Received 1 October 2019; revised 30 July 2020; accepted 6 August 2020.
Date of publication 29 September 2020; date of current version 16 September 2021.

Digital Object Identifier 10.1109/TETC.2020.3027454

Online Fall Detection Using Recurrent Neural Networks on Smart Wearable Devices

MIRTO MUSCI, DANIELE DE MARTINI, NICOLA BLAGO, TULLIO FACCHINETTI, AND MARCO PIASTRA

Mirto Musci, Tullio Facchinetti, and Marco Piastra are with the Department of Electrical, Computer and Biomedical Engineering, University of Pavia, 27100 Pavia, Italy
Daniele De Martini is with the Department of Engineering Science, University of Oxford, OX1 2JD, Oxford U.K.
Nicola Blago is with the Artificial Intelligence Engineer, K Linx srl,
CORRESPONDING AUTHOR: MIRTO MUSCI (mirto.musci@unipv.it)

ABSTRACT Unintentional falls can cause severe injuries and even death, especially if no immediate assistance is given. A fall detection system aims to detect a fall as soon as it occurs, therefore issuing an automatic assistance request. Wearable embedded sensors are emerging as the most viable solution for continuous monitoring since they are more effective, less intrusive and less expensive than other systems. Tailoring a deep learning method to the requirements of microcontrollers entails matching very stringent constraints in terms of both memory and computational power. In addition, datasets acquired with wearable devices are relatively scarce and not necessarily devised for supervised learning. In this work, we discuss the design of a software architecture based on recurrent neural networks which can be effective for fall detection while running entirely onboard a wearable device. The well-known and publicly-available SisFall dataset was adopted and extended with fine-grained temporal annotations to cope with the supervised training of recurrent neural networks. We then show that an appropriate process of architectural minimization together with accurate hyperparameters selection leads to a workable model which compares favorably with other detection techniques. The embedding of the resulting architecture has been validated using a state-of-art hardware device.

INDEX TERMS C.3.d real-time and embedded systems, I.2.6.g machine learning, J.3.b health, I.2.9.j sensors, I.2.m.d wearable AI

I. INTRODUCTION

Unintentional falls are the leading cause of fatal injuries and the most common cause of non-fatal trauma-related hospital admissions among elderly people. More than 25 percent of people over 65 years old fall every year [1]. This percentage increases up to 42 percent for those over 70. Moreover, 30–50 percent of people living in long-term care facilities fall each year, with almost half of them experiencing recurrent falls. Falls lead to 20–30 percent of mild to severe injuries and to 40 percent of all injury-related deaths. Elderly people are not the only demographic that is heavily affected by unintentional falls: any person with some sort of fragility, such as mild disability or a post-operative situation, experiences similar risks. The situation is worsened when people live alone, as they might not receive immediate assistance in case of accident [2].

The main objective of Fall Detection System (FDS) is to implement *online* (i.e., continuous) monitoring of vulnerable subjects in order to detect occurring falls, hence issuing automatic assistance requests so that timely aid can be provided. From both a practical and psychological standpoint [3], an effective FDS can drastically improve the quality of life for patients. Wearable embedded sensors based on specific hardware are gaining momentum in the literature [4], since they are considered less intrusive and more cost-effective than ambient sensors–like video cameras–and are more power-efficient than using smartphones [5].

Generally, application-specific wearable devices for fall detection make use of tri-axial accelerometers and/or gyroscopes. While data collection and filtering is always performed on the wearable device, in the line of principle Fall Detection (FD) could be performed either on-board or remotely, in cloud (via a gateway of some sort) or on a smartphone. Typically, the communication module is the most power-consuming one in any wireless wearable device [6], even with low-energy standards such as Bluetooth Low Energy (BLE). For this reason, the trend in recent literature is to move the entire detection infrastructure on a *smart* wearable device, that is a device that is capable to perform as much signal processing as possible *onboard*.

An online FD method that runs continuously on an embedded device must obey stringent constraints due to the limited resources available. In particular, any such method must fit the limited memory—typically in the range of tens of kilobytes—and computational power available on microcontrollers. Overall, the techniques adopted must have an ultra-low power-consumption footprint, to achieve the longest battery duration possible and lessen the need to switch devices.

Several embedded solutions have been proposed for online FD, using simple statistical indicators and/or classic machine learning. In addition, several *offline* methods for fall detection using Deep Learning (DL) techniques have been proposed in the literature (Section II-A). Nevertheless, to the best of our knowledge, no study currently exists about the applicability of DL techniques to online fall detection, except for the preliminary work published by the present authors.

The main contribution of this paper is an in-depth discussion of the design and implementation of a deep learning architecture for fall detection, which can run continuously on a wearable device while still obeying the stringent constraints dictated by the embedding on small microcontrollers. The main design goal of the method presented is to find a working trade-off among detection performance, power consumption and architectural complexity.

In particular, we focus on Long Short-Term Memory (LSTM)-based architectures, a well-known kind of Recurrent Neural Network (RNN) that is typically employed for time series analysis [7]. Due to the adoption of RNN as the method of reference, the dataset used for supervised training needs to be *temporally annotated*, meaning that each occurrence of a relevant event—falls in particular—in sequences of sensor readings must be marked by a temporal interval. After a careful analysis of publicly available fall datasets (Section II-B), we found that none of them contained the temporal annotations required. For this reason, we selected SisFall [8], [9] as a dataset of reference and extended it by manually adding event markers. Such extension has been made publicly-available.[1]

In this work, several architectural variants based on stacked LSTM cells were considered and trained on the annotated SisFall dataset to detect the best performing one (Section III). In Section V we show that a simple network configuration can achieve high detection accuracy on all three classes in the test set, outperforming the statistical indicators of reference, while respecting the design constraint discussed above. We also show that the proposed network reaches comparable performance to a state-of-art SVM implementation.

Finally, to validate the proposed approach, we discuss the actual embedding of a LSTM architecture of reference on a state-of-art hardware device. Specifically, we chose the SensorTile miniaturized board by STMicroelectronics,[2] which is explicitly designed as an host for wearable smart sensors.

The organization of the paper is as follows: Section II contains a review of the related literature; Section III describes the enhanced annotation process and design of the RNN architecture of reference; Section IV discusses the implementation of the proposed system, while Section V presents some selected experimental result achieved on the extended SisFall dataset in terms of both classification performance and embedded footprint. The embedding procedure is described in Section VI. Section VII concludes the paper.

II. RELATED WORKS

A. FALL DETECTION

Xu *et al.* [4] presents an updated review of FDS, building on the data presented by Habib *et al.* [5] and Mohamed *et al.* [10]. According to the authors, two main trends can be identified in recent literature.

The first main trend is the decline of classic ambient-based FDS [11], [12]—with the exception of newly proposed Kinect-based solutions—and smartphone-based FDS [13] in favor of wearable-device-based solutions. As is, ambient-based systems may affect privacy and can only cover areas within the range of the selected sensors, while smartphone-based systems suffer from power consumption problems. In contrast, wearable, low-power devices, offer better applicability as they can be used regardless of the user location, and are best-suited to work in a low-power usage scenario.

Although recent smartphone- and smartwatch-based FDS have reported battery lifetime in the order of tens of hours [14], [15], [16], specialized hardware might yield more robust and predictable systems in terms of power consumption. In [17], the authors present a wearable module where the battery lifetime is estimated in about forty to fifty hours, depending on the number of alarms.

One of the major drawbacks of specialized hardware is the limited computational power, which often leads to the design of hybrid systems, where only part of the computation is made on-board and the remaining part is delegated to a remote device, for instance a smartphone. A hybrid scenario requires establishing a communication channel to the remote device. Continuous communication, however, tends to drain the battery too quickly for a 24×7 application. For example, [18] and [19] consider prototypes equipped with Bluetooth modules, which deplete the system batteries in four and seven hours respectively. Wi-Fi was used instead in [20] for the communication between a smartphone and a smart-shoe: reportedly, the smartphone battery was completely discharged in about three hours. A more recent architecture employing *nRF* communication is described in [21] and the authors report a battery duration of up to 76 hours.

This paper presents a FDS which, with the use of specialized hardware and network and communication optimization for energy consumption-reduction, can reach more than 130 hours of continuous operation onboard a Micro-Controller

[1]http://bitbucket.org/unipv_cvmlab/sisfalltemporallyannotated.
[2]http://www.st.com/en/evaluation-tools/steval-stlcs01v1.html

Unit (MCU), and possibly up to a few weeks with further optimization.

Nevertheless, automatic FD from wearable sensor data is still considered an open problem, mainly due to the difficulties in achieving a good trade-off between recognition performance, energy efficiency and computational requirements [21].

The second main trend is the rise in adoption of Machine Learning (ML) algorithms [22] instead of statistical methods and traditional digital processing. However, the paper also notes that *no* DL solutions has been employed for online wearable fall detection. The authors speculate that the reason relies on the scarcity of adequately large and properly annotated datasets.

The most widely adopted sensor in wearable devices for FDS is the 3-axis accelerometer, due to its low cost and tiny size. Rotational sensors (i.e., gyroscopes) are used for instance by Bou *et al.* [23], in which very good results are reported for FD with the use of gyroscopes measurements alone. In [24] the authors make use of both accelerometers and gyroscopes readings obtaining "low computational cost and fast response".

The common approach in wearable FDS is to first filter raw acceleration sensor readings and then apply a feature extraction method of some sort to segment out falls from other background activities, also called Activities of Daily Living (ADL). In [8], [25], the occurrence of a fall is detected by comparing statistical indicators like standard deviation of acceleration magnitudes with predefined thresholds.

In [26] the authors compare two different ML techniques: k-Nearest Neighbors (kNN) and Support Vector Machine (SVM), while a simple feed-forward Artificial Neural Network (ANN) is used in [27]. Similarly, the authors of [28] illustrate that a very simple ANN solution is able to detect falls on accelerometer data recorded by a wrist-worn device with near 100 percent sensitivity. In [29], the authors combine different techniques to improve the prediction of the classifier: they investigate the use of ANN, kNN, Radial Basis Function (RBF), Probabilistic Principal Component Analysis (PPCA) and Linear Discriminant Analysis (LDA). In [30] the authors propose a Finite State Machine (FSM) model to extract relevant episodes from input sequences. These episodes are then subdivided into features that are fed to a kNN classifier to distinguish between falls and ADL.

The positioning of the sensor on the body is a critical factor in obtaining a satisfying detection performance. Sensors in specific body positions, in general, allow for better detection performance; for instance, sensors positioned near the center of mass perform much better than sensor located on the subjects' wrist or ankle. The authors of [31] compares statistical indicators, SVM and ANN on several database, obtaining good recognition performance on wrist-collected sensor data.

Most of the above methods extract features from a *sliding window* of a predefined length. Thus, it seems natural to take into account RNN-based solutions, which also make use of sliding windows (see below), to try to improve the FDS performance.

TABLE 1. List of publicly-available datasets for fall detection.

Dataset	Ref.	Number of subjects	Number of activities	Sensing device
DLR	[37]	16	6	Smartphone
tFall	[26]	10	8	Smartphone
Project Gravity	[25]	3	19	Smartphone
MobiFall	[38]	24	13	Smartphone
UniMiB SHAR	[40]	30	17	Smartphone
SisFall	[8]	38	34	Custom
UMAFall	[39]	17	11	Custom

Several DL solutions for offline wearable FD have been presented in the literature in recent years. In [32], a Deep Convolutional Neural Network (DCNN) followed by a RNN layer is proposed, where the DCNN extracts features from sensor signals, while the RNN detects a temporal relationship among the extracted features. Reportedly, however, these approaches are computationally expensive and were performed on a workstation. Nonetheless, in [33] an offline DL solutions is shown to outperform statistical and ML methods in the wrist-based FD. In [34], DCNN, LSTM and ConvLSTM-based architectures are compared for the closely related problem of fall prediction [35].

B. DATASETS

For RNN network supervision, the basic requirements for a dataset of sensor readings are:

1) the availability of the raw, unfiltered signals as read from onboard sensors;
2) the presence of several different falls and ADL;
3) an adequate number of both ADL and falls to train a DL architecture;
4) the availability of temporal annotations (see below).

Many datasets of human activities containing fall events have been proposed in the literature [36]. Almost every one of these contains simulated events re-enacted by young volunteers, which can be challenging to translate into a real-world scenario [4]. Among them, we selected 7 datasets for consideration [8], [25], [26], [37], [38], [39], [40].

Table 1 lists these datasets together with their main features, highlighting the number of different subjects involved in the experiments and the number of different activities performed by volunteers.

After a comparative evaluation, we selected SisFall as the most suitable dataset for this work. The main reasons are the size of the dataset and the hardware used for the recordings– a wearable device that closely resembles the SensorTile device selected for validating the system proposed in this paper (Section VI).

The SisFall dataset includes recordings from a total of 38 volunteers: 23 young subjects and 15 elderly subjects, each performing 34 different activities in a controlled scenario (19 ADL and 15 falls) with several retries, for a total of 4510 complete sequences. Moreover, the set of activities has been validated by a medical staff and each type of tracked activity is described by a prototype video recorded by an instructor. The SisFall dataset was recorded with a custom board including two tri-axial accelerometers and a tri-axial gyroscope, both operating at a frequency of 200 Hz. The recording device was positioned on the waist of the volunteers, with fixed orientation to the body.

The SisFall dataset includes annotations that classify each *whole* activity as either a fall or an ADL. No specific indication is provided about *when* in the sequence of readings a fall occurs, nor about when a particular ADL takes place. For instance, the annotation could read “collapsing into a chair while trying to stand up” without any indication when either the standing up or the collapse occur. Moreover, the sequences may include both a fall event and other ADL sub-sequences, such as walking, sitting, standing, etc.

It is worth mentioning that none of the datasets listed in Table 1 include a temporal annotation of the kind required. As a consequence, none of these dataset can be directly applicable to supervised DL without extending the annotations.

C. DEEP LEARNING FOR EMBEDDED SYSTEM

In general, the end-to-end training of a deep network on an embedded system is unsustainable due to the computational cost involved. In contrast, the embedding of run-time inference modules on microcontrollers is definitely more feasible. Still, specific optimizations are required to make such implementation possible.

A first and obvious optimization strategy is to prune the training model by removing all parameters that are only required for training, such as those required by stateful optimizers.

More sophisticated optimizations described in the literature are parameter compression [41], which includes weight pruning and quantization; weight matrix approximation [42]; weight clustering [43]; and fixed-point representation [44]. Integer quantization is another, even more aggressive strategy for reducing memory occupancy as it could be based on 8-bit integers [44]. This approach could be brought to the limit with ultra-low precision weights, such as binary or ternary representations [45], [46].

Widespread software support for such solutions is only expected to improve in the future. New tools are emerging, such as hardware interfaces (e.g., TVM [47]) or frameworks like TensorFlow Lite,[3] which is available for mobile and high-end embedded device. However, in the instance of ultra-low power MCU, the support from publicly-available frameworks is still very limited, to the point that building custom software may be the only viable strategy, as in our case.

III. DESIGN

In this section, we discuss the overall design of the proposed FDS. We assume that the entire training process is to be performed on a workstation due to its computational cost. We also assume that a highly-optimized, detection-only RNN inference module will be the embedded component to be run on the wearable device. The only outbound communication from the run-time module will be that of issuing a remote notification in case of a fall or other relevant events (e.g., battery discharging), thus entailing an extremely limited transmission traffic.

A. ENHANCED LABELING

As discussed in Section II-B, we selected SisFall dataset [8] as a reference. However, the original SisFall annotation of sequences as a whole is not sufficient to support the training of a RNN method aiming to the real-time detection of falls.

Therefore, the annotations must be made in terms of event-specific time intervals, to suit the training process.

In order to do so, we first defined the following three classes of events to be associated with temporal sub-sequences:

- *FALL*: the time interval in which the person is experiencing an uncontrolled transition towards an unwanted, potentially catastrophic state, i.e., a fall;
- *ALERT*: the time interval in which the person is experiencing an uncontrolled transition towards a desired state, e.g., a near fall such as a stumble followed by a recovery;
- *BKG*: all time intervals in which the person is in control and in a desired state (i.e., background).

The BKG class is intended to contain all daily activities that are not related to a fall, such as walking, going up and down the stairs, sitting on a chair, and so on. The ALERT class has been introduced in the annotations for maximal generality and potential reuse of the dataset for different purposes, such as risk assessment instead of fall detection.

Introducing a third class increases the challenge of correct detection. In the literature, approaches usually deal with two classes (i.e., BKG and FALL). For a fair comparison, the ALERT class can be collapsed into BKG.

The three classes above were used for marking temporal intervals in all SisFall sequences. The implementation of the actual annotation procedure is described in detail in Section IV-A.

B. RNN ARCHITECTURE AND TRAINING

The reference RNN architecture is depicted in Figure 1.

The core of the network is based on one or more LSTM cells stacked in layers [48] (Layers 4 and 6). LSTM cells are popular DL architectures, due to their capability to efficiently capture long-term relationships in the input data [7]. In passing, we also considered the lighter Gated Recurrent Unit (GRU) [49] architecture as an alternative but, in our experiments, such choice entailed a reduction in sensitivity–which is critical for FD–by over 4 percent.

We also considered bidirectional LSTM [50] as well, although we did not follow that line of inquiry since we did not achieve performance improvements worth of the increase in computational complexity and memory occupancy.

The size of the tensors inside each LSTM gate [48], commonly referred to as the inner dimension, is a design

[3]http://www.tensorflow.org/lite

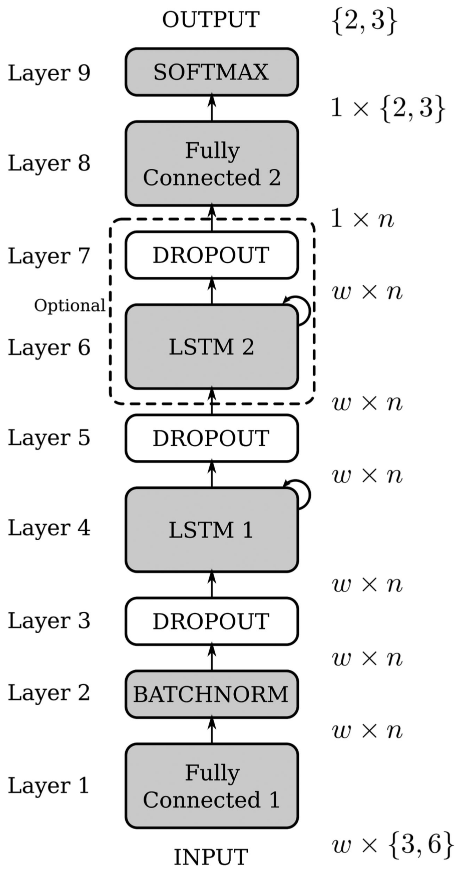


FIGURE 1. The RNN architecture of reference. The input size is determined by the number of sensors (accelerometers and/or gyroscopes), the sensory input dimensions and the size w of the sliding window; n represents the cell inner dimension. The output size depends on the number of classes used in annotating the dataset, namely two (BKG and FALL) or three (BKG, FALL and ALERT). Layers 6 and 7 can be replicated up to k times. Implementation with different values of k and n represents architectural variants of the base model. White blocks are active during the training phase only and are not used in the inference module.

hyperparameter and will be denoted with n from now on. Finding a good trade-off value for n is challenging: on the one hand, n has to be large enough to provide good generalization and prediction capabilities; on the other, its size should be as small as possible, to contain the overall network size.

The number of LSTM layers is also a design hyperparameter and will be referred as k. For the reasons above, in the proposed architecture k is assumed to be in $\{1, 2\}$, as shown in Figure 1.

Variants of the reference architecture considered are characterized by selecting different values of the hyperparameters k and n and by selecting different sensor inputs.

Input signals are preprocessed as explained in Section IV-B. Then, as shown in Figure 1, the input is processed by the fully-connected Layer 1, while a second fully-connected layer (Layer 8) collects the output from the LSTM cell at Layer 4 (or 6) and feeds its output to the final Softmax classifier (Layer 9), which produces the classification according to the number of classes considered: these will be three, when considering also the ALERT class, or two when considering BKG and FALLs only.

The architecture also includes a batch normalization layer [51] (Layer 2), to regularize input data, and three *dropout* layers [52] (Layers 3, 5 and 7). These latter layers are used during training to improve generalization, while they are removed from the inference module embedded on the device.

The training of LSTM cells is based on the idea of temporal unfolding [48], as shown in Figure 2. Temporal unfolding entails that the input sequence (both in training and during online inference) is first partitioned in sub-sequences, called *windows*, having a predefined length of w input samples. This is obtained by sampling each input sequence at fixed periods of length $s \leq w$. The term s will be referred to as *stride*. The values of w and s are both positive integers. Then, each LSTM cell is unfolded layer-wise into exactly w copies of itself, with such copies sharing the same set of

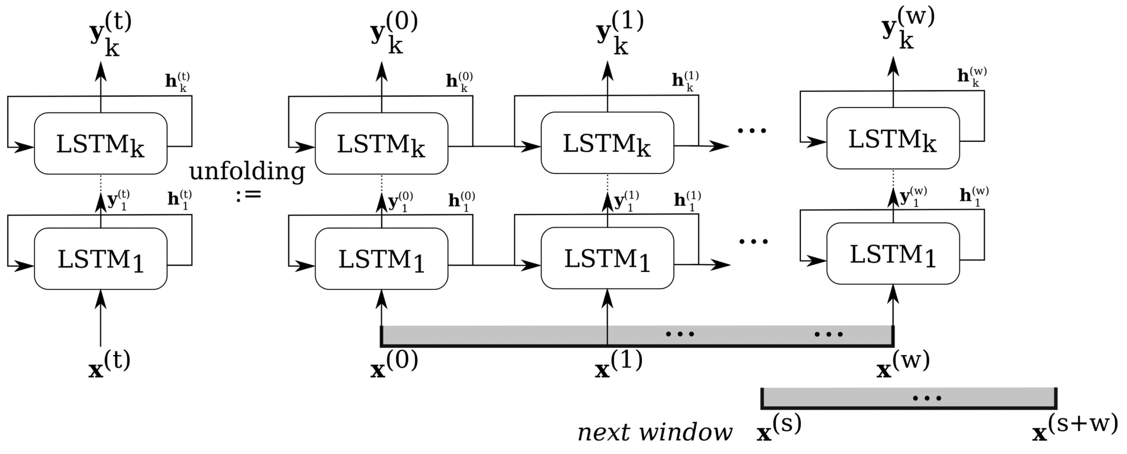


FIGURE 2. Temporal unfolding of a RNN architecture with k LSTM layers. The network is fed with windows containing w samples from the input stream $\mathbf{x}$. A new window is taken each s input signals. The network output is called $\mathbf{y}$, and the internal state is referred as $\mathbf{h}$. For simplicity we assume that the network is composed by LSTM cells only.

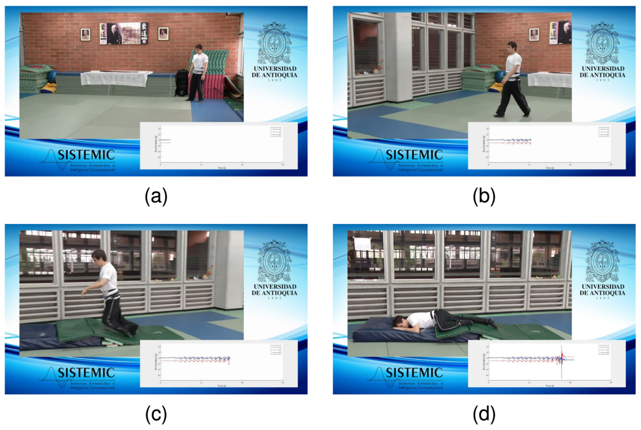


FIGURE 3. **An example of a SisFall [8] prototype video in which an instructor performs a specific activity (fall from a slip). The sequence of sensor readings is superimposed.**

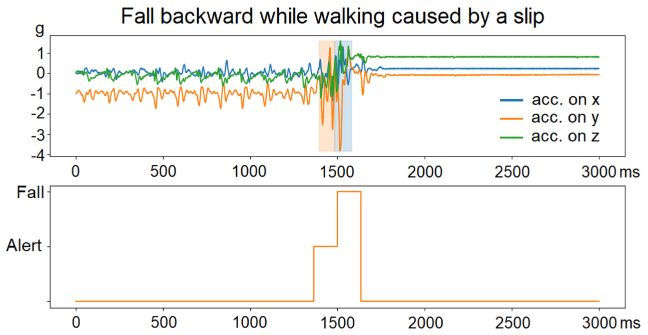


FIGURE 4. **The annotation tool used to enhance the SisFall dataset. A sequence of accelerometer readings (top pane) is marked as either ALERT (in orange) or FALL (in blue). The result is translated into a time series of labels (bottom pane). The tool offers a simple mouse and keyboard interface for batch annotations.**

numerical parameters. Each copy receives two inputs: the output from the previous cell $(h^{(t)})$ in the temporal unfolding at the corresponding layer, and the input signal $x^{(t)}$ from the window at the corresponding index t (Figure 2).

Temporal unfolding allows training of a RNN, like the LSTM, as a non-recurrent deep network, which entails the applicability of all techniques that are typical in the DL field.

As will be described in Section V, the window width w represents another critical hyperparameter for performances. As w decreases, larger parts of interesting events—those marked as FALL or ALERT—do not fit in a single window, making it harder for the network to capture the overall dynamics of the ongoing events. As w increases, windows tend to contain more background than relevant events.

Conversely, the choice of the stride s is dictated by practical considerations. The lower the value of s, the more the training data is augmented with overlapping windows, and thus the higher the training time. Beside a certain threshold, however, there is no guarantee that a lower s will improve network performance.

Please note that the embedded inference module can use a different value for s than the one chosen during training. Using lower values for s implies a more frequent analysis on the input stream. Using higher values for s, on the other hand, entails a reduced computational load on the wearable device. A good choice of s during inference is thus implementation-dependent. For simplicity we used the same value for s in both training and inference.

IV. IMPLEMENTATION

A. ANNOTATION PROCEDURE

As discussed in Section III-A, the annotation task consists of associating a description of the temporal intervals that correspond to the FALL and ALERT classes to each sequence in the SisFall dataset, while the rest is considered BKG by default.

As already mentioned, each activity type included in the SisFall protocol is described by a prototype video (Figure 3). In our work, the ensemble of prototype videos was used to define the criteria for associating temporal intervals corresponding to each class (FALL, ALERT and BKG) to the sequences of sensor readings. These criteria were subsequently used to annotate each sequence in the dataset.

The actual annotation was performed with a custom graphical software tool which allows the visual inspection of a selected sequence and the marking of the temporal intervals corresponding to FALL and ALERT. An example of the graphical interface of the annotation tool is shown in Figure 4.

In the figure, the upper pane shows the sequence of readings for the tri-axial accelerometers. In the same pane, the temporal intervals can be marked using the mouse pointer. The lower pane shows the resulting annotation, as a time series of labels.

The annotation was performed by a team comprising the authors and several colleagues. At the outset, the team agreed on common classification criteria, then each member performed the annotations separately and finally they jointly reviewed them in multiple passages to minimize inter-annotator noise. In defining the criteria and reviewing the annotations, the team received valuable insight by a physician specialized in physiatry.

The extended annotation to the SisFall dataset is publicly available online as an addendum to the original dataset.[4]

B. DATA PREPARATION

First, each raw SisFall sequence was passed through a low-pass Butterworth filter to remove high-frequency noise. Then, all readings were translated into the $[-1, +1]$ range using standard Z-Score normalization. Low-pass filtering is necessary as raw sensor readings often include high-frequency noise and outliers that can affect the normalization procedure; normalization, on the other hand, is critical when using sensor data from different input sources such as gyroscopes and accelerometers.

[4]http://bitbucket.org/unipv_cvmlab/sisfalltemporallyannotated.

TABLE 2. Configuration of the training parameters.

Parameter	Value
Number of epochs	300
Batch size	500
Optimizer	Adam
Learning rate	2.5×10^{-3}
$\mathcal{L}2$ regularization	1.5×10^{-3}
Dropout rate	0.8

Afterwards, each filtered and normalized sequence was divided into windows of width w, sampled with stride s from the sequence itself.

Each window was then to be labeled with one of the three classes: FALL, ALERT or BKG. We performed extensive experimentation with different criteria of classification to translate temporal labels to window labels. Eventually, we adopted the following rules:

- a window containing at least 10 percent of readings within a FALL temporal interval is labeled as FALL;
- otherwise, a window containing a majority of readings within an ALERT temporal interval is labeled as ALERT;
- in any other case, the window is labeled as BKG.

The first criterion was chosen to ensure that the number of FALL windows remained significant even with large values of w. Under our annotation criteria, time intervals marked as FALL were typically lasting just a few hundred milliseconds, and thus could be easily overlooked by any majority-based labeling rule. On the other hand, a smaller percentage entails collecting too few correlated samples in a FALL window. The value of 10 percent was determined experimentally.

C. TRAINING

The typical learning process for a RNN is performed by splitting the dataset into a training and a test set. As already stated, the SisFall dataset includes data acquired from 38 different subjects. To avoid identity bias, each subject in SisFall was arbitrarily assigned to either the training or the test set. As a result, the training dataset included 30 subjects (12 elders), and the test dataset included 8 subjects (3 elders).

Sequences in both training and test sets were translated into labeled windows of width w, according to the method described in Section IV-B. The training set was further divided into train and validation sets using a random 80%/20% split on the number of windows.

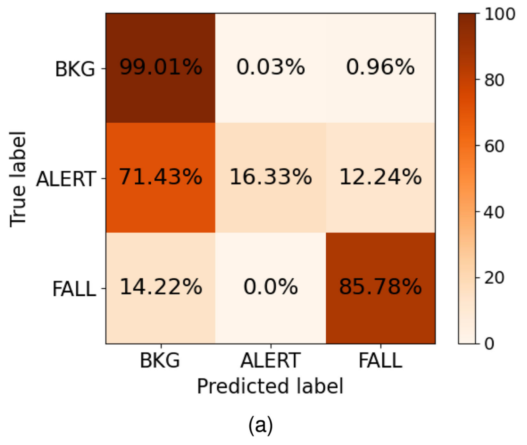
All experiments presented in this paper were performed with the training configuration reported in Table 2, which was determined via an exhaustive grid search to be the best performing in terms of classification.

1) LOSS FUNCTION

With the enhanced SisFall dataset, the number of windows labeled as either BKG, ALERT or FALL was largely unbalanced. Clearly, this is not specific of SisFall but is to be expected for any dataset comprising both ADL and simulated falls, since their respective durations are considerably different. Considering our annotation criteria, the duration of BKG temporal intervals may be up to several seconds, while FALL intervals will be in the range of 500 ms to two seconds. Therefore, depending on the window size w, the number of windows labeled as BKG can be up to two orders of magnitude larger than those labeled as ALERT or FALL.

Class imbalance may lead to substantial inaccuracies in the classification performed by the trained RNN. Figure 5(a) shows the confusion matrix computed on the test set after a training with a typical cross-entropy loss function [53] with $\mathcal{L}2$ regularization and $w = 256$. As expected, BKG activities were classified accurately, whereas FALLs were poorly detected and ALERTs went almost undetected.

To balance this out, we adopted a weighted cross-entropy loss function in which the contribution of each window to the gradient has a weight that is inversely proportional to the size of the corresponding class in the training dataset.



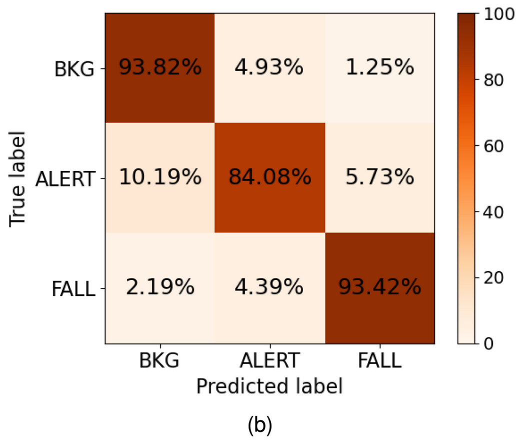


FIGURE 5. Confusion matrices obtained with $w = 256$ and the non-weighted (a) and weighted (b) cross-entropy loss functions. *True labels* on the y-axis correspond to the manual annotations, whereas *predicted labels* are those produced by the RNN classifier. Both confusion matrices were measured on the test set with $k = 2$ and $n = 32$.

To be precise, denoting $\mathcal{L}_i$ the component of the multiclass cross-entropy loss computed on the ith window, we employed the weighted cross-entropy function $\mathcal{L}^W$

$$\mathcal{L}_i = -\sum_{c \in C} y_{i,c} \log(p_{i,c}) \tag{1}$$

$$\mathcal{L}^W = \sum_i m_i \, \mathcal{L}_i, \tag{2}$$

where $C = \{\text{FALL, ALERT, BKG}\}$, y is a binary indicator that is equal to 1 if c is the label associated to the window i and 0 otherwise, p is the probability that the window i belongs to class c as predicted by the given model, and the weight m_i applied to the ith component of the loss is defined as

$$m_i = \begin{cases} 1 & \text{if } y_{i,\text{BKG}} = 1 \\ |\text{BKG}|/|\text{ALERT}| & \text{if } y_{i,\text{ALERT}} = 1 \\ |\text{BKG}|/|\text{FALL}| & \text{if } y_{i,\text{FALL}} = 1 \end{cases}. \tag{3}$$

Figure 5(b) shows the confusion matrix after training with the loss function in Eq. (2). A drastic rise of the accuracy can be observed, especially for the ALERT class.

2) SELECTING WINDOW WIDTH AND STRIDE

The selection of the values for the two critical hyperparameters w and s can be performed with comparative experiments. For instance, Figure 6 shows how the sensitivity of the RNN classifier varies when the window width w is varied between 32 and 1024. The stride values were set to $s = \frac{1}{2}w$. A window width of 256 samples (i.e., 1.28s with a sampling rate of 200 Hz) proved to be the most effective for automatic FD, as can be seen in the figure.

The same procedure was applied to each variant of the base RNN architecture.

Two trends were observed. Lower values of w, as mentioned in Section III-B, imply that only a small portion of an event (i.e., ALERT or FALL) is contained in a window. Thus the network cannot capture the complete dynamic of events and increasingly relies on the absolute amplitude of the signal. As w decreases (especially for $w < 32$), this effect becomes more and more pronounced. Increasingly high values of w, on the other hand, imply that the network is fed with windows containing multiple types of events which, combined with the labeling rule that gives prominence to the FALL class, leads to a network which is unable to distinguish relevant events from the background.

V. RESULTS

A. SELECTING THE MINIMAL CONFIGURATION

The reference RNN was implemented using TensorFlow 1.8 [54] and trained on a Dell 5820 workstation, equipped with an NVIDIA GTX 1080Ti GPU.

Several experiments (average training time of ~ 3.5 hours) were performed in order to identify the combinations of network parameters that produce the minimal footprint in terms of memory occupancy and computational cycles required, while keeping good detection performance.

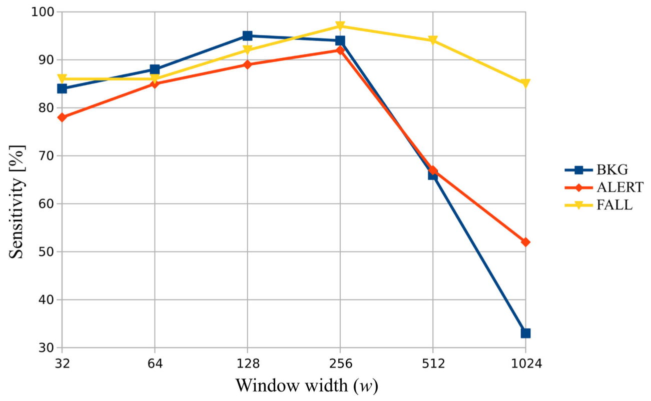


FIGURE 6. Sensitivity of the RNN classifier with $k = 2$ and $n = 32$ for different values of w. The stride value is set to $\frac{1}{2}w$.

The main hyperparameter to be investigated is the number of LSTM layers k. Preliminary experiments showed that using more than two layers ($k > 2$) produced no improvement in terms of accuracy, despite the extra complexity introduced. Therefore only the cases of $k = 1$ and $k = 2$ were considered in full.

Given that most tensor parameters in an LSTM cell are squared, both memory occupancy and computational load grow quadratically with the value of n and only linearly with k [55]. Thus, the inner dimension of each LSTM cell n has a strong impact on both footprints.

Another key aspect to be assessed is the type and combination of sensor signals to feed as input to the network. The options are: 1) accelerometers readings; 2) gyroscope readings; 3) both. The selection of input signals is crucial from a memory occupancy standpoint, even if it has a lesser effect on computational load than either k or n [55].

We performed extended comparative experiments on the annotated SisFall dataset by considering all possible combinations of:

- Input sensor readings: 3D accelerometers, 3D gyroscopes or both.
- Number of LSTM layers: 1 or 2 cells (k);
- Inner LSTM cell size: 4, 8, 16 and 32 (n);

Each experiment was evaluated in terms of the mean accuracy achieved on the three detection classes (BKG, ALERT, and FALL). In order to provide a robust evaluation, we used a 3-fold cross-validation. Figure 7 reports the mean accuracy on the three folds of the validation and the test set.

The experiments are grouped at three levels. At the outermost level they are classified according to the type of sensor data used as input: the first eight figures relate to gyroscope only, the next batch to accelerometers only and the final batch to both. Each batch of eight experiments is then classified according to the number of LSTM layers k. The last classification level pertains to the inner cell size: from left to right, the value of n goes from 4 to 32.

As it can be seen in the figure, there is no significant difference in the performance for $k = 1$ and $k = 2$. However, an architecture with an additional layer clearly requires more

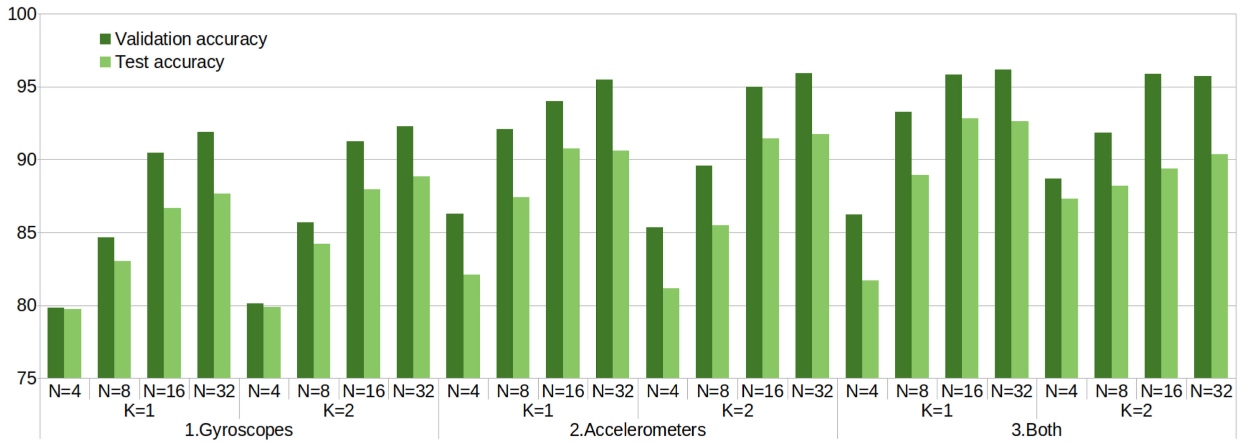


FIGURE 7. Mean accuracy of several variants of the RNN fall detection module. The most effective network configuration is sought for in terms of three main design parameters: type of sensor readings, number of LSTM layers k, inner cell dimension n. The diagram shows the outcome of the experiments for the 24 possible combinations by comparing both validation and test accuracy.

computational cycles for inference, and more memory to store additional network weights. For these reasons, the choice of $k = 1$ is to be preferred.

Considering $k = 1$ only, $n = 16$ and $n = 32$ produce much better test and validation accuracy than lower values of n. However, although the inner dimension of 32 yields better results in validation accuracy, the more relevant test accuracy does not improve meaningfully with respect to $n = 16$.

The results in figure also confirm that, in terms of accuracy, using accelerometers only is indeed more effective than using gyroscope only, as reported in [56]. It is worth noting that, using both types of sensors combined leads to significant improvements (about 2 percent in the best case) yet at the cost of a slight increase in power consumption.

In conclusion, we select the configuration being the most effective in terms of test accuracy as the one with $k = 1$, $n = 16$ using both accelerometer and gyroscope data. From the experience collected, we believe that these same considerations apply to the embedding of any RNN architecture for fall detection.

B. COMPARISON WITH STATISTICAL INDICATORS

To complete the performance assessment of the embeddable RNN architecture, we wanted to perform a comparative of some sort with other applicable techniques. While an extensive study was outside the scope of our work, we wanted to assess our approach against *lighter* techniques. We thus proceeded to do a double comparison: the first one using statistical indicators as benchmarks, the second with a supervised learning approach based on a SVM (Section V-C).

We start by comparing the classification performance of the RNN model to the classification performance of best statistical indicator proposed in the SisFall paper [8], namely

$$C_9 := \sqrt{\sigma^2(a_x) + \sigma^2(a_y) + \sigma^2(a_z)}, \qquad (4)$$

where σ^2 is the variance and a_x, a_y and a_z are respectively the variables representing acceleration readings along the x, y and z axes.

Using this statistical indicator, we can define a C_9 classifier where each sequence of the original SisFall dataset is classified as a fall if at any point the computed value of the C_9 indicator was greater than a predefined threshold, or as an ADL if lower.

In [8], the C_9 indicator was applied to entire sequences and not to specific subsets or windows. Furthermore, C_9 was used for detecting two classes and not three. To make the C_9 indicator and the RNN architecture comparable, we will follow two different approaches:

1) Adapt the C_9 classifier to work on windows and with three classes;
2) Adapt the RNN classifier to work on whole sequences and with two classes.

All the comparisons described in the following were made in terms of sensitivity (SE), specificity (SP) and accuracy (AC), using the standard definitions [53], with the understanding that *positives* represent falls.

1) ADAPTING THE C_9 CLASSIFIER

We applied the C_9 indicator to each window obtained from the preprocessing step described above first by computing the variances σ^2 along each axis and for the same window; then by comparing the value in Eq. (4) with two thresholds, a lower one for ALERT ($T_1 = 0.271$) and an higher one for FALL ($T_2 = 0.701$). Each window was classified as belonging to the class for which the C_9 value was at any point greater than the higher of the two thresholds. The two thresholds were chosen by performing a grid search and selecting those that produced the best classification accuracy over the validation set.

Table 3 shows the classification performance of the adapted C_9 classifier and the RNN classifier. As discussed, the adapted C_9 is applied on windows instead of entire sequences.

TABLE 3. Comparing performance between the adapted C_9 classifier on the entire SisFall dataset and the RNN classifier.

		Adapted C_9	RNN Val.	RNN Test
Sensitivity	BKG	75.01	**93.62**	**93.82**
	ALERT	68.15	**93.78**	**84.08**
	FALL	75.79	**93.81**	**93.41**
Specificity	BKG	92.52	**96.83**	**96.02**
	ALERT	83.30	**95.25**	**95.08**
	FALL	91.57	**98.41**	**98.71**
Accuracy	BKG	83.77	**95.23**	**94.92**
	ALERT	75.73	**94.52**	**89.58**
	FALL	83.68	**96.11**	**96.06**

The SisFall dataset was preprocessed into windows of size $w = 256$ and stride $s = 128$.

2) ADAPTING THE RNN CLASSIFIER

The original C_9 classifier was intended for two classes only [8]. To adapt the RNN classifier accordingly, we needed to train it on a temporally annotated dataset with only two classes: BKG and FALL. In order to do so, we collapsed all ALERT annotations in both the training and test sets into BKG annotations. We then retrained the RNN classifier. With reference to Figure 1, the upmost two layers were adapted to have a two-dimensional output.

We then applied the retrained classifier to whole sequences in the following way: if a sequence contained at least one window identified by the RNN classifier as a FALL, then the entire sequence was classified as a fall; if no windows were classified as FALLs, the entire sequence was classified as an ADL. Table 4 shows the classification performance of both the C_9 classifier and the adapted RNN classifier on the entire SisFall dataset.

C. COMPARISON WITH A SVM CLASSIFIER

We further compared the proposed RNN method with a supervised learning approach based on SVM, which is fully described in [57]. The method was evaluated on the same annotated SisFall dataset to allow an effective comparison between the obtained results.

The authors evaluated a large set of combinations of features to determine the best balance between recall (i.e., sensitivity) and precision. The paper shows that the best performance on different metrics changes sensibly depending on the set of chosen features and a filtering threshold that is used to identify the windows associated to FALL events.

TABLE 4. Comparing performance between the C_9 classifier and the RNN classifier on the original, unmodified SisFall dataset.

	C_9	Adapted RNN
Sensitivity	**97.41**	96.73
Specificity	87.3	**97.15**
Accuracy	91.75	**96.94**

The RNN classifier was adapted to work on entire sequences.

TABLE 5. The performance of the SVM classifier developed in [57] in three different scenarios: optimized for the best sensitivity, specificity or accuracy.

	SVM(sens)	SVM(spec)	SVM(accu)
Sensitivity	99.72	65.02	97.89
Specificity	91.77	99.80	96.99
Accuracy	95.05	85.44	97.93
#features	40	80	14
threshold	1	14	3

The table also reports the number of features and the filtering threshold used to obtain the corresponding results.

Table 5 shows the results of the method in three different scenarios: when the method is optimized for the best sensitivity, specificity or accuracy. The scenarios are denoted with SVM(sens), SVM(spec) and SVM(accu), respectively.

D. DISCUSSION

From Tables 3 and 4 we can see that the RNN classifier significantly outperforms the C_9 classifier on both specificity and accuracy in both cases, especially in the first one, where timely detection was required.

Analyzing the data, the DL architecture is able to capture the high variability of the challenging ALERT class in a way that a simple statistical indicator is unable to.

The only exception is the sensitivity of the second scenario, which is slightly lower to the one obtained with the C_9 classifier. Nevertheless, we do not perceive this as problematic, since we also measured an *event-wise* sensitivity of 100 percent. According to this metric we scored a true positive whenever a fall was detected at least once during the event duration.

Figure 8 shows an example in which the RNN classifier (in red) oscillates and thus the fall event (in yellow) is not detected for the entire duration. We considered that as a successful event-wise detection, since an alarm would be raised anyway.

On the other hand the RNN classifier drastically improves the specificity, that is it minimizes false positives in fall

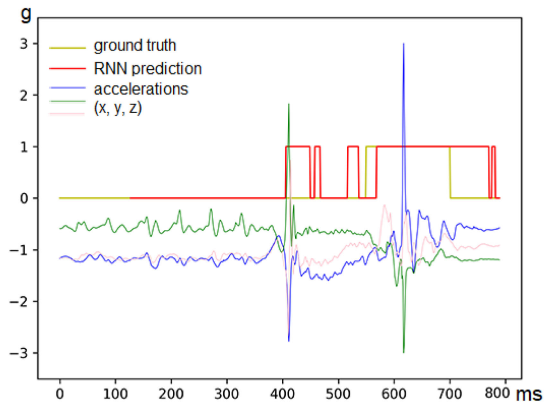


FIGURE 8. Ground truth classification (in yellow) versus RNN classification (in red) on a forward fall preceded by a slip. Although the classifier oscillates, probably due to the hygh dynamic of the slip, the event-wise detection is successful.

detection. This is a desirable property for a wide user adoption in a real-world application.

Table 5 shows that the RNN and SVM classifiers have comparable performance when the SVM is optimized for accuracy, although there is no clear winner among the two. Nevertheless, it is worth mentioning that the SVM approach is considered to be rather hard to implement on an embedded system with the characteristics described in Section VI due to the large number of features and the ensuing requirements in terms of memory and processing power [57].

Overall, the detection performances of the RNN classifier are very satisfactory, and prove that an online DL approach can be effective in a realistic scenario.

VI. DEVICE EMBEDDING

Once a minimal RNN architecture has been identified using the process described in Section V-A, the final step for a complete implementation is to embed it on the target wearable device.

In order to assess the feasibility of embedding our RNN model architecture, we chose a hardware device of reference. The SensorTile board integrates a wide array of sensors, including 3D accelerometers and gyroscopes; it is connected to a 100+ mAh Li-Ion battery and mounts a BLE radio module for wireless communication. The SensorTile embeds an ultra-low power ARM Cortex M4 MCU running at 80 MHz with 128 KB of RAM.[5]

The embedded implementation was performed by manually porting a TensorFlow-compatible implementation of the reference RNN model to the Cortex MCU, harnessing the highly-optimized CMSIS library.[6] The original numerical representation in 32-bit floating-point adopted with TensorFlow was preserved, in order to keep the same classification performance with respect to the workstation implementation. Note that the embedding only concerns the inference capabilities of the model. Thus, with reference to Figure 1, only the blocks in grey were ported.

The implementation details of the run-time module are described in [55] and are not repeated here.

A. VALIDATION

The embedded implementation was validated in the following way:

- The chosen network was trained on a workstation using the training set defined in Section IV-C;
- Inference was made using the TensorFlow implementation on the workstation on the test set defined in Section IV-C;
- Inference was then made using the embedded implementation on the device on the same test set;
- Finally, the numerical values produced during both inference processes were compared.

[5]http://www.st.com/en/evaluation-tools/steval-stlcs01v1.html
[6]https://www2.keil.com/mdk5/cmsis.

The resulting mean squared numerical error is in the order of 10^{-7} on the entire test set, meaning that the output of the embedded implementation is effectively indistinguishable from the output of the workstation implementation.

B. MEMORY OCCUPANCY AND BATTERY DURATION

The authors of [55] derived a general method to estimate memory occupancy and battery duration, in a scenario where the optimized FD implementation is ran online 24×7.

Applying the same formulas, we can derive memory occupancy and computational workload for the minimal network selected in Section V-A with $k = 1$ and $n = 16$. This results in a memory footprint of less than 18.5 KB, and a processing time per window of 0.051 s. These measures shows the suitability of the RNN architecture for embedded real-time processing.

Furthermore, by using the STM32CubeMX Power Consumption Calculator, we were able to estimate that a wearable device running the minimal architecture could be operative, with a battery of 100 mAh, for about 132 hours without recharging, going well beyond the minimum requirement of a single-day autonomy for a practical application.

Using accelerometers as the only input data slightly modifies these results: the processing time decreases by 0.001 s, the memory occupation decreases by 2.5 KB and the autonomy increases by 4 hours. In our opinion, these figures justify the choice of using both accelerometers and gyroscopes to improve the classification performance.

C. VARIANCE FILTER

The results presented so far assume that the DL inference module will be running continuously on the MCU. We believe this is not necessary in order to maintain satisfactory performance because in a real-life scenario, many sequences of data readings will be produced when the user is at rest or performing low-dynamic activities; such sequences could be safely classified as background by a simple variance filter.

A precise estimation of the filter threshold and of the trade-off between the loss in prediction accuracy and the gain in battery duration would require further experiments "in the wild". Still, a crude formula for the estimation of the difference in battery duration B can be inferred by assuming that the execution time of the variance filter is negligible with respect to the RNN module

$$B_{new} = \frac{B_{prev}}{1 - w_{filtered}}, \tag{5}$$

where $w_{filtered} \in [0, 1)$ is the fraction of filtered windows with respect to the total number of processed windows. Thus, the battery duration grows hyperbolically as more windows are filtered, and it can potentially last for several weeks in a real-world application.

This line of research has been followed by some of the present authors in [58] on the same SensorTile MCU employed in this work. The authors included two preprocessing filters in the inference pipeline. Both filters use an optimized implementation of the Welford algorithm to compute the variance of a given window before the input to the RNN architecture. If the variance is below either threshold, the window is ignored and could never generate an alarm.

Table 1 in [58] shows that the power consumption of the variance filters is several orders of magnitude below the one of the embedded inference module, thus confirming the assumption at the base of the above formula.

VII. CONCLUSIONS AND FUTURE WORKS

This work discusses the design and implementation of a DL technique for human fall detection, targeted to a resource-constrained wearable device.

A reference RNN architecture is proposed, based on LSTM cells. Several architectural variants are discussed and comparatively analyzed, looking for a viable trade-off among detection performance, power consumption and architectural complexity.

Detection capabilities were evaluated on the publicly-available SisFall dataset. In order to make it suitable for supervised learning, temporal annotations were added manually for three classes of relevant events: BKG (background), ALERT and FALL.

The specificity, sensitivity and accuracy results obtained on a unbiased test set derived from SisFall show that the minimal RNN architecture can reach comparable performances to the state-of-art.

The feasibility of MCU embedding was assessed with an actual implementation of the run-time detection module of the RNN model for the SensorTile board. Experimental data on both memory occupation and battery duration show the viability of the proposed approach for online real-time processing that can last for several days.

The results presented suggest a suitable design strategy for FD using a DL approach. Namely, that of finding the simplest network architecture that could meet the desired performance levels, while leaving a limited footprint in terms of computation power and memory required.

The experience made during this work revealed the lack of a complete and extensive datasets with appropriate annotations. For this reasons, we have begun collecting a new dataset of simulated falls made with a body network of multiple wearable sensors connected to a gateway via BLE. We are also experimenting augmentation techniques such as rotations and oscillations of the recorded data that should improve the quality of classification. To ease the task of annotation, each activity performed by the volunteers will be associated to a video recording, so that temporal intervals can be identified by looking at the body posture of the volunteer instead of at the signals themselves. In our view, this extended dataset will allow a more careful tuning of the network architecture with the purpose of further improving the embedded implementation for wearable devices. Once such implementation will have been achieved, tests with real subjects in real-life scenarios will be planned and conducted.

ACKNOWLEDGMENTS

The authors acknowledge the financial support from Regione Lombardia, under the projects *“Home of IoT”* (ID: 139625, co-funded by POR FESR 2014-2020) and *“Intelligent Personal Health Safety Domotic Monitoring*“ (IPHSDM) (ID 379273, “Smart Living”). The authors would also like to thank Antonio Nardone, PhD MD, for his valuable suggestions about the labelling procedure, and Michela Pompei for her linguistic revision.

REFERENCES

[1] World HealthOrganization, *WHO Global Report on Falls Prevention in Older Age*. Geneva, Switzerland: World Health Organization, 2008.

[2] J. Fleming and C. Brayne, “Inability to get up after falling, subsequent time on floor, and summoning help: Prospective cohort study in people over 90,” *Brit. Med. J.*, vol. 337, pp. 1279–1282, 2008.

[3] C.-N. Huang, C.-Y. Chiang, G.-C. Chen, S. Hsu, W.-C. Chu, and C.-T. Chan, “Fall detection system for healthcare quality improvement in residential care facilities,” *J. Med. Biol. Eng.*, vol. 30, no. 4, pp. 247–252, 2010.

[4] T. Xu, Y. Zhou, and J. Zhu, “New advances and challenges of fall detection systems: A survey,” *Appl. Sci.*, vol. 8, no. 3, 2018, Art. no. 418.

[5] M. Habib, M. Mohktar, S. Kamaruzzaman, K. Lim, T. Pin, and F. Ibrahim, “Smartphone-based solutions for fall detection and prevention: Challenges and open issues,” *Sensors*, vol. 14, no. 4, pp. 7181–7208, 2014.

[6] E. Torti *et al.*, “Embedding recurrent neural networks in wearable systems for real-time fall detection,” *Microprocessors Microsyst.*, vol. 71, 2019, Art. no. 102895.

[7] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” 2015, *arXiv:1506.00019*.

[8] A. Sucerquia, J. López, and J. Vargas-Bonilla, “SisFall: A fall and movement dataset,” *Sensors*, vol. 17, no. 1, 2017, Art. no. 198.

[9] A. Sucerquia, J. D. López, and J. F. Vargas-Bonilla, “Real-life/real-time elderly fall detection with a triaxial accelerometer,” *Sensors*, vol. 18, no. 4, 2018, Art. no. 1101.

[10] O. Mohamed, H. J. Choi, and Y. Iraqi, “Fall detection systems for elderly care: A survey,” in *Proc. 6th Int. Conf. New Technol. Mobility Secur.*, 2014, pp. 1–4.

[11] G. Baldewijns, G. Debard, G. Mertes, T. Croonenborghs, and B. Vanrumste, “Improving the accuracy of existing camera based fall detection algorithms through late fusion,” in *Proc. 39th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, 2017, pp. 2667–2671.

[12] M. Kepski and B. Kwolek, “Fall detection using ceiling-mounted 3D depth camera,” in *Proc. Int. Conf. Comput. Vis. Theory Appl.*, 2014, pp. 640–647.

[13] E. Casilari, R. Luque, and M.-J. Morón, “Analysis of android device-based solutions for fall detection,” *Sensors*, vol. 15, no. 8, pp. 17 827–17 894, 2015.

[14] M. Ángel Álvarez de la Concepción, L. M. S. Morillo, J. A. Álvarez Garcо́a, and L. González-Abril, “Mobile activity recognition and fall detection system for elderly people using Ameva algorithm,” *Pervasive Mobile Comput.*, vol. 34, pp. 3–13, 2017.

[15] F. J. González-Cañete and E. Casilari, “Consumption analysis of smartphone based fall detection systems with multiple external wireless sensors,” *Sensors*, vol. 20, no. 3, 2020, Art. no. 622.

[16] I. Maglogiannis, C. Ioannou, G. Spyroglou, and P. Tsanakas, “Fall detection using commodity smart watch and smart phone,” in *Proc. IFIP Int. Conf. Artif. Intell. Appl. Innov.*, 2014, pp. 70–78.

[17] F. Luna-Perejón, M. J. Domо́nguez-Morales, and A. Civit-Balcells, “Wearable fall detector using recurrent neural networks,” *Sensors*, vol. 19, no. 22, 2019, Art. no. 4885.

[18] C. Wang, W. Lu, M. R. Narayanan, S. J. Redmond, and N. H. Lovell, “Low-power technologies for wearable telecare and telehealth systems: A review,” *Biomed. Eng. Lett.*, vol. 5, no. 1, pp. 1–9, 2015.

[19] Y. Hou, N. Li, and Z. Huang, “Triaxial accelerometer-based real time fall event detection,” in *Proc. Int. Conf. Inf. Soc.*, 2012, pp. 386–390.

[20] A. J. A. Majumder, I. Zerin, S. I. Ahamed, and R. O. Smith, “A multi-sensor approach for fall risk prediction and prevention in elderly,” *ACM SIGAPP Appl. Comput. Rev.*, vol. 14, no. 1, pp. 41–52, 2014.

[21] T. N. Gia *et al.*, “Energy efficient wearable sensor node for IoT-based fall detection systems,” *Microprocessors Microsyst.*, vol. 56, pp. 34–46, 2018.

[22] N. Zerrouki, F. Harrou, A. Houacine, and Y. Sun, “Fall detection using supervised machine learning algorithms: A comparative study,” in *Proc. 8th Int. Conf. Modelling Identification Control*, 2016, pp. 665–670.

[23] A. Bourke and G. Lyons, “A threshold-based fall-detection algorithm using a bi-axial gyroscope sensor,” *Med. Eng. Phys.*, vol. 30, no. 1, pp. 84–90, 2008.

[24] Q. Li, J. A. Stankovic, M. A. Hanson, A. T. Barth, J. Lach, and G. Zhou, “Accurate, fast fall detection using gyroscopes and accelerometer derived posture information,” in *Proc. 6th Int. Workshop Wearable Implantable Body Sensor Netw.*, 2009, pp. 138–143.

[25] T. Vilarinho *et al.*, “A combined smartphone and smartwatch fall detection system,” in *Proc. IEEE Int. Conf. Comput. Inf. Technol.*, 2015, pp. 1443–1448.

[26] C. Medrano, R. Igual, I. Plaza, and M. Castro, “Detecting falls as novelties in acceleration patterns acquired with smartphones,” *PLoS One*, vol. 9, no. 4, pp. 1–9, Apr. 2014.

[27] S. Abbate, M. Avvenuti, F. Bonatesta, G. Cola, P. Corsini, and A. Vecchio, “A smartphone-based fall detection system,” *Pervasive Mobile Comput.*, vol. 8, no. 6, pp. 883–899, 2012.

[28] S. Yoo and D. Oh, “An artificial neural network–based fall detection,” *Int. J. Eng. Bus. Manage.*, vol. 10, 2018, Art. no. 1847979018787905.

[29] R. M. Gibson, A. Amira, N. Ramzan, P. C. de-la Higuera, and Z. Pervez, “Multiple comparator classifier framework for accelerometer-based fall detection and diagnostic,” *Appl. Soft Comput.*, vol. 39, pp. 94–103, 2016.

[30] P. Tsinganos and A. Skodras, “A smartphone-based fall detection system for the elderly,” in *Proc. 10th Int. Symp. Image Signal Process. Anal.*, 2017, pp. 53–58.

[31] S. B. Khojasteh, J. R. Villar, C. Chira, V. M. González, and E. de la Cal, “Improving fall detection using an on-wrist wearable accelerometer,” *Sensors*, vol. 18, no. 5, 2018, Art. no. 1350.

[32] F. J. Ordóñez and D. Roggen, “Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition,” *Sensors*, vol. 16, no. 1, 2016, Art. no. 115.

[33] T. Mauldin, M. Canby, V. Metsis, A. Ngu, and C. Rivera, “SmartFall: A smartwatch-based fall detection system using deep learning,” *Sensors*, vol. 18, no. 10, 2018, Art. no. 3363.

[34] A. N. Aicha, G. Englebienne, K. van Schooten, M. Pijnappels, and B. Kröse, “Deep learning to predict falls in older adults based on daily-life trunk accelerometry,” *Sensors*, vol. 18, no. 5, 2018, Art. no. 1654.

[35] R. Rajagopalan, I. Litvan, and T.-P. Jung, “Fall prediction and prevention systems: Recent trends, challenges, and future research directions,” *Sensors*, vol. 17, no. 11, 2017, Art. no. 2509.

[36] E. Casilari, J.-A. Santoyo-Ramón, and J.-M. Cano-Garcõa, “Analysis of public datasets for wearable fall detection systems,” *Sensors*, vol. 17, no. 7, 2017, Art. no. 1513.

[37] K. Frank, M. J. Vera Nadales, P. Robertson, and T. Pfeifer, “Bayesian recognition of motion related activities with inertial sensors,” in *Proc. 12th ACM Int. Conf. Adjunct Papers Ubiquitous Comput.*, 2010, pp. 445–446.

[38] G. Vavoulas, M. Pediaditis, C. Chatzaki, E. Spanakis, and M. Tsiknakis, “The MobiFall dataset: Fall detection and classification with a smartphone,” *Int. J. Monit. Surveillance Technol. Res.*, vol. 2, no. 1, pp. 44–56, 2014.

[39] E. Casilari, J. A. Santoyo-Ram ón, and J. M. Cano-Garc õa,“UMAFall: A multisensor dataset for the research on automatic fall detection,” *Procedia Comput. Sci.*, vol. 110, pp. 32–39, 2017.

[40] D. Micucci, M. Mobilio, and P. Napoletano, “UniMiB SHAR: A dataset for human activity recognition using acceleration data from smartphones,” *Appl. Sci.*, vol. 7, no. 10, 2017, Art. no. 1101.

[41] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in *Proc. 4th Int. Conf. Learn. Representations*, 2016.

[42] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, “Predicting parameters in deep learning,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 2148–2156.

[43] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” 2014, *arXiv:1412.6115*.

[44] S. Anwar, K. Hwang, and W. Sung, “Fixed point optimization of deep convolutional neural networks for object recognition,” in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2015, pp. 1131–1135.

[45] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” *Adv. Neural Inf. Process. Syst.*, pp. 4107–4115, 2016.

[46] K. Hwang and W. Sung, “Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1,” in *Proc. IEEE Workshop Signal Process. Syst.*, 2014, pp. 1–6.

[47] T. Chen *et al.*, “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[48] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[49] K. Cho *et al.*, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1724–1734.

[50] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[51] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, vol. 37, pp. 448–456.

[52] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” 2012, *arXiv:1207.0580*.

[53] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[54] M. Abadi *et al.*,“TensorFlow: A system for large-scale machine learning,” in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[55] E. Torti *et al.*, “Embedded real-time fall detection with deep learning on wearable devices,” in *Proc. 21st Euromicro Conf. Digit. Syst. Des.*, 2018, pp. 405–412.

[56] Q. T. Huynh, U. D. Nguyen, L. B. Irazabal, N. Ghassemian, and B. Q. Tran, “Optimization of an accelerometer and gyroscope-based fall detection algorithm,” *J. Sensors*, vol. 2015, 2015, Art. no. 452078.

[57] D. Giuffrida, G. Benetti, D. De Martini, and T. Facchinetti, “Fall detection with supervised machine learning using wearable sensors,” in *Proc. 17th IEEE Int. Conf. Ind. Informat.*, 2019, pp. 253–259.

[58] E. Torti, M. Musci, F. Guareschi, F. Leporati, and M. Piastra, “Deep recurrent neural networks for edge monitoring of personal risk and warning situations,” *Sci. Program.*, vol. 2019, 2019, Art. no. 9135196.

MIRTO MUSCI received the PhD degree in computer engineering from the University of Pavia, Pavia, Italy, in 2014. He is a research assistant with the Computer Vision and Multimedia Lab, University of Pavia, under the supervision of Professor Virginio Cantoni. He also teaches parallel programming with the University of Pavia. He is the author of more than 20 publications in international journals and conference proceedings. His main research interest include parallel computing in bioinformatics and deep learning for assisted living.

DANIELE DE MARTINI received the PhD degree from the Università degli Studi di Pavia, Pavia, Italy. He is a postdoctoral research assistant with the Oxford Robotics Institute, part of the Department of Engineering Science, University of Oxford, and a junior research fellow of Kellogg College. He joined the Oxford Robotics Institute in 2018 and since then his work focused on autonomous vehicles, with particular interest in innovative solutions that can lead to the wide adoption of radar technology as main sensor modality.

NICOLA BLAGO received the master's degree in computer engineering from the University of Pavia, Pavia, Italy, in 2017. He currently works as a professional programmer in the field of artificial intelligence.

TULLIO FACCHINETTI is a researcher with the Department of Industrial, Information and Biomedical Engineering, University di Pavia. He is heading the research activities with the Robotics Laboratory on mobile robotics, embedded systems and smart sensors, indoor localization, home automation and energy efficiency, internet of things, and machine learning. He published more than 80 articles in international peer reviewed journals and conference proceedings on these subjects. He is inventor of one international patent.

MARCO PIASTRA received the PhD degree in electronics and computer engineering from the University of Pavia, Pavia, Italy, in 2002. He is professor of artificial intelligence and of deep learning with the University of Pavia and of machine learning with the University of Brescia. 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. His research interests include now focused on applied deep learning and deep reinforcement learning to several application fields in industry, including embedded systems, and robotics.