in the literature and even the computing times. Only [24,25] take less time than the proposed solution (that in any case remains real time) but at the price of a hugely higher power consumption.

As a future research line, the proposed architecture can be extended in order to consider data fusion from different sensors. The main parts of the architecture will be used as main blocks for this new modular architecture, extending them with new features for the data fusion. In addition, hardware architectures for other kinds of RNNs will be designed.

6. References

- [1] A.D. Orjuela-Cañón, A. Cerquera, J.A. Freund, G. Juliá-Serdá, A.G. Ravelo-García, Sleep apnea: Tracking effects of a first session of CPAP therapy by means of Granger causality, *Comput. Methods Programs Biomed.* 187 (2020) 105235. <https://doi.org/10.1016/j.cmpb.2019.105235>.

- [2] P. Pierleoni, A. Belli, O. Bazgir, L. Maurizi, M. Paniccia, L. Palma, A Smart Inertial System for 24h Monitoring and Classification of Tremor and Freezing of Gait in Parkinson's Disease, *IEEE Sens. J.* 19 (2019) 11612–11623. <https://doi.org/10.1109/JSEN.2019.2932584>.
- [3] L.M. Posthuma, C. Downey, M.J. Visscher, D.A. Ghazali, M. Joshi, H. Ashrafian, S. Khan, A. Darzi, J. Goldstone, B. Preckel, Remote wireless vital signs monitoring on the ward for early detection of deteriorating patients: A case series, *Int. J. Nurs. Stud.* 104 (2020). <https://doi.org/10.1016/j.ijnurstu.2019.103515>.
- [4] E. Torti, A. Fontanella, M. Musci, N. Blago, D. Pau, F. Leporati, M. Piastra, Embedded real-time fall detection with deep learning on wearable devices, in: *Proc. - 21st Euromicro Conf. Digit. Syst. Des. DSD 2018, Prague, 2018*: pp. 405–412. <https://doi.org/10.1109/DSD.2018.00075>.
- [5] Y. Nizam, M.M.A. Jamil, Classification of Daily Life Activities for Human Fall Detection: A Systematic Review of the Techniques and Approaches, in: *Stud. Syst. Decis. Control*, Springer, 2020: pp. 137–179. https://doi.org/10.1007/978-3-030-38748-8_7.
- [6] S. Khan, R. Qamar, R. Zaheen, A.R. Al-Ali, A. Al Nabulsi, H. Al-Nashash, Internet of things based multi-sensor patient fall detection system, *Healthc. Technol. Lett.* 6 (2019) 132–137. <https://doi.org/10.1049/htl.2018.5121>.
- [7] T.H. Nguyen, T.T. Nguyen, B.V. Ngo, A SVM algorithm for falling detection in an IoTs-based system, in: *Intell. Syst. Ref. Libr.*, Springer Science and Business Media Deutschland GmbH, 2020: pp. 139–170. https://doi.org/10.1007/978-3-030-23983-1_6.
- [8] T. Mauldin, M. Canby, V. Metsis, A. Ngu, C. Rivera, T.R. Mauldin, M.E. Canby, V. Metsis, A.H.H. Ngu, C.C. Rivera, SmartFall: A Smartwatch-Based Fall Detection System Using Deep Learning, *Sensors.* 18 (2018) 3363. <https://doi.org/10.3390/s18103363>.
- [9] G. Danese, M. Giachero, F. Leporati, N. Nazzicari, An embedded multi-core biometric identification system, *Microprocess. Microsyst.* 35 (2011) 510–521. <https://doi.org/10.1016/j.micpro.2011.03.003>.
- [10] E. Torti, D. Koliopoulos, M. Matraxia, G. Danese, F. Leporati, Custom FPGA processing for real-time fetal ECG extraction and identification, *Comput. Biol. Med.* 80 (2017) 30–38. <https://doi.org/10.1016/j.compbio.2016.11.006>.
- [11] E. Torti, C. D'Amato, G. Danese, F. Leporati, An Hardware Recurrent Neural Network for Wearable Devices, in: *Proc. - 24th Euromicro Conf. Digit. Syst. Des. DSD 2020, Institute of Electrical and Electronics Engineers (IEEE), 2020*: pp. 293–300. <https://doi.org/10.1109/dsd51259.2020.00055>.
- [12] R. Mouleeshuwarappabu, N. Kasthuri, Nonlinear vector decomposed neural network based EEG signal feature extraction and detection of seizure, *Microprocess. Microsyst.* 76 (2020) 103075. <https://doi.org/10.1016/j.micpro.2020.103075>.
- [13] B. Chettri, T. Kinnunen, E. Benetos, Deep Generative Variational Autoencoding for Replay Spoof Detection in Automatic Speaker Verification, *Comput. Speech Lang.* 63 (2020) 101092. <https://doi.org/10.1016/j.csl.2020.101092>.
- [14] H. Fabelo, M. Halicek, S. Ortega, M. Shahedi, A. Szolna, J. Piñeiro, C. Sosa, A. O'Shanahan, S. Bisshopp, C. Espino, M. Márquez, M. Hernández, D. Carrera, J. Morera, G. Callico, R. Sarmiento, B. Fei, Deep Learning-Based Framework for In Vivo Identification of Glioblastoma Tumor using Hyperspectral Images of Human Brain, *Sensors.* 19 (2019) 920. <https://doi.org/10.3390/s19040920>.
- [15] E. Torti, A. Fontanella, A. Plaza, J. Plaza, F. Leporati, Hyperspectral Image Classification Using Parallel Autoencoding Diabolo Networks on Multi-Core and Many-Core Architectures, *Electronics.* 7 (2018) 411. <https://doi.org/10.3390/electronics7120411>.
- [16] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.

- [17] R. Jozefowicz, W. Zaremba, I. Sutskever, An empirical exploration of recurrent network architectures, in: ICML'15 Proc. 32nd Int. Conf. Mach. Learn. - Vol. 37, 2015: pp. 2342–2350.
- [18] M. Hermans, B. Schrauwen, Training and Analysing Deep Recurrent Neural Networks, in: C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, K.Q. Weinberger (Eds.), *Adv. Neural Inf. Process. Syst.* 26, Curran Associates, Inc., 2013: pp. 190–198. <http://papers.nips.cc/paper/5166-training-and-analysing-deep-recurrent-neural-networks.pdf>.
- [19] Y. Guan, Z. Yuan, G. Sun, J. Cong, FPGA-based accelerator for long short-term memory recurrent neural networks, in: *Proc. Asia South Pacific Des. Autom. Conf. ASP-DAC*, Institute of Electrical and Electronics Engineers Inc., 2017: pp. 629–634. <https://doi.org/10.1109/ASPDAC.2017.7858394>.
- [20] A.X.M. Chang, E. Culurciello, Hardware accelerators for recurrent neural networks on FPGA, in: *Proc. - IEEE Int. Symp. Circuits Syst.*, Institute of Electrical and Electronics Engineers Inc., 2017. <https://doi.org/10.1109/ISCAS.2017.8050816>.
- [21] K. Chen, L. Huang, M. Li, X. Zeng, Y. Fan, A Compact and Configurable Long Short-Term Memory Neural Network Hardware Architecture, in: *Proc. - Int. Conf. Image Process. ICIP*, IEEE Computer Society, 2018: pp. 4168–4172. <https://doi.org/10.1109/ICIP.2018.8451053>.
- [22] E. Bank-Tavakoli, S.A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, M. Pedram, POLAR: A Pipelined/Overlapped FPGA-Based LSTM Accelerator, *IEEE Trans. Very Large Scale Integr. Syst.* 28 (2020) 838–842. <https://doi.org/10.1109/TVLSI.2019.2947639>.
- [23] U. Yoshimura, T. Inoue, A. Tsuchiya, K. Kishine, Implementation of Low-Energy LSTM with Parallel and Pipelined Algorithm in Small-Scale FPGA, in: *Int. Conf. Electron. Information, Commun.*, Institute of Electrical and Electronics Engineers (IEEE), Jeju, Korea (South), 2021: pp. 1–4. <https://doi.org/10.1109/iceic51217.2021.9369806>.
- [24] J. Yin, J. Han, R. Xie, C. Wang, X. Duan, Y. Rong, X.Y. Zeng, J. Tao, MC-LSTM: Real-time 3D Human Action Detection System for Intelligent Healthcare Applications, *IEEE Trans. Biomed. Circuits Syst.* (2021). <https://doi.org/10.1109/TBCAS.2021.3064841>.
- [25] W. Zhang, F. Ge, C. Cui, Y. Yang, F. Zhou, N. Wu, Design and Implementation of LSTM Accelerator Based on FPGA, in: *Int. Conf. Commun. Technol. Proceedings, ICCT*, Institute of Electrical and Electronics Engineers Inc., 2020: pp. 1675–1679. <https://doi.org/10.1109/ICCT50939.2020.9295665>.