Microprocessors and Microsystems 71 (2019) 102895

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Embedding Recurrent Neural Networks in Wearable Systems for Real-Time Fall Detection

Emanuele Torti[a,*], Alessandro Fontanella[a], Mirto Musci[a], Nicola Blago[a], Danilo Pau[b], Francesco Leporati[a], Marco Piastra[a]

[a] Department of Electrical, Computer and Biomedical Engineering University of Pavia, Pavia, Italy I-27100
[b] Advanced System Technology, STMicroelectronics, Agrate Brianza, Italy

ARTICLE INFO

Article history:
Received 24 January 2019
Accepted 13 September 2019
Available online 17 September 2019

Keywords:
Fall detection
Embedded systems
Deep Learning
Recurrent Neural Networks
Wearable devices

ABSTRACT

Accidental falls are the preminent cause of fatal injuries and the most common cause of nonfatal trauma-related hospital admissions among elderly adults. An automated monitoring system that detects occurring falls and issues remote notifications will prove very valuable to improve the level of care that could be provided to vulnerable people. The paper focuses on the wearable-device approach to real-time fall detection, and presents the design of an embedded software for wearable devices that are connected in wireless mode to a remote monitoring system. In particular, the work proposes the embedding of a *recurrent neural network* (RNN) architecture on a micro controller unit (MCU) fed by tri-axial accelerometers data recorded by onboard sensors. To address the feasibility of such resource-costrained deep learning approach, the paper presents a few general formulas to determine memory occupation, computational load and power consumption. The formulas have been validated with the implementation of the run-time detection module for the SensorTile® device by STMicroelectronics.

1. Introduction

Accidental falls are the preminent cause of fatal injuries and the most common cause of nonfatal trauma-related hospital admissions among elderly adults. Even when falls are less severe, the associated disconfort, especially when no immediate care can be given, significantly lessens the quality of life of vulnerable subjects. Accidental falls are more common as the age increases: more than 25% of people aged over 65 years old falls every year; this figure grows to 32%–42% for those over 70 [1].

Falls are very common in domestic scenarios and are often uncared which leads to immediate disconfort and long-term consequences especially with patients living without daily care. People living in long-term care institutions, nearly 30%–50% of them fall each year, with almost half of them experiencing recurrent falls. Moreover, elderly people are not the only group that is heavily affected by unintentional falls: any person with some sort of vulnerability, such as post-operative patients or people with disabilities, is part of similar statistics. Overall, falls lead to 20%–30% of mild to severe injuries and 40% of all injury deaths. The average cost of a single hospitalization for fall-related injuries in 65 years old people reached $17,483 in the US in 2004, with a forecast to $240 billion of total costs by 2040.

In such scenario, an automated monitoring system that can promptly detect falls as they occurs and issue a remote notification, could be extremely valuable for improving the level of care for people with fragilities of any sort, from both a clinical and a psychological point-of-view. Since there is no way to predict when or where a fall may occurr, such system should ideally run in real-time and continuously in a 24×7 scenario.

There are many strategies and designs to implement either an ambient based or sensor-based automatic fall detection system. Among the latter, two main approaches have been considered: smartphone-based and wearable-devices-based. Recent trends in the fall detection literature [2] point out that embedded wearable devices are the most interesting choice since they are generally less intrusive and less expensive than ambient-based solutions and are more power-effective with respect to smartphone-based solutions [3]. Overall, these features increase the level of acceptance and allow for ubiquitous and continous monitoring.

The overall aim of this paper is to study the design of the embedded software component of a wearable-based fall detection system. The software will be tailored for devices equipped with a micro-controller unit (MCU) and will govern sensory data

* Corresponding author.
E-mail addresses: emanuele.torti@unipv.it (E. Torti), alessandro.fontanella01@universitadipavia.it (A. Fontanella), mirto.musci@unipv.it (M. Musci), nicola.blago01@universitadipavia.it (N. Blago), danilo.pau@st.com (D. Pau), francesco.leporati@unipv.it (F. Leporati), marco.piastra@unipv.it (M. Piastra).

https://doi.org/10.1016/j.micpro.2019.102895
0141-9331/

acquisition, run algorithms for real-time fall detection and manage the remote communication process to issue alert notifications towards a remote monitoring system.

In our analysis, the overall design of wearable sensory devices for fall detection should fulfill the following three basic requirements:

(1) be at all times connected in wireless mode to a remote, cloud-based system that will ensure the proper notification of alerts to caregivers;
(2) be as small and lightweight as possible, to minimize the inconvenience to the wearer and increase acceptance;
(3) be running on battery for as long as possible without recharge, as the monitoring activity needs to be performed 24×7.

Obviously, these general requirements entail several software design constraints that should be carefully considered. First of all, sensory data should be processed locally, i.e. on the device itself, as their complete transmission to a remote end would involve a dramatic increase in power consumption [4]. Consequently, algorithms for real-time fall detection should be explicitly designed and implemented for low-power MCUs and thus take into-account severe limitations in both memory occupancy and computational power.

Generally speaking, wearable-based fall detection relies on the real-time analysis on either or both tri-axial accelerometer and gyroscope signals collected by the onboard sensors. In the literature, real-time fall detection from accelerometer data is still considered an open research problem [5].

Traditionally, such analysis has been carried-out with threshold-based algorithms and classical machine learning with good overall results [6]. As described in Section 2.2, deep learning techniques have been recently taken into account in developing fall detection algorithm and showed very promising potential [4,7,8].

However, to the best of our knowledge, no studies exist about the applicability of deep learning techniques to online fall detection on embedded devices, with the exception of the preliminary work by Musci et al. [9] which should be considered as a companion to the present paper.

In this paper we investigate the MCU embedding of a particular class of DL networks, namely *recurrent neural networks* (RNNs), for real-time fall detection. The main contribution thus obtained is the abstraction of a set of formulas for evaluating the requirements in terms of memory, computing power and power consumption for the embedding of a generic RNN architecture on an MCU. Thus the proposed analysis assesses which kind of trade-offs should be taken into account while porting a DL-based solution to an embedded device.

The abstracted formulas are subsequently validated through an experimental procedure involving two main software components:

- a workstation-based TensorFlow [10] complete implementation of a reference RNN architecture for training and detection [9];
- a specific implementation of the run-time detection module for the same RNN architecture adapted to the SensorTile® device by STMicroelectronics, which includes an ARM® Cortex® M4 (i.e. STM32) MCU.

The implementation of the run-time detection module is the second main contribution of the paper. As stated before, such module must perform its computation directly on the embedded device, unlike the state of the art solutions presented in Section 2.1.

As it will be described below, the training phase of the proposed RNN architecture has been performed on a workstation with the well-known SisFall dataset [11], while the resulting binary models were transferred to the embedded component for the final validation. Several implementation variants, in particular for the numerical encoding of binary RNN models, were also considered [12]. The companion paper [9] presents all the details of the workstation implementation and training.

The proposed activity lies in the wake of previous activities from the authors, such as embedded and wearable devices implementations [13–16].

The remainder of the paper is organized as follows: Section 2 provides a description of the state of art for the relevant context, in particular for automated fall detection methods, deep learning techniques and the software/hardware design of low-power, wearable sensory devices; Section 3 describes the feasibility analysis performed and the resulting abstract formulas for the embedding of RNN architectures; Section 4 explores the design space, while Section 5 details the network optimization. The performed experiments are contained in Section 6; finally, Section 7 depicts the conclusions and future work.

2. Context

2.1. Wearable devices

Wearable devices are configurations of miniaturized electronic components including both sensors and a processing unit that can be worn by an user under, with or on top of clothing. Being designed for the specific purpose, such devices are better suited than a smartphone as they do not suffer from the typical limitations of the latter, such as the fact that sensors are shared devices managed by the operating system, possibly running multiple applications [17].

Several wearable devices have been proposed in the fall detection literature. Nyan et. al [18] proposed a system using accelerometers and gyroscopes. In this approach, sensors are connected via Zigbee transceivers to a board based on an Intel PXA255 processor, where the actual processing takes place. Jung et al. in [19] describe a configuration made of a three-axial accelerometer, a MicroController Unit (MCU) and a Bluetooth module. All these components are attached to a jacket and are connected with each other via stretchable conductive nylon. In another system [20], an integrated board carries the accelerometers and an FPGA for signal processing. Cola et al. [21] developed a head-worn device containing an accelerometer and a barometer integrated with a TI MSP430 MCU. From a software standpoint, all the fall detection methods above are executed on the devices themselves by making use of a predefined threshold applied to either raw signals or statistical indicators.

More sophisticated methods are adopted in other approaches but at the cost of a remote processing that takes place on a connected workstation. The approach described in [22] is based on the SHIMMER (Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability) integrated sensor platform [23]. In this case, the remote processing of accelerometer signals is performed through Support Vector Machine (SVM); the authors also compare the latter method with k-Nearest Neighbors (kNN) and Complex Trees. A similar approach is described in [8], which performs offline elaboration of signals acquired by accelerometers and gyroscopes. In these cases, data are analyzed through machine learning approaches.

Power consumption is a critical design goal that has been addressed by many authors. In particular authors of [24] designed a custom board with the specific goal of minimize power consumption in a IoT-based fall detection system. However, they rely on remote communication to run an external fall detection algorithm.

Apart from the above-mentioned SHIMMER platform, other COTS devices are emerging at present. Among those, the SensorTile miniaturized board produced by STMicroelectronics has interesting characteristics in terms of low power consumption,

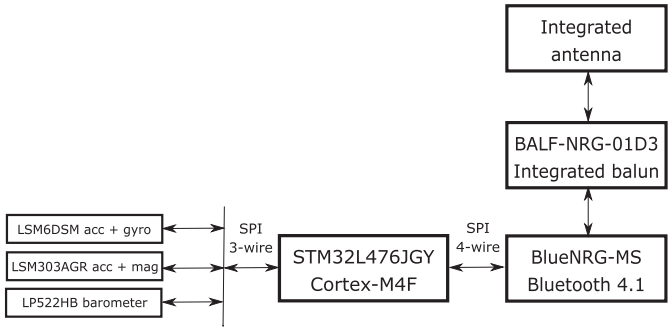

Fig. 1. SensorTile architecture.

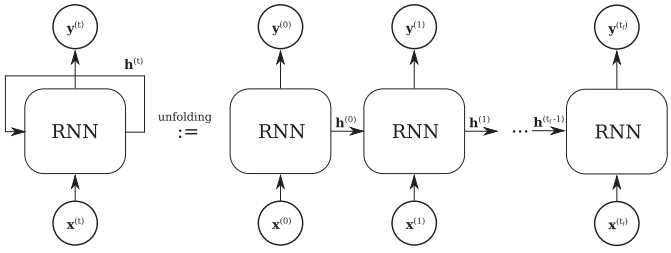

Fig. 2. A single RNN cell (on the left) is temporally unfolded for training. The depth of unfolding is also the depth of the network.

memory and computational capabilities. The overall architecture of the SensorTile is shown in Fig. 1. The device is equipped with an STM32L476JGY MCU with a maximum working frequency of 80 MHz. It also integrates two three-axial accelerometers, a gyroscope, a magnetometer and a barometer. It also integrates a Bluetooth 4.1 module, a popular technology for IoT devices. The on-board MCU is an ARM® Cortex®-M4 core featuring a Floating Point Unit (FPU) which fully supports single-precision data-processing instructions and data types. The MCU also implements a full set of DSP instructions. It is equipped with 1 MB of flash memory and 128 KB of SRAM memory. Usage of the SensorTile board in a wearable device was proposed in [25] for human activity recognition.

2.2. Deep learning for fall detection

Recently, Deep Learning (DL) has quickly become the most promising and discussed research paradigm, by the scientific community. DL techniques have been proved to outperform an increasing number of established algorithms in the more disparate fields, from typical computer vision problems, such as person re-identification or object recognition, to sequence-to-sequence language translation, and are inducing new approaches in unusual domains such as robotics, arts, gaming, etcetera [26].

The general consensus is that such impressing developments have been possible for two main reasons: the continous release of datasets of suitable size on which to train complex networks, and the easy availability of computing systems capable of sustaining the workload involved with training deep networks. Such considerations also hold true for the fall detection domain.

Recurrent Neural Networks (RNNs) are a particular form of deep neural networks [26] in which a part of the output produced is fed back to the input, as shown in Fig. 2 on the left. While Convolutional Neural Network (CNN) [26] allows to capture spatial correlation in the data, the recurrence in RNNs allows to capture temporal correlations.

More formally a Recurrent Neural Network (RNN) can be described by:

$$\mathbf{y}^{(t)} := \mathbf{w}g(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{b}) + \mathbf{c} \tag{1}$$

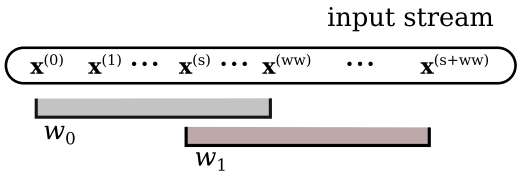

Fig. 3. Graphical representation of a sliding window w_i, with width *ww* and stride *s* ($s < ww$) during the inference process of a typical RNN fed from a generic input data stream.

where

$$\mathbf{h}^{(t)} := g(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{b}).$$

In this representation, $\mathbf{x}^{(t)}$ and $\mathbf{y}^{(t)}$ are, respectively, the input and output at time *t*; *g* is a non-linear function, as it will be described below, and **w**, **W**, **U**, **b** and **c** are the parameters. $\mathbf{h}^{(t)}$ is usually referred to as the *hidden state* of the network [26].

Due to recurrence, RNNs are on the one hand most suited for analyzing time series of signals yet, on the other hand, they are very difficult to train in general. The typical technique adopted for RNN training is that of *temporal unfolding* as shown in Fig. 2, in which each training input sequence of pre-defined length *n* is fed in input of *n* replicas of the base architecture. The parameter optimization process takes place via *stochastic gradient descent* over a *loss function* measured on a training dataset of labelled input sequences.

RNN are mainly used to process data in form of sequences and lists, especially signals that vary over time. Real-time data from accelerometer and gyroscope sensors clearly falls in this category; typically, in this case, the input stream of data will be scanned through a *sliding window* of a width *ww* that matches the predefined level of unfolding for training (see above).

The objective of training is producing the (possibly) optimal set of the above parameters. In a practical real-time application, once training is complete, an RNN of this kind will be applied to an input stream of signals, by sliding a window of the same width *ww* over the stream and resetting and re-running the RNN on each input window. This process is called *inference* since it aims to recognize specific patterns in input sequence. To reduce the computational burden in inference, the input window w_i is typically slid at intervals $s >> 1$ of constant length called *strides* (see Fig. 3).

As seen in Fig. 2, temporally unfolding is what makes RNNs to be trained as if they were deep networks; this makes it possible to apply to the same training process all the techniques that become popular with DL [26], while still keeping inference as a relatively inexpensive process from a computation standpoint.

Long Short-Term Memory (LSTM) [27,28] are a special kind of RNN, capable of learning long-term temporal dependencies (see Fig. 4), since they feature the capability to learn how to 'forget' and filter part of their hidden state during the inference process. In spite of the extra complexity added in their structure, LSTM become easier to train with the temporal unfolding approach because they do not incur in the so-called *vanishing gradient problem*.

Fig. 4 shows a LSTM *cell*. At each time step *t* the cell is fed by the concatenation of $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t-1)}$. The difference with the RNN above is that the input is distributed over four *gates*: *cell gate* **c**, *input gate* **i**; *output gate* **o** and *forget gate* **f**. Each gate has a structure similar to Eq. 1, and is described by a parameter matrix **W** and a bias vector **b** and a non-linear function which is typically a sigmoid or a hyperbolic tangent (like in our case).

Formally, a single LSTM cell is described by the following equations:

$$\mathbf{c}_{\text{in}}^{(t)} = \tanh(\mathbf{W}_{xc}\mathbf{x}^{(t)} + \mathbf{W}_{hc}\mathbf{h}^{(t-1)} + \mathbf{b}_c) \tag{2}$$

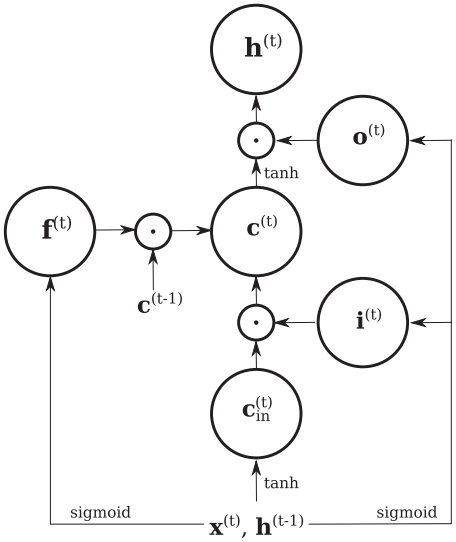

Fig. 4. The basic structure of an LSTM cell. Each circle represents an equation in the mathematical formulation (Eqs. 2–7). For each arrow pointing to a circle, an addition is performed. Dots represent vector/matrix multiplications.

$$\mathbf{i}^{(t)} = \text{sigmoid}(\mathbf{W}_{xi}x^{(t)} + \mathbf{W}_{hi}h^{(t-1)} + \mathbf{b}_i) \tag{3}$$

$$\mathbf{o}^{(t)} = \text{sigmoid}(\mathbf{W}_{xo}\mathbf{x}^{(t)} + \mathbf{W}_{ho}h^{(t-1)} + \mathbf{b}_o + \mathbf{b}_{\text{forget}}) \tag{4}$$

$$\mathbf{f}^{(t)} = \text{sigmoid}(\mathbf{W}_{xf}\mathbf{x}^{(t)} + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{5}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)}\mathbf{c}^{(t-1)} + \mathbf{i}^{(t)}\mathbf{c}^{(t)}_{\text{in}} \tag{6}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \tanh \mathbf{c}^{(t)} \tag{7}$$

where

$\mathbf{W} \in \mathbb{R}^{LS \times LS}$

$\mathbf{b}, \mathbf{x}^{(t)}, \mathbf{h}^{(t)}, \mathbf{c}^{(t)}_{in}, \mathbf{i}^{(t)}, \mathbf{o}^{(t)}, \mathbf{f}^{(t)}, \mathbf{c}^{(t)} \in \mathbb{R}^{LS}$.

LS is called the LSTM *size* and is an *hyperparameter*, which is defined upfront by design, and is constant among all cells.

In Eq. 4, $\mathbf{b}_{\text{forget}}$ is also called *forget bias* and has been made evident because it is critical to improve LSTM performance as described in [29]. $\mathbf{b}_{\text{forget}}$ is typically set equal to 1.

Several DL solutions for fall detection have been presented in the literature in recent years. In [7], a CNN followed by a RNN layer is proposed, where the CNN extracts features from sensor signals, while the RNN detects a temporal relationship among the extracted features. Detection, however, is computationally expensive and is performed on a workstation. In [30] an offline DL solutions is shown to outperform statistical and Machine Learning (ML) methods in the wrist-based Fall Detection (FD). In [31], CNN, LSTM and ConvLSTM-based architectures are compared for the closely related problem of fall prediction [32]. Similarly, the authors of [33] use RNN techniques to study gait difference between young and elder subjects.

To the best of our knowledge, all proposed DL-based fall detection systems belong to one of two categories (with the relevant exception of [9]), both unsuitable for an embedded online implementation. They either learn a complex model with millions of parameters (such as [7]); or they rely on remote communication to a workstation for processing. In the first case, even a simple inference model is too large and/or complex for an embedded implementation. In the second case, the cost of remote communication drains the battery too quickly for a 24/7 application [24].

2.3. Deep learning for embedded system

Recently, DL has become to emerge as a viable approach on embedded devices. For example, the most widespread and complete software frameworks for DL, TensorFlow [10], is now available on both mobile and high-end embedded devices in the form of TensorFlow Lite[1].

In general, training an end-to-end deep network on a MCU is very challenging, as the computational cost of such endeavor is often unsustainable. However, when limiting the embedding to the inference module only it becomes much more practical.

Nevertheless, the limitations in computational power and memory require careful design and adaptation of DL architectures, together with a MCU specific implementation, in order to achieve reasonable performances.

A first and obvious strategy to make such implementation feasible is to prune the training model, by removing all parameters that are required for training but ineffective for inference, such as all parameters required by stateful optimizers as Adam [10].

More sophisticated strategies described in the literature are parameter compression [34], which includes weight pruning and quantization, weight matrix approximation [35], weight clustering [36] and fixed-point representation [37].

In particular, fixed-point representation is an attractive strategy as it allows reducing the memory occupancy of the parameters, albeit at the expense of precision. From the computation standpoint, we need to distinguish between linear algebra operations and non-linear functions, as for instance in Eqs. 2–7. In fact, as it will be discussed below, addition and multiplication in fixed point are generally performed in a more efficient way, whereas the direct computation of hyperbolic tangents and sigmoids in fixed point format leads to substantial losses in precision with respect to the floating point counterpart. The ReLU non-linearity, which is widely employed in the DL community, does not suffer from such loss in precision; however, LSTM-based architectures are in general unsuited for the usage of ReLU, unless special provisions are adopted [38].

Integer quantization is another, even more aggressive strategy for reducing memory occupancy as it could be based on 8-bit integers [37]. This strategy involves defining a range of values for both parameters and intermediate variables and then using an integer-based encoding strategy, tipically linear, for translating back and forth floating-point values into such range. Integer quantization strategy, however, is considered to be still an open research topic since it involves careful implementation of linear algebra operations and, in most practical approaches, non-linear functions are implemented by first decoding and then re-encoding values into the floating point representation.

This approach could be brought to the extreme with ultra-low precision weights, such as binary $(-1/+1)$ or ternary $(-1/0/+1)$ representations [39,40]. The latter approaches, however, require to re-implement the whole network structure, and may lead to significant loss in precision.

Widespread software support for such solutions is only expected to increase in the future. For instance, the latest Tensorflow releases allow for an almost transparent conversion from a fully-fledged model trained on a workstation, to an optimized and weight quantized light model for Tensorflow-Lite.

[1] https://www.tensorflow.org/lite/.

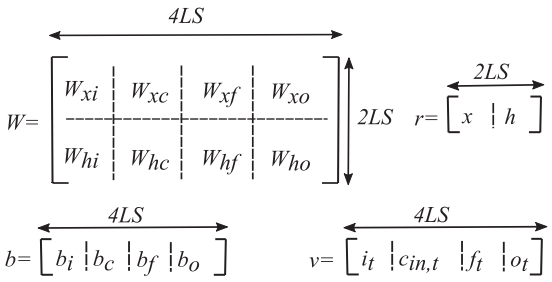

Fig. 5. Schema of data organization.

However, as the capabilities of the device decrease, such as in the case for MCUs, the support from publicly available framework decreases too, to the point that, as in the present case, custom built software is the only viable solution.

3. Feasibility analysis

3.1. Optimizations

In the embedded implementation of LSTM cells on MCUs, the design of the data storage layout has a critical importance, because this influences both computations and memory occupancy. In particular, a suitable data storage layout can increase the efficiency with which the operations involved in Eqs. 2–7 are carried out. Fig. 5 depicts the data layout adopted in this work. The main advantage of such layout is that, due to the stacking of the sub-matrices and values in W and b, the i_t, the functions $c_{in,t}$, f_t and o_t can be evaluated in just one matrix multiplication followed by an addition and the result is stored in the v array:

$$v := W \cdot r + b.$$

Then, the $\mathbf{b}_{\text{forget}}$ term is added just to the part of the v vector representing the o_t elements. The non-linear functions sigmoid and tanh are applied element-wise to the four parts of the v vector on the basis of their position (see Eqs. 2–7). After that, c_t is evaluated using another temporary vector of dimension LS (not shown in figure). It is of crucial importance that the memory allocated for the v and r vectors can be reused for different LSTM cells to be computed in cascade, while the W matrix and the b vector require a memory allocation which is private to each cell.

It is also important to consider the numerical representation of data as mentioned in Section 2.3. In the considered application, it is mandatory to adopt single precision floating point arithmetic, because integer (or fixed) arithmetic would result in significant precision losses. Moreover, ARM Cortex-M family can exploit the ARM CMSIS library[2] which includes optimized floating point routines.

3.2. General metrics

Considering the proposed data layout, we can now introduce three specific metrics about memory occupancy, computational complexity and power consumption for the implementation of each LSTM cell on an MCU. Such metrics are general because they rely on parameters describing the values specific to a particular MCU. They have a structure which is invariant across multiple MCUs.

[2] https://developer.arm.com/embedded/cmsis.

The memory occupancy required by an LSTM cell can be estimated using the following formula:

$$Size_{Net} = In_{El} \cdot Size_{In} + \sum_{i=1}^{i=N_W} W_{El,i} \cdot Size_{W,i} + \sum_{j=1}^{j=N_b} b_{El,j} \cdot Size_{b,j} + \sum_{k=1}^{k=N_T} T_{El,k} \cdot Size_{T,k} + Out_{El} \cdot Size_{Out} \quad (8)$$

where In_{El} is the total number of input values, $Size_{In}$ is the number of bytes used for input representation, $W_{El,i}$ is the element number of the i^{th} W matrix, which is represented using $Size_{W,i}$ bytes. Moreover, $b_{El,j}$ represents the element number of the j^{th} b vector; the number of temporary k arrays are taken into account by the term $T_{El,k}$, together with their representation $Size_{T,k}$ in bytes. Finally, Out_{El} is the number of output elements, represented using $Size_{Out}$ bytes.

Estimating the computational complexity of an LSTM cell is more complex since the numerical representation of choice strongly affects the definition of the metric itself. In the following, we make the assumption that 32 bits floating point format is adopted. In particular, the adoption of either fixed point format or integer quantization requires each a specific metric; although not presented here, such metrics can be developed along the same lines presented below.

Operations involved in $c_{in,t}$ require a number of floating point operations (FLOP) which can be estimated as $\approx 4LS^2 + 22LS$ ($4LS^2$ FLOP for the linear algebra operations and $22LS$ FLOP for the tanh evaluations). The i_t and o_t computations are similar; the only difference is that in the latter cases the tanh function is replaced with the sigmoid, which requires approximately 15 FLOP and is evaluated LS times. Therefore, the total number of FLOP required for each Eqs (1–4) is $\approx 4LS^2 + 15LS$. In addition, evaluating the f_t function also involves the $\mathbf{b}_{\text{forget}}$ term, which takes LS sums, for a total weight of $4LS^2 + 16LS$. Moreover, c_t requires an extra $3LS$ FLOP since each element needs 2 multiplications and a sum, while h_t requires an extra $\approx 23LS$ FLOP since each element needs a multiplication and a tanh.

To compute the overall computational complexity it is important to keep in mind that each LSTM cell must repeat all the above operations for each sample in the input set of dimension window width (i.e. ww times). It is possible to affirm that the total number of FLOP is approximately $ww(16LS^2 + 94LS)$. The FLOP of the ReLU function depend mainly from the matrix-matrix multiplication and matrix addition. Considering their dimensionality, the FLOP are $2NF \cdot ww \cdot LS$, where NF is the number of features. Finally, the *softmax* function requires $2LS \cdot NC + 14NC$ FLOP, where NC is the number of classes. It is possible to compute the total number of FLOP needed by a network with N LSTM cell using this formula:

$$\underbrace{2NF \cdot ww \cdot LS}_{ReLU} + N \cdot \underbrace{ww(16LS^2 + 94LS)}_{LSTM} + \underbrace{2LS \cdot NC + 14NC}_{softmax} \quad (9)$$

with a consequent computational complexity of $\mathcal{O}(N \cdot ww \cdot LS^2)$.

With this information, the proposed metrics can be written as:

$$\frac{\text{Net}_{\text{FLOP}}}{\text{MCU}_{\text{FLOPS}}} < \frac{ww}{f_s} \quad (10)$$

$$\text{MCU}_{\text{RAM}} > \text{Size}_{\text{Net}} \quad (11)$$

where Net_{FLOP} is the number of floating point instructions of the network, MCU_{FLOPS} is the number of floating point instructions per second performed by the MCU and f_s is the sampling frequency used for acquiring the samples. The term Net_{FLOP} can be estimated using Eq. 9. Concerning the second metric, the term MCU_{RAM} is the RAM size of the considered MCU, while $Size_{Net}$ is the memory occupancy of the considered network, computed using Eq. 8.

Concerning power consumption, the battery life can be estimated as the ratio between the capacity (B_Q) of the battery expressed in mAh and the total current absorbed by the MCU (I_{abs}):

$$Time = \frac{B_Q}{I_{abs}}. \tag{12}$$

4. Design space exploration

In this section, we make use of the previously introduced metrics for processing time, memory occupancy and battery life to identify which parameters are compliant with the constraints of the adopted MCU. To this purpose, we fixed the values of $NC = 3$, $NF = 3$ and $ww = 100$, since those are suitable values for our target application. The exploration of the design space has been conducted by varying the number of LSTM cells and the inner dimension of each cell. In particular, the number of LSTM cells considered were in the range from 1 to 4, while for the inner dimension we considered the values 4, 8, 16, 32, 64. Fig. 6 shows how the parameters affect the processing time, which is indicated by the blue surface. The red surface represents the constraint of the application in point, i. e. the time limit of 1 s in order to guarantee a real-time analysis. This chart clearly highlights the configurations that can be successfully ported on the MCU under the real-time constraint. As it can be seen, all the networks with $N = 1$ and LS ranging from 4 to 64 are real-time compliant. If the number of N grows up until 4, only the network with an inner dimension lower than 64 are real-time compliant.

Similar considerations can be made for memory occupancy. The blue surface of Fig. 7 represents the memory occupancy, expressed in KB, referred to the variations of N and LS, while the red surface indicates the memory limit, which is 128 KB for the SensorTile. In this case, no network with $LS = 64$ can fit the MCU RAM memory, while all the other configurations can be implemented on the MCU.

Fig. 8 shows the battery life (blue surface) expressed in hours with respect to N and LS. In the figure, the red surface indicates the minimum battery life required, which is arbitrarily set to 48 hours. In this case, it is possible to affirm that all the networks with $LS >= 32$ have a battery life lower than the threshold of 48 hours.

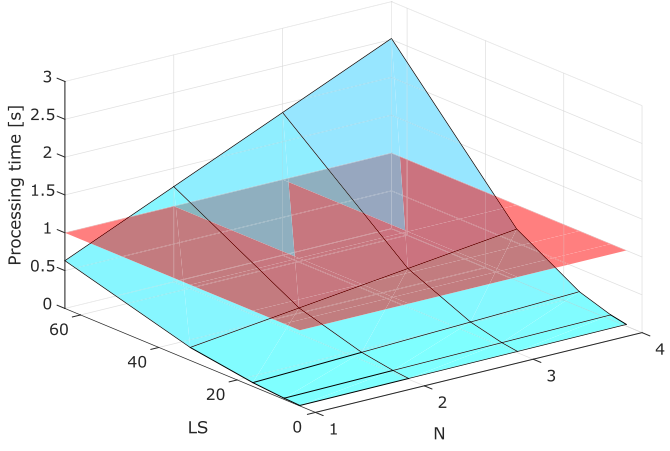

Fig. 6. Processing time compared to the number of LSTM cells N and the inner dimension LS. The blue surface is the processing time, while the red surface represents the time limit of the considered application.

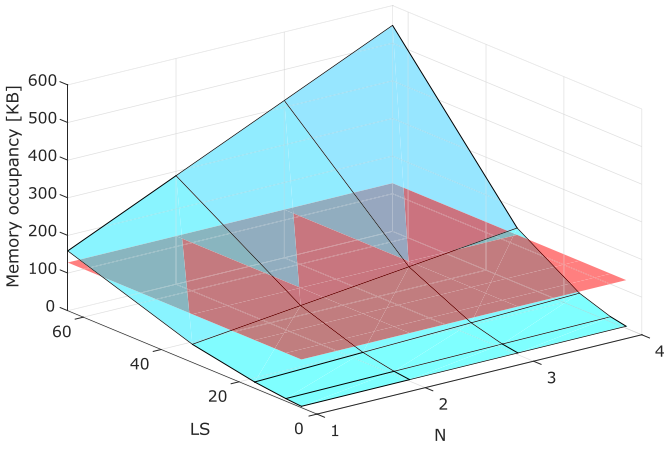

Fig. 7. Memory occupancy compared to the number of LSTM cells N and the inner dimension LS. The blue surface is the memory occupancy, while the red surface represents the memory limit of the SensorTile.

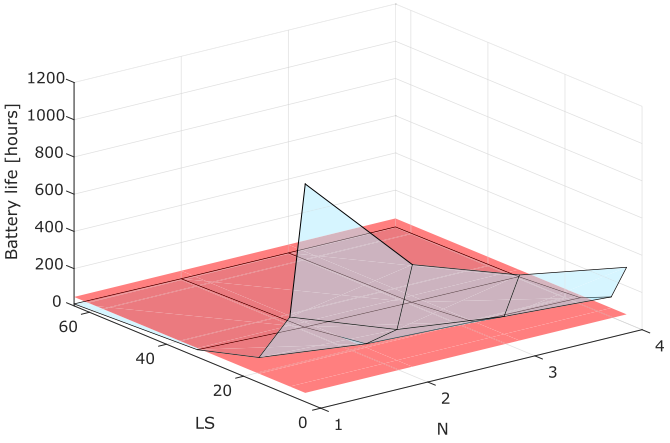

Fig. 8. Battery life compared to the number of LSTM cells N and the inner dimension LS. The blue surface is the battery life, while the red surface represents the minimum required battery life.

Summarizing, no network with $LS = 64$ can be implemented on the MCU due to memory constraint and real-time noncompliance. Moreover, the network made up of 4 LSTM network with an inner dimension of 32 does not respect the memory occupancy constraint. Finally, the network with $N = 3$ and $LS = 32$ has a memory occupancy of 114 KB which is too close to the maximum available memory. For those reasons, in the following, we will consider only networks made up of 1 or 2 LSTM cells and with a maximum inner dimension of 32.

5. Network Optimization

In order to identify the best network to deploy on the target device, we implemented a family of RNN architectures on a Dell® 5810 workstation using TensorFlow 1.8.

As previously stated, networks with more than 2 layers and with an inner dimension greater than 32 have a memory occupancy which is too high for the target architecture. Moreover, experiments performed on networks with more than 2 LSTM cells did not show improvements in terms of accuracy. Thus, the next experiments will consider only networks with 1 and 2 LSTM cells.

The implementation of the RNN architecture on the workstation also allows to test which sensors are suitable to use as input for

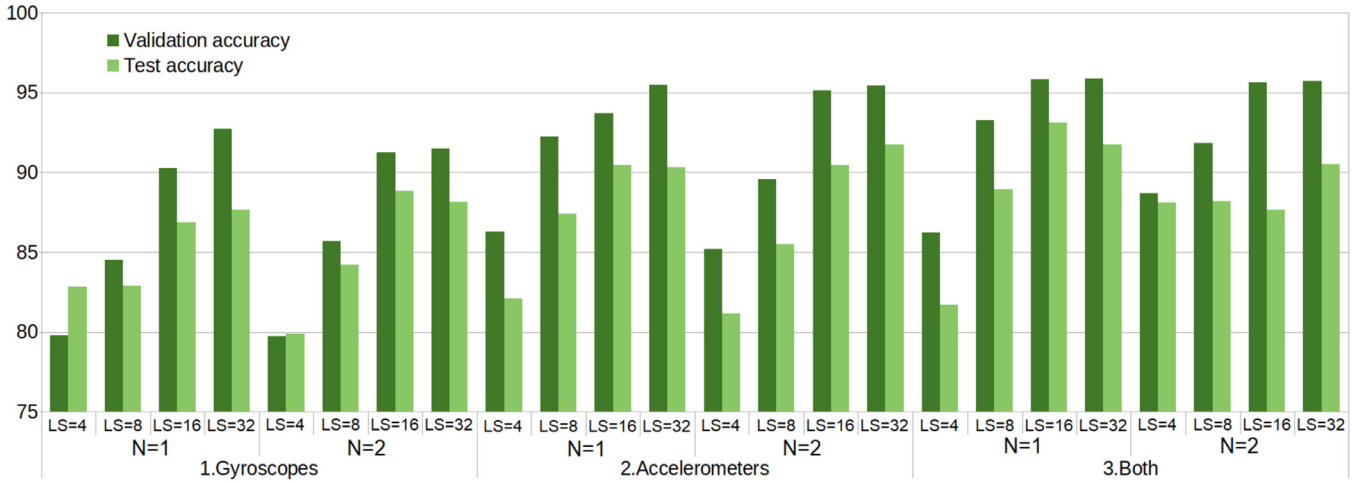

Fig. 9. Validation and test accuracy compared to the number of LSTM cell, the inner dimension and the input type.

the network. In particular, we consider the use of 3D accelerometers, 3D gyroscope or both these signals. This is a key aspect to evaluate, keeping in mind that the memory occupancy increment resulting from adopting two 3D signals instead of a single 3D one is negligible.

To perform a comparative analysis of the available options, we performed extended experiments on the annotated SisFall dataset (see Section 6.1), by considering all possible combinations of the following elements:

- number of LSTM layers: 1 or 2 cells (N);
- inner LSTM cell size: 4, 8, 16 and 32 (LS);
- input sensor readings: 3D accelerometers, 3D gyroscopes or both.

The experiments evaluated the network performance in terms of validation accuracy and test accuracy.

As it can be observed from Fig. 9, the difference of accuracy between $N = 1$ and $N = 2$ is negligible. Therefore, since $N = 2$ requires more memory and has an higher computational complexity than $N = 1$, we choose to discard this option.

Observing only results of the network with only one LSTM cells, the best accuracy is achieved by the networks with $LS =$ 16 and $LS = 32$. In addition, the differences in terms of test accuracy are negligible, making preferrable the smallest network, for the reasons already discussed.

Finally, it is possible to note that the use of both accelerometers and gyroscopes results in a higher accuracy. However, the difference is of about 2% and the gyroscope consumes 500 μA, while the accelerometer consumes about 150 μA. Therefore, the best solution is given by a network made up of only one LSTM cell with an inner dimension of 16, receiving as input only accelerometers data.

6. Experiments

6.1. Dataset

Among the several datasets available for testing and validating our Fall Detection System (FDS), we chose SisFall [11], as it proved to be the most adequate to our purposes.

The SisFall dataset includes the largest amount of data, both in terms of number and heterogeneity of Activities of Daily Living (ADL) and in terms of subjects involved. Specifically, it includes a total of 38 identities: 23 young subjects and 15 elderly subject, simulating 34 different activities in a controlled scenario (19 ADLs and 15 falls) with several retries, for a total of 4510 complete recordings.

SisFall data were acquired via a dedicated embedded device that is fixed on body as a belt buckle. The device was equipped with two different 3D accelerometers (recording at 200 Hz) and a gyroscope. The device was not designed to perform on-board detection.

In general, the training of a DL detection system based on a dataset of measured values requires specific *labeling* that includes temporal annotations, i.e. the explicit indication of temporal intervals associated to the relevant events of interest.

Unfortunately, to the best of our knowledge, temporal labeling is not available in any publicly-available datasets, SisFall included. Therefore, the existing data belonging to the SisFall dataset were manually labeled in order to use them for training and validation set for RNN [9].

To this purpose, two main event classes were defined: FALLS and ALERTS, the latter being events of interest which may or may not directly lead to a proper fall. We also defined a third background class, called BKG which is considered to absorb all the activities that are not related to a fall (ADLs), such as walking, jumping, walking up the stairs, sitting on a chair, etc.

More details about this labeling strategy are discussed in [9].

6.2. Network architecture

The design of the DL architecture for our FDS was driven by the requirement that it should be able to run in real-time on top of a relatively cheap and resource-constrained device, such as the SensorTile.

The core of the DL architecture is depicted in Fig. 10. The network is based on an LSTM cell, with an inner dimension LS of 16 units. The input preprocessing is performed by the fully connected Layer 1, while a second fully connected layer (Layer 6) collects the output from the LSTM cell at Layer 5 and feeds its output to the final classification layer (Layer 7) that provides the classification in the three classes described above.

The adopted solution includes a batch normalization layer [41] (Layer 2) to regularize input data, and two dropout layers [42] (Layers 3 and 5) to improve generalization on the testing dataset. The latter are used during network training to improve generalization, while they are removed in the deployed inference module.

Training such an architecture on the SisFall dataset requires specific care in designing a proper weighted loss function, able to

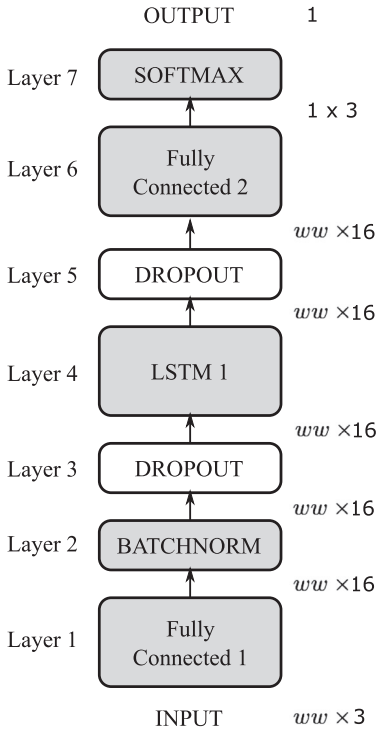

Fig. 10. The model architecture of the proposed solution. White blocks are active during the training phase only and are removed when doing inference on the device. The input size depends on the number of selected features (in our case the 3 axis of the accelerometer) and on the size *w* of the windows to be fed into the model. The output size depends on the number of classes (i.e. BKG, ALERT and FALL).

Table 1
Results of the experimental validation on workstation. The table reports sensitivity, specificity and accuracy on the test dataset for both our proposal and the SisFall baseline.

		SisFall	Our Approach
Sensitivity	BKG	75.01	**88.39**
	ALERT	68.15	**91.08**
	FALL	75.79	**98.73**
Specificity	BKG	92.52	**97.85**
	ALERT	83.30	**90.77**
	FALL	91.57	**97.93**
Accuracy	BKG	83.77	**93.12**
	ALERT	75.73	**90.93**
	FALL	83.68	**98.33**

manage the unbalance in the labelled dataset. Indeed, with a range that depends on the specifics of manual labeling, the BKG class is at least 40 times more represented in the data than either of the two remaining classes.

6.3. Implementation on workstation

The architecture described in Section 6.2 has been implemented using the TensorFlow (TF) library [10], with the Python programming language. All training procedures were performed on a Dell 5810 workstation, equipped with a Nvidia Quadro K5000 GPU.

We used a typical 80%/20% train/test split: the training dataset included 30 subjects (12 elders), while the test dataset included 8 (3 elders). We took particular care in avoiding the so-called *identity bias*: persons present in the train set are not present in the test set, and viceversa.

Table 1 shows the metric resulting after the validation of the proposed architecture on a workstation and on a reimplementation of the SisFall detection, adapted to the manual labelling discussed above (Section 6.1).

On the test data, the model was able to consistently outperform the results presented in the SisFall paper, and achieve good overall accuracy, sensitivity and specificity [9].

In particular we were able to achieve an excellent (98%) accuracy in the detection of falls, with no significant differences in precision between young and elderly subjects.

In order to provide a reference baseline for the embedded implementation, we also tried to apply the TF Quantization API[3] to the graph resulting from the trained model described in Section 6.2.

Two approaches are currently available in TF. The first amounts to weight quantization [34] described in Section 2.3. The second consists in the so-called node-quantization, which replaces any calculation node in the TF graph with their eight-bit integer equivalents. In order to allow remaining float operations (i.e. hyperbolic tangent and sigmoid) to interoperate, conversion layers are added to the graph.

Unfortunately, node-quantization resulted in a catastrophic loss of precision (more than 9% on overall). Indeed, as stated by TF authors, "support for [eight-bit quantization] is still experimental and evolving".

6.4. Run-time on microcontroller

The inference module was developed for the SensorTile device by adopting the data layout described in section 3.1. The implementation is based on the CMSIS library, which includes highly-optimized routines for linear algebra operations. In particular, we adopt the *arm_matrix_instance_f32* structure for all computations and for encoding both weights and inputs; this allows performing all linear algebra operations required with the *arm_mat_mult_f32* and the *arm_add_f32* routines. The working frequency of the MCU has been set to 80 MHz, which is the maximum available.

The pseudocode of the implementation proposed is shown in Algorithm 1.

Algorithm 1 LSTM network.

```
1:  Initialize network                          ▷ Load weights in memory
2:  while true do                               ▷ Repeat computation on input stream
3:      Read input ww
4:      for i:=1 to ww do
5:          Apply fully connected with ReLU layer
6:          for j:=1 to N do
7:              Compute LSTM cell               ▷ evaluate Eq. 2-7
8:          end for
9:          Apply softmax layer
10:     end for
11: end while
```

Line 1 performs RAM memory allocation and the loading of trained weights, which were initially stored in the flash memory. The *while* loop in line 2 is repeated forever to process the incoming signals, by caching windows of width *ww* (line 2) with the predefined stride (not shown). The outer for loop 4 applies the RNN to the sequence of signals in the input window. This is performed by first applying the fully connected layer with the ReLU function and then entering the inner loop 6 which applies the cascade of LSTM cells to the output of the previous layer (in the case considered, *N* is equal to 2). Finally, the softmax layer is applied to the latter

[3] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/graph_transforms.

output. The resulting output classification for the entire input window is the one produced by the softmax layer when $i = ww$, i.e. at the end of the outer loop.

A direct comparison with the solutions presented in Section 2.1 is not fair, since all those works do not perform computation on the devices or adopt simple threshold methods for fall detection.

6.5. Validation

The validation of the runtime module was performed by comparing the numerical outputs of the above implementation with a corresponding implementation made with TensorFlow and run on a workstation. This test was repeated on several input sequences of 100 three-dimensional signals each and the raw numerical outputs of the softmax layers of the two implementations were collected. The mean squared error (MSE) computed on the latter outputs of the two implementations was about 10^{-7}, which gives us reasonable confidence in the computational equivalence of the two implementations.

The validation of the metrics proposed was performed using input windows of $ww = 100$, which corresponds to 1 second at a sampling frequency of 100 Hz. To estimate the $\mathsf{MCU}_{\mathsf{FLOPS}}$, we divided the clock frequency, i. e. 80 MHz, by the average number of clock cycles required to perform each assembly instruction, which in our case equals to 7. With such working frequency, the $\mathsf{MCU}_{\mathsf{FLOPS}}$ value can be estimated at 11.4 MFLOPS, which entails that the $\mathsf{Net}_{\mathsf{FLOP}}$ value, estimated using Eq. 9, is 569,738 FLOP. By inserting the latter values into Eq. 10 we obtain:

$$\frac{569,738}{11.4 \cdot 10^6} < \frac{100}{100} \rightarrow 0.050 < 1. \tag{13}$$

Such value is in very good agreement with the actual processing time, as measured on the device, which was about 0.055 s.

The memory occupancy of the actual implementation is obviously related to Eq. 11 and in the actual implementation is of about 17 KB versus the total amount of memory available on the target MCU, which is 128 KB.

Finally, to evaluate the expected battery charge duration, we first used the STM32CubeMX Power Consumption Calculator to estimate the absorbed current. In doing this, we considered only the MCU power consumption and intentionally neglected any other peripherals involved, in particular the Bluetooth module, since its actual consumption will depend on the adopted communication protocol adopted. On the other hand, the current absorbed by the accelerometers themselves could be neglected since, according to device specifications this is an order of magnitude below that absorbed by the MCU. The overall result yields an estimate equal to about 5 mA. By applying Eq. 12 and considering that the the SensorTile battery has a capacity of 100 mAh, we obtained that the device could be active for about 130 hours without recharging (about 5 days and 10 hours). A final comment concerns the decision to not include the gyroscope data into this embedded implementation. As said before, this sensor absorbs 500 μA, which will lead to an increment of the 10% in the absorbed current (i. e. the total absorbed current will become 5.5 mA). In other words, this will cause a battery life reduction of the 10% (about 13 hours).

7. Conclusion and future works

In this paper, we discuss how to develop a RNN architecture for fall detection on a wearable device equipped with an MCU. First, the reference RNN network has been implemented on a workstation using TensorFlow. This allowed to train and test the architecture with an annotated version of the SisFall dataset, achieving a fall detection accuracy of 98%. After that, the network has been optimized for the SensorTile device, which is suitable to develop a wearable device for fall detection. This network has been validated against the numerical results obtained with TensorFlow on a workstation. Moreover, the test on the embedded device showed that the proposed implementation is real-time compliant. The embedded implementation also validates the three abstract metrics presented about computation power, memory occupancy and battery duration. In particular, the latter metric says that, with this RNN implementation, the device could be operative for about 130 hours without recharging.

Future research line will explore the *quantization* of both the numerical representation of the weights and the processing of linear and non-linear operations [12], which is still an open research problem. Moreover, a crucial importance will be given to the developement of a suitable Bluetooth Low Energy communication protocol which should allow the device to be reliably connected while keeping the lowest possible consumption profile (i. e. limit the amount of communicated data).

Declaration of Competing Interest

The authors declare no conflict of interest.

Acknowlgedgments

The authors acknowledge the financial support from Regione Lombardia, under the “Intelligent Personal Health Safety Domotic Monitoring” (IPHSDM) (ID: 379273).

References

[1] W.H. Organization., WHO global report on falls prevention in older age, World Health Organization Geneva, 2008.
[2] T. Xu, Y. Zhou, J. Zhu, New advances and challenges of fall detection systems: A survey, Applied Sciences 8 (3) (2018) 418.
[3] M. Habib, M. Mohktar, S. Kamaruzzaman, K. Lim, T. Pin, F. Ibrahim, Smartphone-based solutions for fall detection and prevention: challenges and open issues, Sensors 14 (4) (2014) 7181–7208.
[4] X. Fafoutis, et al., Extending the Battery Lifetime of Wearable Sensors with Embedded Machine Learning, in: 2018 IEEE 4rd World Forum on Internet of Things (IEEE WF-IoT), IEEE, 2018.
[5] N. Pannurat, et al., Automatic Fall Monitoring: A Review, Sensors 14 (7) (2014) 12900–12936, doi:10.3390/s140712900.
[6] N. Zerrouki, F. Harrou, A. Houacine, Y. Sun, in: Fall detection using supervised machine learning algorithms: A comparative study, 2016, pp. 665–670, doi:10.1109/ICMIC.2016.7804195.
[7] F.J. Ordonez, et al., Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition, Sensors 16 (1) (2016).
[8] M. Hemmatpour, et al., Springer, Cham, 2017, pp. 241–248, doi:10.1007/978-3-319-58877-3_32.
[9] M. Musci, et al., Online fall detection using recurrent neural networks (2018). arXiv: 1804.04976.
[10] M. Abadi, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org. [Online]. Available: www.tensorflow.org/.
[11] A. Sucerquia, et al., Sisfall: A fall and movement dataset, in: Sensors, 2017, p. 198.
[12] Y. Zhang, et al., in: Hello Edge: Keyword Spotting on Microcontrollers, 2017.
[13] G. Danese, et al., An embedded multi-core biometric identification system, Microprocessors and Microsystems - Embedded Hardware Design 35 (5) (2011) 510–521, doi:10.1016/j.micpro.2011.03.003.
[14] E. Torti, et al., Custom FPGA processing for real-time fetal ECG extraction and identification, Computers in Biology and Medicine 80 (2017) 30–38, doi:10.1016/j.compbiomed.2016.11.006.
[15] S. Rampazzi, et al., A localized surface plasmon resonance-based portable instrument for quick on-site biomolecular detection, IEEE Trans. Instrumentation and Measurement 65 (2) (2016) 317–327, doi:10.1109/TIM.2015.2465691.
[16] P. Canale, et al., Development of a real-time heart rate estimation algorithm on a low-power device, in: 2017 6th Mediterranean Conference on Embedded Computing (MECO), 2017, pp. 1–4, doi:10.1109/MECO.2017.7977199.
[17] R. Igual, et al., Challenges, issues and trends in fall detection systems, BioMedical Engineering OnLine 12 (1) (2013) 66, doi:10.1186/1475-925X-12-66.
[18] M. Nyan, et al., A wearable system for pre-impact fall detection, Journal of Biomechanics 41 (16) (2008) 3475–3481, doi:10.1016/j.jbiomech.2008.08.009.
[19] S. Jung, et al., Wearable Fall Detector using Integrated Sensors and Energy Devices, Scientific Reports 5 (1) (2015) 17081, doi:10.1038/srep17081.
[20] S. Abdelhedi, et al., Design and implementation of a fall detection system on a Zynq board, in: 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), IEEE, 2016, pp. 1–7, doi:10.1109/AICCSA.2016.7945775.

[21] G. Cola, et al., in: Fall Detection Using a Head-Worn Barometer, Springer, Cham, 2017, pp. 217–224, doi:10.1007/978-3-319-58877-3_29.
[22] F. Hossain, et al., A direction-sensitive fall detection system using single 3D accelerometer and learning classifier, in: 2016 International Conference on Medical Engineering, Health Informatics and Technology (MediTec), IEEE, 2016, pp. 1–6, doi:10.1109/MEDITEC.2016.7835372.
[23] A. Burns, et al., Shimmer™; a wireless sensor platform for noninvasive biomedical research, IEEE Sensors Journal 10 (9) (2010) 1527–1534, doi:10.1109/JSEN.2010.2045498.
[24] T.N. Gia, V.K. Sarker, I. Tcarenko, A.M. Rahmani, T. Westerlund, P. Liljeberg, H. Tenhunen, Energy efficient wearable sensor node for iot-based fall detection systems, Microprocessors and Microsystems 56 (2018) 34–46.
[25] A. Nicosia, et al., Efficient light harvesting for accurate neural classification of human activities, in: 2018 IEEE International Conference on Consumer Electronics (ICCE), 2018, pp. 1–4, doi:10.1109/ICCE.2018.8326103.
[26] I. Goodfellow, et al., Deep Learning, MIT Press, 2016. http://www.deeplearningbook.org.
[27] S. Hochreiter, et al., Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.
[28] F.A. Gers, et al., Learning to forget: Continual prediction with LSTM, 1999.
[29] R. Jozefowicz, et al., An empirical exploration of recurrent network architectures, in: International Conference on Machine Learning, 2015, pp. 2342–2350.
[30] T. Mauldin, M. Canby, V. Metsis, A. Ngu, C. Rivera, Smartfall: a smartwatch-based fall detection system using deep learning, Sensors 18 (10) (2018) 3363.
[31] A. Nait Aicha, G. Englebienne, K. van Schooten, M. Pijnappels, B. Kröse, Deep learning to predict falls in older adults based on daily-life trunk accelerometry, Sensors 18 (5) (2018) 1654.
[32] R. Rajagopalan, I. Litvan, T.-P. Jung, Fall prediction and prevention systems: recent trends, challenges, and future research directions, Sensors 17 (11) (2017) 2509.
[33] B. Hu, P. Dixon, J. Jacobs, J. Dennerlein, J. Schiffman, Machine learning algorithms based on signals from a single wearable inertial sensor can detect surface-and age-related differences in walking, Journal of biomechanics 71 (2018) 37–42.
[34] S. Han, et al., Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding (2015). arXiv: 1510.00149.
[35] M. Denil, et al., Predicting parameters in deep learning, in: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, in: NIPS'13, Curran Associates Inc., USA, 2013, pp. 2148–2156.
[36] Y. Gong, et al., Compressing deep convolutional networks using vector quantization (2014). CoRR arXiv: 1412.6115.
[37] S. Anwar, et al., Fixed point optimization of deep convolutional neural networks for object recognition, in: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015, pp. 1131–1135, doi:10.1109/ICASSP.2015.7178146.
[38] S.S. Talathi, et al., Improving performance of recurrent neural network with relu nonlinearity (2015). CoRR arXiv: 1511.03771.
[39] M. Courbariaux, et al., in: Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1, 2016.
[40] K. Hwang, et al., Fixed-point feedforward deep neural network design using weights −1, 0, and +1, in: 2014 IEEE Workshop on Signal Processing Systems (SiPS), 2014, pp. 1–6, doi:10.1109/SiPS.2014.6986082.
[41] S. Ioffe, et al., Batch normalization: Accelerating deep network training by reducing internal covariate shift (2015). CoRR arXiv: 1502.03167.
[42] G.E. Hinton, et al., Improving neural networks by preventing co-adaptation of feature detectors (2012). CoRR arXiv: 1207.0580.

Emanuele Torti was born in Voghera, Italy, in 1987. He received the Ph.D. degree in electronics and computer science engineering from University of Pavia, Pavia, Italy in 2014. He received the Bachelor's Degree in Electronic Engineering and Master's Degree in Computer Science Engineering (cum laude) from University of Pavia in 2009 and 2011, respectively. He is a post-doc researcher at the Engineering Faculty of the University of Pavia. His research is focused on high performance architectures for real-time image processing and signal elaboration.

Alessandro Fontanella was born in Pavia, Italy, in 1991. He received the Bachelor's degree in computer science engineering and Master's degree in computer science engineering (cum laude) from the University of Pavia, Pavia, Italy, in 2013 and 2015, respectively, where he is currently pursuing the Ph.D. degree in computer science engineering. His research is focused on high-performance architectures for real-time image processing.

Mirto Musci received a Ph.D. in Computer Engineering in 2014 from the University of Pavia. He is currently research fellow in the Computer Vision lab of University of Pavia, under the supervision of Professor Virginio Cantoni. He also teaches Parallel Programming at the same university. Mirto Musci is the author of several publications in international journals and conference proceedings. His main research interest is parallel computing in bioinformatics. Recently, he has begun working with Deep Neural Networks.

Nicola Blago received the Master Degree in Computer Engineering from the University of Pavia in 2017. He is cofounder of the student association Associazione Giasoniani del Maino. He currently works as a programmer in the field of Artificial Intelligence.

Danilo Pietro Pau graduated at Politecnico di Milano, Italy on 1992 in Electronic Engineering. Since 1991 he joined STMicroelectronics in System Research and Applications department. Danilo worked on different R&D subjects such memory reduced HDMAC and MPEG video decoding, video coding, video transcoding, embedded 3D and 2D graphics, computer vision and deep learning, aiming to transfer them into company products. Currently he holds Technical Director, Fellow Member of Technical Staff position. He founded and served as Chairman of the STMicroelectronics Technical Staff Italian Community. Since 2019 is IEEE Fellow and serves Action for Industry as Industry Ambassador member for IEEE Region 8 South Europe and be vice chair of Task Force on "Intelligent Cyber-Physical Systems" within IEEE Computational Intelligence Society.

Francesco Leporati achieved the Laurea degree in Electronics Engineering on 1988 and the Ph.D. degree in Electronics and Computer Engineering, at the University of Pavia on 1993. At present he is associate professor at the Computer Science Dept. of the University of Pavia where he teaches Industrial Electronics, Industrial Informatics and Embedded Systems and Digital Systems Design. His research activity concerns the design and the implementation of architectures for high performance computing, in particular exploiting FPGA and GPU technologies, applied to Signal and image processing, Automotive applications, Biomedical instrumentation. Some of these activities were awarded by Altera, Altran Engineering and Texas. Francesco Leporati collaborated with different companies (Neuricam inc., Ferrari inc., Marelli Motorsport, ST Microelectronics) and public research institutions. He is author of more than 70 journal or conference papers and reviewer of several ACM or IEEE journals. In particular he is member of the Elsevier Microprocessors and Microsystems Editorial Board where he serves as Handling Editor in the Subject Area of FPGA-based Systems and Applications. Francesco Leporati was Chair of the XI IEEE/Euromicro Conference on Parallel and Distributed Processing (2001), of the IEEE/Euromicro on Software Engineering and Advanced Applications and on Digital System Design (SEEA/DSD) (2008) and Program Chair of the IEEE/Euromicro Conference on Digital Systems Design (2014). He is member of IEEE Computer Society and of the Euromicro Society (Director of Italian correspondents and member of the Steering Committee of the DSD conference). He serves as Italian Representative in the EU ICT COST Action IC1204 Trusworthy Manufacturing and Utilization of Secure Devices.

Marco Piastra is Contract Professor of Artificial Intelligence and of Deep Learning at Università degli Studi di Pavia, Italy and Contract Professor of Machine Learning at Università Cattolica del Sacro Cuore, Brescia, Italy. He received the Master degree in Electric Engineering and the Ph.D. degree in Electronics and Computer Engineering from the University of Pavia in 1997 and 2002, respectively. In his professional activity, he has been leading international projects and activities involving artificial intelligence and machine learning techniques applied to the fields of automotive engineering, financial trading, mechanical engineering, and e-government since 1988. He has been member of the editorial board for international journals and conferences in the field. His research interests are now focused on applied deep learning and deep reinforcement learning to several application fields in industry, including robotics.