2018 21st Euromicro Conference on Digital System Design

Embedded Real-Time Fall Detection with Deep Learning on Wearable Devices

Emanuele Torti*, Alessandro Fontanella*, Mirto Musci*, Nicola Blago*,
Danilo Pau†, Francesco Leporati* and Marco Piastra*
*Department of Electrical, Computer and Biomedical Engineering
University of Pavia, Pavia, Italy I-27100
Email: {emanuele.torti, mirto.musci, francesco.leporati, marco. piastra}@unipv.it,
{alessandro.fontanella01, nicola.blago01}@universitadipavia.it
†Advanced System Technology, STMicroelectronics, Agrate Brianza, Italy
Email: danilo.pau@st.com

***Abstract*—Unintentional falls are the leading cause of fatal injuries and nonfatal trauma among older adults. An automated monitoring system that detects occurring falls and issues remote notifications will prove very valuable for improving the level of care that could be provided to people at higher risk. The work presented focuses on the design of embedded software for wearable devices that are connected in wireless mode to a remote monitoring system. The work focuses on the implementation of *recurrent neural networks* (RNNs) architectures of micro controller units (MCU) for fall detection with tri-axial accelerometers. A few general formulas for determining memory, computing power and power consumption for such architectures are presented. These formulas have been validated with an actual implementation for the SensorTile® device by STMicroelectronics.**

***Index Terms*—Fall detection, Embedded systems, Deep Learning, Recurrent Neural Networks, Wearable devices.**

I. Introduction

Unintentional falls are the leading cause of fatal injuries and the most common cause of nonfatal trauma-related hospital admissions among older adults. As stated in [1], more than 25% of people aged over 65 years old falls every year increasing to 32%–42% for those over 70. Moreover, 30%–50% of people living in long-term care institutions fall each year, with almost half of them experiencing recurrent falls. 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. Elderly people are not the only group that is heavily affected by unintentional falls: any person with some sort of fragility is part of similar statistics.

In such scenario, an automated monitoring system that can promptly detect occurring falls and issue remote notifications, so that timely aid can be given, will prove extremely valuable for enhancing the level of care that could be provided to people with fragilities of any sort.

The overall aim of this paper is to study the design of embedded software for wearable sensing devices equipped with a micro-controller unit (MCU) that governs sensory data acquisition, runs algorithms for fall detection in real time and manages 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;
3) be running on battery for as long as possible without recharge, as the monitoring activity needs to be performed 24×7.

These general requirements also entail several software design issues that should be considered carefully. First of all, ideally, 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 [2]. In consequence, algorithms for real-time fall detection should be explicitly designed and implemented for low-power MCUs.

In the literature, automated fall detection in real-time from data produced by sensors such as tri-axial accelerometers is considered still an open research problem [3], for which machine and deep learning techniques have shown a very promising potential [2], [4].

In this paper we investigate in particular the MCU embedding of *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.

Furthermore, in the work presented, such abstract formulas were validated through an experimental procedure involving two main software components:

- a workstation-based TensorFlow [5] complete implementation of a reference RNN architecture for training and detection;
- 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.

Moreover, this run-time detection module performs computation directly on the embedded device, unlike state of the art solutions presented in Section II.A.

As it will be described below, the training phase of the RNN architecture was performed on a workstation with the well-known SisFall dataset [6], while the resulting binary models were transfered to the embedded component for the final validation. Several implementation variants, in particular for the numerical encoding of binary RNN models, were also considered [7].

The proposed activity prolongs the tradition of the authors' laboratory, where, in the past, embedded and wearable devices were developed [8]–[10].

The rest of the paper is organized as follows: Section II 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 III describes the feasibility analysis performed and the resulting abstract formulas for the embedding of RNN architectures; Section IV describes the validation experiments performed and, finally, Section V contains the conclusions and future work.

II. Context

A. 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 [11].
Several wearable devices have been proposed in the literature for fall detection. Nyan et. al [12] 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 [13] 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 [14], an integrated board carries the accelerometers and an FPGA for signal processing. Cola et al. [15] 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 pre-defined 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 [16] is based on the SHIMMER (Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability) integrated sensor platform [17]. 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 [4], which performs offline elaboration of signals acquired by accelerometers and gyroscopes. In these cases, data are analyzed through machine learning approaches.

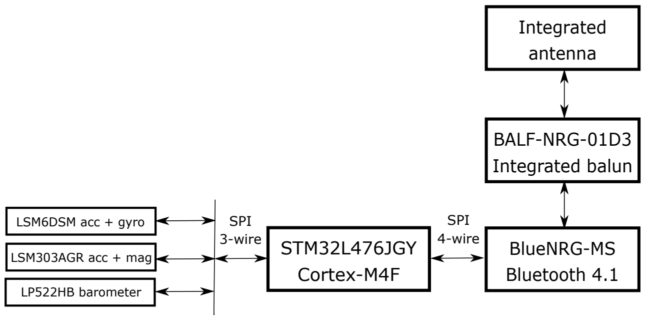


Fig. 1. SensorTile architecture

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, memory and computational capabilities. The overall architecture of the SensorTile is shown in Figure 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 [18] for human activity recognition.

B. Deep Learning

Nowadays Deep Learning (DL) is one of the most promising and discussed paradigm, both by the scientific community and by the general public. DL techniques are being developed to perform an increasing number of tasks in the more disparate fields, from playing board games to recognizing objects in a real-time video, to translate between two different languages.

Recurrent Neural Network (RNN) are a particular form of artificial neural network [19] in which a part of the output produced is fed back to the input, as shown in Fig. 2 on the left.

More formally a RNN can be described by:

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

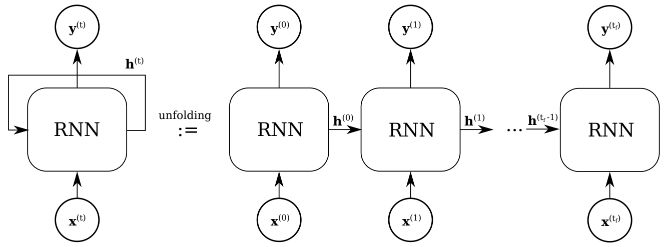
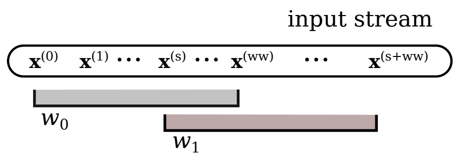


Fig. 2. A single RNN cell (on the left) is temporally unfolded for training. The depth of unfolding is also the depth of the network.



input stream
$\mathbf{x}^{(0)}$ $\mathbf{x}^{(1)}$ ··· $\mathbf{x}^{(s)}$ ··· $\mathbf{x}^{(ww)}$ ··· $\mathbf{x}^{(s+ww)}$
w_0
w_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 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 $\mathbf{w}, \mathbf{W}, \mathbf{U}, \mathbf{b}$ and $\mathbf{c}$ are the parameters. $\mathbf{h}^{(t)}$ is usually referred to as the *hidden state* of the network [19].

Due to recurrence, RNN are on the one hand most suited for analysizing 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 is fed in input to the unfolded network. The parameter optimization process takes place via *stochastic gradient descent* over a training dataset of 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 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 pre-defined 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 of constant length called *strides* (see Fig. 3).

As seen in Fig. 2, temporally unfolding is what makes RNNs to be training as it they were deep networks; this makes it possibile to apply to the same training process all the tecnhniques that become popular with DL [19], while still keeping inference as a relatively inexpensive process from a computation standpoint.

Long Short-Term Memory (LSTM) [20], [21] 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' part of their hidden state during the inference process. Furthermore, in spite of the extra complexity added in their structure, LSTM become relatively easy to train.

Fig. 4 shows an 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* $\mathbf{c}$, *input gate* $\mathbf{i}$; *output gate* $\mathbf{o}$ and *forget gate* $\mathbf{f}$. Each gate has a structure similar to eq. 1, and is described by a parameter matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$ and a non-linear function which is typically a sigmoid or a hyperbolic tangent in this case.

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

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

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

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

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

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

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

where

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

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

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 [22]. $\mathbf{b}_{\text{forget}}$ is typically set equal to 1.

Other authors [23] have proposed a DL technique to detect falls composed by two parts: a Convolutional Neural Network (CNN) and a LSTM. CNNs are mostly concerned in capturing spatial information in the data and are one of the most powerful tools in image recognition and processing [19] In

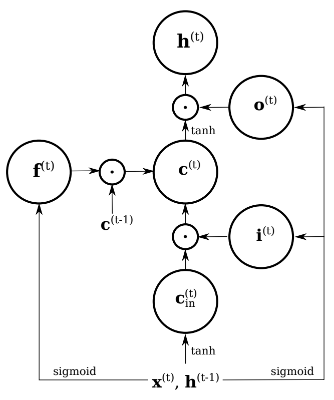


Fig. 4. The basic structure of an LSTM cell. Each circle represents an equation in the mathematical formulation (eq. 2-7). For each arrow pointing to a circle, an addition is performed. Dots represent vector/matrix multiplications.

the above approach, CNNs are leveraged to automatically extract features from sensor signals, while LSTMs recognize a temporal relationship among extracted features. This work has been tested off-line and implemented on a workstation and not in an embedded system.

C. Deep learning for embedded system

More recently, DL has started to emerge as a viable technique on embedded devices. For example, one of the most widespread and complete software frameworks for DL, TensorFlow [5], is now available on smartphones in the form of TensorFlow Lite[1].

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. In general, according to all the works mentioned below, these implementations are for inference only, while training remains very challenging for MCUs.

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 [5].

More sophisticated strategies described in the literature are parameter compression [24], which includes weight pruning and quantization, weight matrix approximation [25], weight clustering [26] and fixed-point representation [27].

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 eq. 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 [28].

Integer quantization is another even more extreme strategy for reducing memory occupancy as it could be based on 8-bit integers [27]. 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 [29], [30]. 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 example preliminary implementations of some quantization strategies are at present included in TensorFlow (TF).

III. Feasibility analysis

A. Optimizations

A fundamental aspect in the embedded implementation of LSTM cells on MCUs is the design of the data storage layout since this has a strong influence on both computations and memory occupancy. In particular, a suitable data storage layout can increase the efficiency with which the operations involved in eq.2-7 are carried out. The layout adopted in this work is shown in Figure 5. 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). Subsequently, c_t is computed using another temporary vector of dimension LS (not shown in figure). Note 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.

Another fundamental aspect of the optimization is about the numerical representation of data as mentioned before. In particular, we adopted single precision floating point (i.e. 32 bits) arithmetic since, with our target application, it is not possible to afford significant losses in precision that pruning or quantization would bring. In addiction, floating

[1]https://www.tensorflow.org/mobile/tflite/

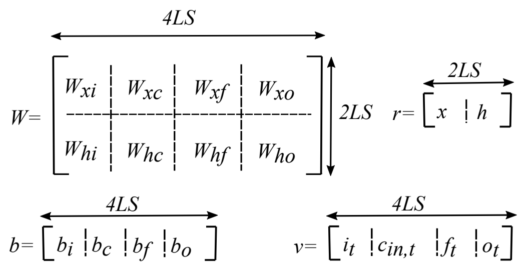

4LS
W=
W_{xi} W_{xc} W_{xf} W_{xo}
W_{hi} W_{hc} W_{hf} W_{ho}
2LS
2LS
r= x h
4LS
b= b_i b_c b_f b_o
4LS
v= i_t $c_{in,t}$ f_t o_t

Fig. 5. Schema of data organization

point operations are already optimized in the ARM CMSIS library [2].

B. General metrics

In the light of the above data storage 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 in the sense that they rely on parameters describing the values specific to a particular MCU. They have a structure which is invariant across multiple MCU.

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

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

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 equation (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 depends 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:

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

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

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

$$\frac{\mathsf{Net_{FLOP}}}{\mathsf{MCU_{FLOPS}}} < \frac{ww}{f_s} \tag{10}$$

$$\mathsf{MCU_{RAM}} > \mathsf{Size_{Net}} \tag{11}$$

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

A final remark is about 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}):

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

IV. Experiments

A. Dataset

Among the several datasets available for testing and validating our Fall Detection System (FDS), we chose SisFall [6], as it was deemed the most complete.

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

In fact, SisFall includes the largest amount of data, both in terms of number and heterogeneity of Activities of Daily Living (ADL). It features moreover a very significant number of subjects among the publicly available datasets. 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.

In addition, most datasets gather data from smartphones, while SisFall adopts a dedicated embedded device that is fixed on body as a belt buckle. Such device includes two different models of 3D accelerometers (recording at 200 Hz) and a gyroscope, but is not able to perform on-chip 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, temporal labeling is not available in any available datasets, SisFall included. Therefore, the existing data belonging to the SisFall dataset was manually labeled to enable the use as a training and validation set for the neural network [31].

In this case, we define two relevant classes: 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 labeling strategy are discussed in [31].

B. 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. 6. The network is based on two LSTM cells stacked on top of each other (Layers 4 and 6). Each cell has a inner dimension LS of 32 units. The input preprocessing is performed by the fully connected Layer 1, while a second fully connected layer (Layer 8) collects the output from the LSTM cell at Layer 6 and feeds its output to the final classification layer (Layer 9) that provides the classification in the three classes described above.

The adopted solution includes a batch normalization layer [32] (Layer 2) to regularize input data, and three dropout layers [33] (Layers 3, 5 and 7) 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 architecture on the SisFall dataset requires specific care in designing a proper weighted loss function, able to 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.

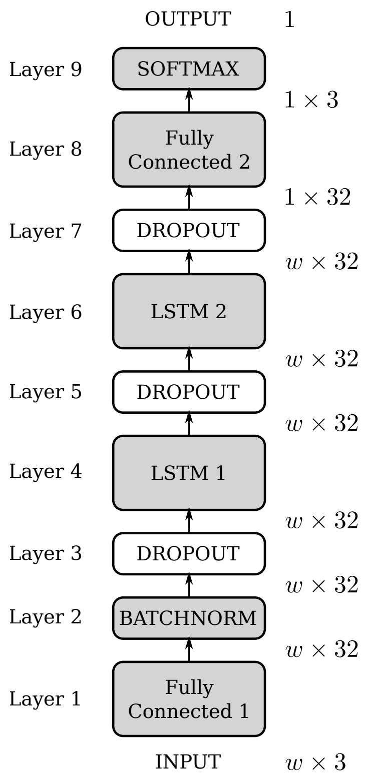


Fig. 6. The model architecture of the proposed solution. White blocks are active during the training phase only and are removed when doing inference on the device. The input size depends on the number of selected features (in our case the 3 axis of the accelerometer) and on the size w of the windows to be fed into the model. The output size depends on the number of classes (i.e. BKG, ALERT and FALL).

C. Implementation on workstation

The architecture described in Section IV-B has been implemented using the TF library [5], 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 I 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 IV-A).

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 [31].

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

		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**

TABLE I
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.

API[3] to the graph resulting from the trained model described in Section IV-B.

Two approaches are currently available in TF. The first amounts to weight quantization [24] described in Section II-C. 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".

D. Run-time on microcontroller

The inference module was developed for the SensorTile device by adopting the data layout described in section III-A. The implementation harnesses the CMSIS library, which includes highly-optimized routines for linear algebra operations. More precisely, he *arm_matrix_instance_f32* structure is adopted 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
```

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

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 3) with the pre-defined 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 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 II.A is not fair, since all those works do not perform computation on the devices or adopt simple threshold methods for fall detection.

E. 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 $\mathsf{MCU_{FLOPS}}$, 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 $\mathsf{MCU_{FLOPS}}$ value can be estimated at 11.4 MFLOPS, which entails that the $\mathsf{Net_{FLOP}}$ value, estimated using eq. 9, is $3,897,843$ FLOP. By inserting the latter values into eq. 10 we obtain:

$$\frac{3897843}{11.4 \cdot 10^6} < \frac{100}{100} \rightarrow 0.342 < 1. \tag{13}$$

Such value is in very good agreement with the actual processing time, as measured on the device, which was about 0.3 s.

The memory occupancy of the actual implementation is obviously related to eq. 11 and in the actual implementation is of about 82 KB versus the total amount of memory available on the target MCU, which is 128 KB.

Finally, for evaluating 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 20 hours without recharging.

V. Conclusion and Future Works

In this paper, we presented the development of a RNN architecture for fall detection which is suitable for an embedded implementation on an MCU. A reference implementation of the RNN architecture was made in TensorFlow. Such architecture was trained and tested on specially-annotated version of the SisFall dataset and, with that, we achieved an overall accuracy of 98% in the detection of falls. Subsequently, a higlhy-optimized, run-time detection module was implemented for the SensorTile device. The implementation was validated against the numerical results obtained with TensorFlow on a workstation. In addition, the test conducted have shown that a time-to-process ratio for input signals of 0.3 can be achieved and this proves the viability of the approach proposed for real-time on-board processing. Such 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 20 hours without recharging.

One interesting direction for future developments is about the *quantization* of both the numerical representation of the weights and the processing of linear and non-linear operations [7], which is still an open research problem. Another aspect to be further investigated is the Bluetooth communication protocol which should allow the device to be reliably connected while keeping the lowest possible consumption profile.

Acknowledgment

The authors acknowledge the financial support from Regione Lombardia, under the "Home of IoT" project (ID: 139625), co-funded by POR FESR 2014-2020.

References

[1] W. H. Organization., *WHO global report on falls prevention in older age*. World Health Organization Geneva, 2008.

[2] 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.

[3] N. Pannurat *et al.*, "Automatic Fall Monitoring: A Review," *Sensors*, vol. 14, no. 7, pp. 12 900–12 936, jul 2014.

[4] M. Hemmatpour *et al.* Springer, Cham, nov 2017, pp. 241–248.

[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] A. Sucerquia *et al.*, "Sisfall: A fall and movement dataset," in *Sensors*, 2017.

[7] Y. Zhang *et al.*, "Hello Edge: Keyword Spotting on Microcontrollers," 2017.

[8] G. Danese *et al.*, "An embedded multi-core biometric identification system," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 35, no. 5, pp. 510–521, 2011.

[9] E. Torti *et al.*, "Custom FPGA processing for real-time fetal ECG extraction and identification," *Computers in Biology and Medicine*, vol. 80, pp. 30–38, 2017.

[10] S. Rampazzi *et al.*, "A localized surface plasmon resonance-based portable instrument for quick on-site biomolecular detection," *IEEE Trans. Instrumentation and Measurement*, vol. 65, no. 2, pp. 317–327, 2016.

[11] R. Igual *et al.*, "Challenges, issues and trends in fall detection systems," *BioMedical Engineering OnLine*, vol. 12, no. 1, p. 66, Jul 2013.

[12] M. Nyan *et al.*, "A wearable system for pre-impact fall detection," *Journal of Biomechanics*, vol. 41, no. 16, pp. 3475–3481, dec 2008.

[13] S. Jung *et al.*, "Wearable Fall Detector using Integrated Sensors and Energy Devices," *Scientific Reports*, vol. 5, no. 1, p. 17081, dec 2015.

[14] 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, nov 2016, pp. 1–7.

[15] G. Cola *et al.*, "Fall Detection Using a Head-Worn Barometer." Springer, Cham, nov 2017, pp. 217–224.

[16] 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, dec 2016, pp. 1–6.

[17] A. Burns *et al.*, "Shimmer$^{\text{TM}}$; a wireless sensor platform for noninvasive biomedical research," *IEEE Sensors Journal*, vol. 10, no. 9, pp. 1527–1534, Sept 2010.

[18] A. Nicosia *et al.*, "Efficient light harvesting for accurate neural classification of human activities," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2018, pp. 1–4.

[19] I. Goodfellow *et al.*, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[20] S. Hochreiter *et al.*, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov 1997.

[21] F. A. Gers *et al.*, *Learning to forget: Continual prediction with LSTM*, 1999.

[22] R. Jozefowicz *et al.*, "An empirical exploration of recurrent network architectures," in *International Conference on Machine Learning*, 2015, pp. 2342–2350.

[23] F. J. Ordez *et al.*, "Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition," *Sensors*, vol. 16, no. 1, 2016.

[24] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[25] M. Denil *et al.*, "Predicting parameters in deep learning," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 2148–2156.

[26] Y. Gong *et al.*, "Compressing deep convolutional networks using vector quantization," *CoRR*, vol. abs/1412.6115, 2014.

[27] 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)*, April 2015, pp. 1131–1135.

[28] S. S. Talathi *et al.*, "Improving performance of recurrent neural network with relu nonlinearity," *CoRR*, vol. abs/1511.03771, 2015.

[29] M. Courbariaux *et al.*, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 02 2016.

[30] 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)*, Oct 2014, pp. 1–6.

[31] M. Musci *et al.*, "Online fall detection using recurrent neural networks," *arXiv preprint arXiv:1804.04976*, 2018.

[32] S. Ioffe *et al.*, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[33] G. E. Hinton *et al.*, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.