Università degli Studi di Pavia
Facoltà di Ingegneria
Dipartimento di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Computer Engineering

Fall Detection with Smartphones Using Recurrent Neural Networks

Tesi di Laurea Magistrale di:
Nicola Blago

Relatore:
Prof. Marco Piastra

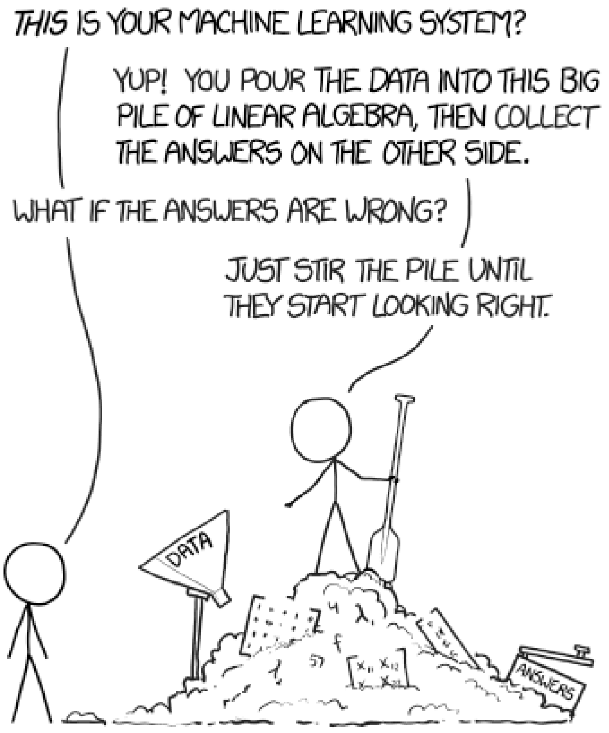
Correlatori:
Dott. Mirto Musci
Ing. Mario Betti

A.·.A.·. 2016/2017

Abstract

This thesis investigates possible approaches of machine learning to automatic fall detection, with a focus on deployment on smartphones. After introducing the theoretical background for both fall detection and machine learning, a summary of the available tools is presented, followed by the rationale behind our research choices.

A detailed description of the experiments made is provided; all the steps followed to achieve a trained neural network capable of detecting falls are thoroughly illustrated, starting from the existing datasets to the creation of a new one. The project also focuses on the development of an Android application capable of using such trained network.


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

(https://xkcd.com/1838/)

Contents

Introduction

Nowadays machine learning is one of the most promising and discussed topics, both by the scientific community and by the general public. Machine learning 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.
These impressing improvements have been possible in recent years due to the interest in these techniques performing often with super-human accuracy.

The increasing popularity of machine learning will bring these techniques in many aspects of daily life.

In particular, unintentional falls are part of our daily life, especially inside the elderly population or any person with frailties. Falls are a major health problem, expensive, and statistically unavoidable.
On the other hand, automatic fall detection is still an open problem, since machines cannot easily recognize a fall.

Over the past years, several falls monitoring systems have been proposed in the literature. Ambient-based sensors, such as cameras, are among the possible solutions. Other proposed approaches are based on wearable devices, usually using a triaxial accelerometer.

Android devices, covering about 88% of the smartphone market, all possess at least one accelerometer. This ubiquitous distribution makes it an ideal target for the development of a fall detection application.

The aim of the project described in this thesis is to study possible approaches of machine learning to automatic fall detection, with the hope to get improved efficiency, accuracy and economic benefit.
Specifically, we are investigating a special kind of neural networks, the "Long Short-Term Memory" (LSTM), specialized in processing signals that vary

over time.

The ultimate goal of the project is to:

- Provide a new dataset of smartphone-acquired accelerometer signals representing falls and activities of daily living.
- Use this dataset to train a neural network.
- Deploy the trained network on a smartphone, to perform real-time fall detection.

Organization

The thesis is structured as follows:

- In **Chapter 1** we introduce the background theory regarding the automatic fall detection techniques and the state of the art, specifically the SisFall dataset. We also introduce machine learning, focusing in particular on recurrent neural networks and LSTM.
- In **Chapter 2** a summary and evaluation of the tools explored is presented, as well as the reasons that lead us to our choices in terms of smartphone operating systems and software for machine learning.
- In **Chapter 3** we present the experiments we performed and the transition from the SisFall dataset to a newer one and the architecture we chose to use.
- In **Chapter 4** we present the results obtained and an outline of the future steps.
- In **Chapter 5** we will draw our conclusions.

Chapter 1

Theory

1.1 Fall Detection

1.1.1 Motivations

Unintentional falls are the leading cause of fatal injury and the most common cause of nonfatal trauma-related hospital admissions among older adults. One in four people aged over 65 years old falls every year. Falls result in more than 2.8 million injuries treated in emergency departments annually, including over 800,000 hospitalizations and more than 27,000 deaths. In 2014, the total cost of fall injuries was \$31 billion. The financial toll is expected to increase as the population ages and may reach \$67.7 billion by 2020.[1]

Elderly people are not the only group that is heavily affected by unintentional falls: every person with some sort of fragility (for example: any kind of mild disability or even post-operative patients) is part of similar statistics. Worse yet, many of these people often live alone, so they may not receive immediate assistance if an accident occurs.

Sadly, falls are statistically inevitable, even with prevention and risk reduction strategies. A good approach is then the remote home-care: it aims at reducing hospitalization as much as possible (and to improve the quality of life doing so). Remote home-care, however, requires constant monitoring, since a fall is not predictable, and a fast response time is essential.
Given these premises, our approach will be to study the technical aspects of using intelligent and interconnected devices for automatic falls monitoring.

[1]`https://www.ncoa.org/news/resources-for-reporters/get-the-facts/falls-prevention-facts/`

1.1.2 Approaches

A human being can easily recognize a fall when the falling subject is in sight, but for a machine it might be difficult, because a fall is not a direct biometric reading (like for example an heartbeat).
Automatic fall detection techniques are usually divided into two main categories: wearable device-based and ambient-based. Ambient-based sensors are mainly based on video cameras but these techniques are intrusive in terms of privacy and do not solve the problem for independent adults, who are not confined to closed spaces. Besides, even with deep learning methods (Feng et al. (2014)) that of automatic fall detection might still be a difficult problem to address.
Wearable devices instead offer portability as they can be used regardless of the user location, as long as a network connection can be guaranteed. The preferred sensor is almost always the triaxial accelerometer due to its low cost, small size, and since it is built-in in almost all smartphones. Smartphones are interesting candidates as wearable sensors since they include a robust hardware, a powerful processor, they are economically affordable, and the whole fall detection system could be implemented in a single app.
When used for monitoring purposes, however, a smartphone has relevant limitations, like cost and battery durability. The ideal solution might then be then a "body network" composed of various intelligent sensors (sensors capable of on-chip computation).

1.1.3 State of the Art

Automatic fall detection from wearable sensor data is still an open problem, and various studies have approached it. The common procedure of existing studies is to record the raw acceleration sensor reading, filter them somehow and then apply a feature extraction method which classifies activities as falls or activities of daily living (ADL).

For our purposes, we were looking for datasets in literature meeting some basic requirements: all activities are well documented, the raw data is freely available, the dataset contains both falls and ADL, and it is reported in a peer-reviewed paper.
Five of such datasets were then considered:

- **DLR** (Frank et al. (2010)).
  Sixteen subjects (23 to 50 years old). They recorded six types of ADL, and the authors did not specify the conditions of the falls (they belong to a single group). The files are too short for some types of analysis.
- **tFall** (Medrano et al. (2014)).
  Ten participants between 20 and 42 years old. They recorded eight types of falls (503 total recordings with two smartphones), and one week of continuous ADL recordings with all participants carrying smartphones in the pockets and a handbag. The ADL trials were not identified by activity.
- **Project gravity** (Vilarinho et al. (2015)).
  Three participants (ages 22, 26, and 32) performed 12 types of falls and seven types of ADL with a smartphone in the pocket.
- **MobiFall** (Vavoulas et al. (2016)).
  Twenty-four volunteers (22 to 42 years old) performed nine types of ADL and four types of falls using a Samsung Galaxy smartphone. Nine subjects performed falls and ADL, while 15 performed only falls (three trials each).
- **SisFall** (Sucerquia et al. (2017)).
  Twenty-three young adults performing 19 ADL and 15 fall types, fourteen healthy and independent elders (over 62 years old) performing 15 ADL types, one 60 years old participants that performed all ADL and falls. All the data is acquired with a self-developed device composed of two types of accelerometers and one gyroscope.

For our purposes, the SisFall dataset was the most interesting, as it includes elderly people, and with a great variety of activities and number of subjects. In addition, all the other listed datasets acquired data using smartphones, while SisFall adopts a special, purpose-built device fixed on body as a belt buckle (see Figure 1.1).

1.2 The SisFall Dataset

1.2.1 Selection of Activities

To determine which falls were to be performed, in addition to those commonly tested in the literature, the SisFall team took a survey among elderly people. As reported by Sucerquia et al. (2017), the survey consisted of three main questions: For each fall incident:

- Which activity were you performing when the fall happened?
- What produced the fall? A sliding, a faint, a trip, other?
- In which orientation did the fall happen? What part of the body received the impact?

ADL were selected based on common activities, activities that are similar (in acceleration waveform) to falls, and activities with high acceleration that can generate false positives.
The two following tables (Falls in Table 1.1 and ADL in Table 1.2) list all the activities included in the dataset.

Code	Activity
F01	Fall forward while walking caused by a slip
F02	Fall backward while walking caused by a slip
F03	Lateral fall while walking caused by a slip
F04	Fall forward while walking caused by a trip
F05	Fall forward while jogging caused by a trip
F06	Vertical fall while walking caused by fainting
F07	Fall while walking, with use of hands in a table to dampen fall, caused by fainting
F08	Fall forward when trying to get up
F09	Lateral fall when trying to get up
F10	Fall forward when trying to sit down
F11	Fall backward when trying to sit down
F12	Lateral fall when trying to sit down
F13	Fall forward while sitting, caused by fainting or falling asleep
F14	Fall backward while sitting, caused by fainting or falling asleep
F15	Lateral fall while sitting, caused by fainting or falling asleep

Table 1.1: Types of falls selected for SisFall.

1.2.2 Partecipants

The thirty-eight volunteers were divided into two groups: elderly people and young adults. Elderly people group was formed by fifteen participants (eight males and seven females), and the young adults group was formed by twenty-three participants (eleven males and twelve females). The elderly people were all healthy and independent, and none of them presented gait problems.

Code	Activity
D01	Walking slowly
D02	Walking quickly
D03	Jogging slowly
D04	Jogging quickly
D05	Walking upstairs and downstairs slowly
D06	Walking upstairs and downstairs quickly
D07	Slowly sit in a half height chair, wait a moment, and up slowly
D08	Quickly sit in a half height chair, wait a moment, and up quickly
D09	Slowly sit in a low height chair, wait a moment, and up slowly
D10	Quickly sit in a low height chair, wait a moment, and up quickly
D11	Sitting a moment, trying to get up, and collapse into a chair
D12	Sitting a moment, lying slowly, wait a moment, and sit again
D13	Sitting a moment, lying quickly, wait a moment, and sit again
D14	Being on one's back change to lateral position, wait a moment, and change to one's back
D15	Standing, slowly bending at knees, and getting up
D16	Standing, slowly bending without bending knees, and getting up
D17	Standing, get into a car, remain seated and get out of the car
D18	Stumble while walking
D19	Gently jump without falling (trying to reach a high object)

Table 1.2: Types of ADL selected for SisFall.

Young adults performed ADL and falls, while elderly people did not perform falls and activities D06, D13, D18, and D19 from Table 1.2. Additionally, some elderly people did not perform some activities due to personal impairments or medical recommendation. The participant of 60 years old identified by code SE06, who is an expert in Judo simulated both falls and ADL.

1.2.3 Experimental Set-Up

The dataset was recorded with a self-developed embedded device composed of two accelerometers and one gyroscope. The SisFall team only used the ADXL345 accelerometer for their fall detection algorithm, but the data recorded with the other two sensors were also made publicly available.
The device was fixed to the waist of the participants, with the positive z-axis in the forward direction, the positive y-axis in the gravity direction, and the positive x-axis pointing to the right side of the participant (Figure 1.1).
All tests were performed with the same frequency sample of 200 Hz.

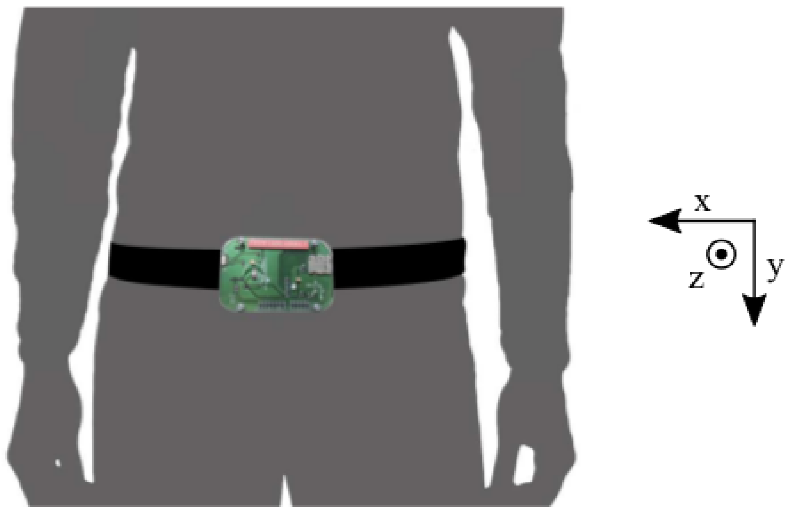


Figure 1.1: Device used by SisFall for acquisition.

1.2.4 Fall Detection Algorithms

The SisFall team also developed some algorithms for fall detection, following the common pipeline to process the data: preprocessing, feature extraction, classification, and validation.

- **Preprocessing**.
  The preprocessing stage consisted of a 4th order IIR Butterworth low-pass filter, used to reduce high frequency noise.

- **Feature extraction**.
  Fourteen features were defined (identified with codes C_1 through C_{14}) that presented a good overall performance in separating ADL from falls. Those features were divided in five groups: amplitude, orientation angle, statistical moments, critical phase time, and area under the curve. Each feature is calculated over a *sliding window* of signal samples.
  A sliding window of a signal (See Figure 1.2) is a set of values from the signal, and it is defined by two parameters:

  - N, the *window length*, that determines how many samples are present inside the window.
  - s, the *stride*, that indicates the distance between the beginning of a window and the beginning of the next one.

- **Classification**.
  The classification algorithm used to discriminate ADL from falls is a

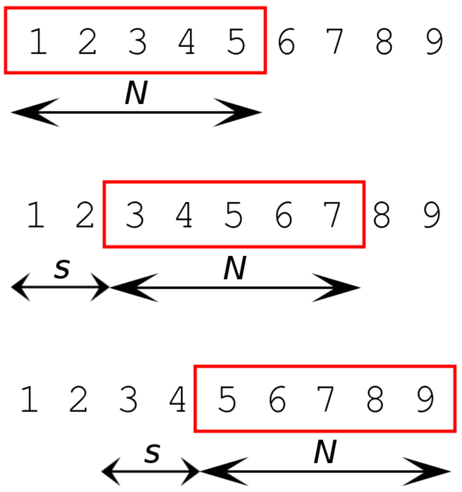


Figure 1.2: Example of a sliding window with $N = 5$ and $s = 2$.

simple threshold-based classifier: if the value of the feature is greater than a given threshold, the sequence is labeled as a fall; otherwise it is labeled as a ADL. The threshold values are chosen in order to maximize either accuracy or sensitivity.
Sensitivity (SE), specificity (SP) and accuracy (AC) are defined as follows:

$$SE = \frac{TP}{TP + FN} \qquad SP = \frac{TN}{TN + FP} \qquad AC = \frac{SE + SP}{2} \tag{1.1}$$

Where TP, TN, FP and FN are, respectively: true positive, true negatives, false positives and false negatives.

- **Validation**.
  The robustness of the classification stage was analyzed with a 10-fold cross-validation. The groups were chosen guaranteeing the same proportion of falls and ADL, and each group was used in one fold as validation data.

1.2.5 Results

Notation
A single sample of acceleration at time t is defined as

$$a[t] := [a_x[t], a_y[t], a_z[t]] \tag{1.2}$$

where $a_x[t]$, $a_y[t]$ and $a_z[t]$ are the individual acceleration values in the three axis.

The starting time of the k-th sliding window of signal is:

$$t_0^{(k)} := k \cdot s \tag{1.3}$$

where s is the stride. The k-th sliding window containing N samples is denoted as:

$$a_x^{(k)} := [a_x[t_0^{(k)}], \ldots, a_x[t_0^{(k)} + N - 1]] \tag{1.4}$$

Among the 14 features, the one that performed best overall was the so-called C_8, that resulted in both an accuracy and a sensitivity of about 90%. C_8, the "Standard deviation magnitude on horizontal plane", is defined as:

$$C_8^{(k)} := \sqrt{\sigma^2(a_x^{(k)}) + \sigma^2(a_z^{(k)})} \tag{1.5}$$

This feature got good results, because (as seen in Figure 1.1) the SisFall device was fixed. For our purposes, where a smartphone cannot be assumed to be fixed, the C_9 feature is a better candidate.
C_9 is described as "Standard deviation magnitude", and is defined by the following equation:

$$C_9^{(k)} := \sqrt{\sigma^2(a_x^{(k)}) + \sigma^2(a_y^{(k)}) + \sigma^2(a_z^{(k)})} \tag{1.6}$$

Where $\sigma^2(a_x^{(k)})$ is the variance of x over the k-th window of the signal.

1.3 Machine Learning

1.3.1 Neural Networks

Artificial neural networks are computing systems that synthesize pattern-detecting procedures by abstracting from data. For example, in image recog-

nition, such networks can learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the analytic results to identify cats in other images.
"Learning" here means that inputs and expected outputs are considered in pairs, and the parameters of the network are progressively adjusted to reduce the difference between actual and expected output according to some predefined loss function.

Machine Learning can be seen as composed of three parts (adapted from Domingos (2012)):

- **Representation**.
  How to effectively represent a target function.
- **Evaluation**.
  How one model is preferred over another: an evaluation function is needed to score different representations.
- **Optimization**.
  How to search the space containing the represented functions: a method is needed to find the highest-scoring one.

Representation

As proved by Cybenko (1989) and Csáji (2001), artificial neural networks are universal approximator of functions: a network of depth at least 2 can approximate any real function, to an arbitrary degree.

Considering a target function

$$y = f^*(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^d \tag{1.7}$$

can be approximated by a feed-forward neural network as

$$\tilde{y} = \boldsymbol{w} \cdot \boldsymbol{h} + b, \quad \boldsymbol{w} \in \mathbb{R}^h, \quad b \in \mathbb{R} \tag{1.8}$$

where $\boldsymbol{h}$, called "hidden layer" is

$$\boldsymbol{h} := g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \quad \boldsymbol{W} \in \mathbb{R}^{h \times d}, \quad \boldsymbol{b} \in \mathbb{R}^h \tag{1.9}$$

and $g(\cdot)$ is a non-linear function. Popular choices are:

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.10}$$

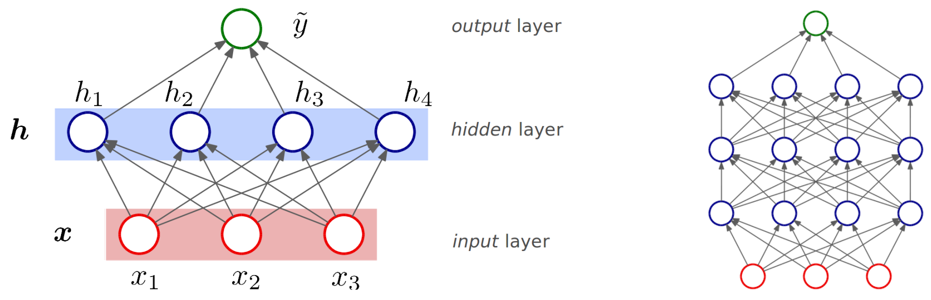
$$g(x) = \tanh(x) \tag{1.11}$$

$$g(x) = ReLU(x) = \max(0, x) \tag{1.12}$$

A “deep” neural network is an artificial neural network with multiple hidden layers (Figure 1.3b). Deeper networks need more parameters. For example, combining Equations 1.8 and 1.9 with a network with three hidden layers, we get

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} g(\boldsymbol{W}^{(3)} \boldsymbol{x} + \boldsymbol{b}^{(3)}) + \boldsymbol{b}^{(2)}) + \boldsymbol{b}^{(1)}) + b \tag{1.13}$$

Deep neural network are provably better than shallow ones. Quoting Goodfellow et al. (2016), *“[. . .] researchers were now able to train deeper neural networks than had been possible before, and to focus attention on the theoretical importance of depth. At this time, deep neural networks outperformed competing AI systems based on other machine learning technologies as well as hand-designed functionality”.*



(a) (b)

Figure 1.3: Representation of two Artificial Neural Networks, with (a) one and (b) three hidden layers.

Evaluation

How can we evaluate how well an artificial neural network approximates a function? That is, how good are the parameters of the network?
Formally, given a target function (Equation 1.7) and a dataset of samples S

$$S := (y^{(i)}, \boldsymbol{x}^{(i)}), \qquad y^{(i)} = f^*(\boldsymbol{x}^{(i)}) \tag{1.14}$$

we want to find the parameters $\boldsymbol{w}$, b, $\boldsymbol{W}$ and $\boldsymbol{b}$ in Equations 1.8 and 1.9 such that the loss $L(S)$

$$L(S) := \frac{1}{|S|} \sum\nolimits_{(y^{(i)}, \boldsymbol{x}^{(i)}) \in S} \left| \tilde{y}^{(i)} - y^{(i)} \right|^2 \tag{1.15}$$

is minimized.
Smaller values of $L(S)$ lead to better approximation, because the represented function is closer to the target function.

Optimization

How can a represented function be learnt from data? How to automatically choose the parameters that compose the best model?
In general, minimizing the loss function cannot be done directly. That is:

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(S) = 0 \tag{1.16}$$

(where $\boldsymbol{\vartheta}$ is any of the four parameters) cannot be solved analytically and has to be solved numerically.
For example, the Gradient Descent algorithm and its variations, widely used in machine learning, can find by iteration the local optimum.

Algorithm: Stochastic Gradient Descent

1. Assign initial values to the four parameters $\boldsymbol{w}$, b, $\boldsymbol{W}$ and $\boldsymbol{b}$.
2. Update the four parameters (with $\eta \ll 1$, $\eta \to 0$ as iteration progress):

   $$\Delta \boldsymbol{w} = -\eta \tfrac{\partial}{\partial \boldsymbol{w}} L(\tilde{y}^{(i)} - y^{(i)}) \qquad \Delta b = -\eta \tfrac{\partial}{\partial b} L(\tilde{y}^{(i)} - y^{(i)})$$

   $$\Delta \boldsymbol{W} = -\eta \tfrac{\partial}{\partial \boldsymbol{W}} L(\tilde{y}^{(i)} - y^{(i)}) \qquad \Delta \boldsymbol{b} = -\eta \tfrac{\partial}{\partial \boldsymbol{b}} L(\tilde{y}^{(i)} - y^{(i)})$$

   Where $L(\tilde{y}^{(i)} - y^{(i)}) := (\tilde{y}^{(i)} - y^{(i)})^2$ is the pointwise loss for a specific sample in the dataset.
3. Unless complete, return to step 2.

In summary, given a dataset of examples, an artificial neural network is capable of learning the parameters that best approximate the representation of any function.

1.3.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are networks with loops in them, allowing information to persist. The basic idea is to make the output depend on the

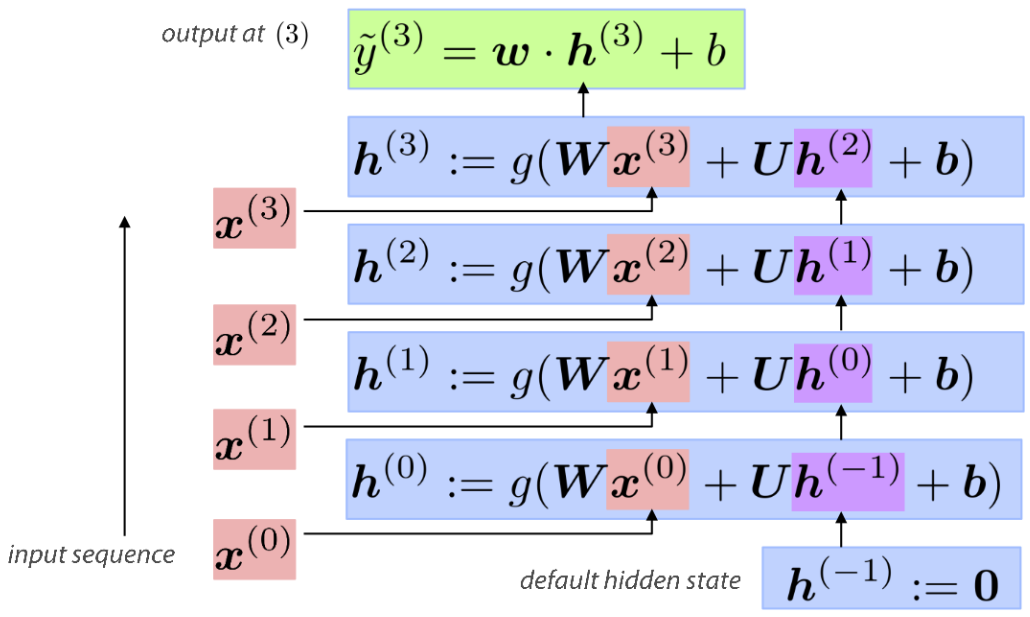
past history. They are mainly used to treat data in form of sequences and lists, especially signals that very over time.
The expression of a recurrent neural network is:

$$\tilde{y}^{(t)} = \boldsymbol{w} \cdot \boldsymbol{h}^{(t)} + b \tag{1.17}$$

Where $\boldsymbol{h}^{(t)}$, the hidden layer at time t, is

$$\boldsymbol{h}^{(t)} = g(\boldsymbol{W}\boldsymbol{x}^{(t)} + \boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}) \tag{1.18}$$

In figure 1.4 the single mathematical expressions can be seen.



output at (3)
$\tilde{y}^{(3)} = \boldsymbol{w} \cdot \boldsymbol{h}^{(3)} + b$
$\boldsymbol{h}^{(3)} := g(\boldsymbol{W}\boldsymbol{x}^{(3)} + \boldsymbol{U}\boldsymbol{h}^{(2)} + \boldsymbol{b})$
$\boldsymbol{x}^{(3)}$
$\boldsymbol{h}^{(2)} := g(\boldsymbol{W}\boldsymbol{x}^{(2)} + \boldsymbol{U}\boldsymbol{h}^{(1)} + \boldsymbol{b})$
$\boldsymbol{x}^{(2)}$
$\boldsymbol{h}^{(1)} := g(\boldsymbol{W}\boldsymbol{x}^{(1)} + \boldsymbol{U}\boldsymbol{h}^{(0)} + \boldsymbol{b})$
$\boldsymbol{x}^{(1)}$
$\boldsymbol{h}^{(0)} := g(\boldsymbol{W}\boldsymbol{x}^{(0)} + \boldsymbol{U}\boldsymbol{h}^{(-1)} + \boldsymbol{b})$
input sequence
$\boldsymbol{x}^{(0)}$
default hidden state
$\boldsymbol{h}^{(-1)} := \boldsymbol{0}$

Figure 1.4: Mathematical form of an unrolled RNN.

Recurrent neural networks are more powerful than feed-forward neural networks: they can approximate any Turing machine (Siegelmann and Sontag (1995)). They are altough much harder to train, but using temporal unrolling, gradient descent methods can be applied.

1.4 LSTM and composite models

1.4.1 The Problem of Long-Term Dependencies

When the gap between some relevant information and the place where it is needed is small, recurrent neural networks can learn very well to use that

past information.
To train a recurrent neural network, temporal unrolling is used (Figure 1.5). But temporal unrolling is limited, so also the time depth is. This means that when the gap between the relevant information and the point where it is needed is very large (Figure 1.6), the network cannot access that past information. In theory, recurrent neural networks are capable of handling these "long-term dependencies", but in practice they do not seem to be able to learn them.[2]

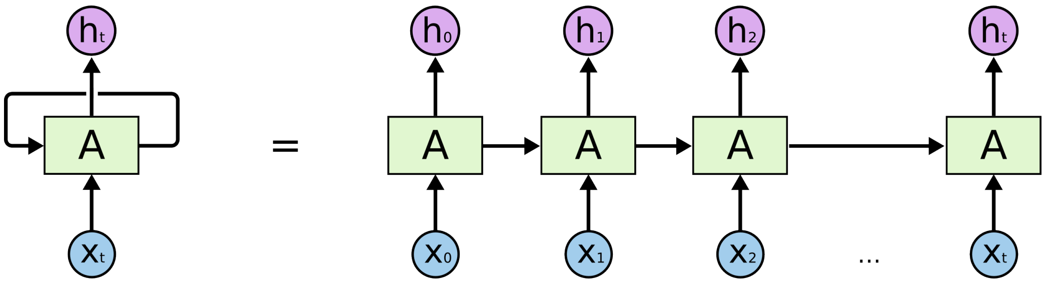


Figure 1.5: An unrolled RNN.

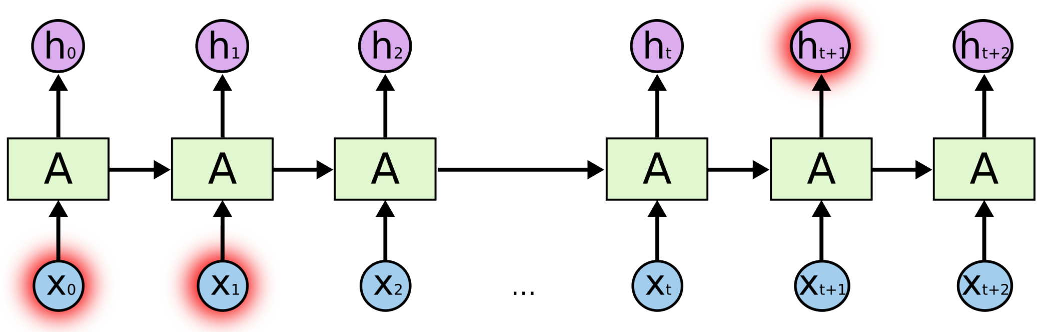


Figure 1.6: Long-term dependencies in a RNN.

The problem was explored in depth by Bengio et al. (1994), who found some fundamental reasons why it might be difficult: either the system will be very sensitive to noise, or the derivatives of the loss function will converge exponentially to zero (called the "vanishing gradient" problem).

[2]All the images in this chapter are taken from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

1.4.2 LSTM Networks

Long Short-Term Memory networks ("LSTM") are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997) and improved by Gers et al. (1999). They work very well on a large variety of problems, and are now widely used.
LSTM also have the chain-like structure of the RNN (Figure 1.7), but the repeating module has a different structure. Instead of having a single function $g(\cdot)$, they have four, interacting in different ways (Figure 1.8).

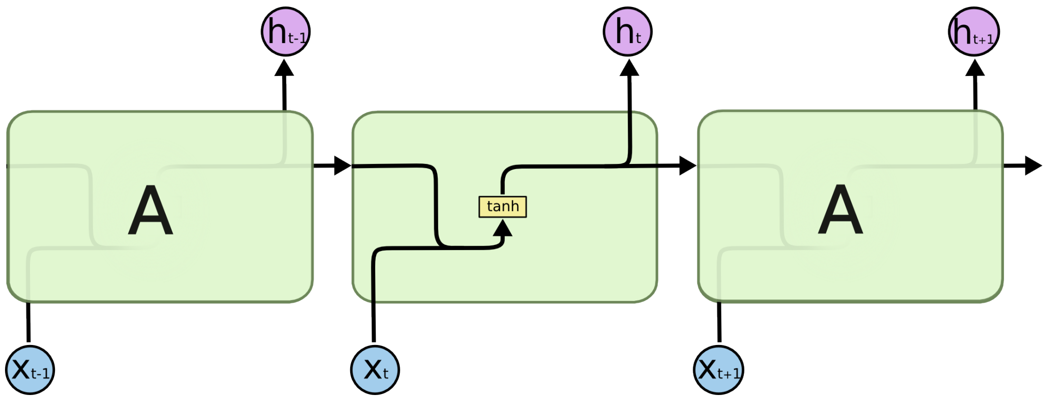


Figure 1.7: The repeating module in a standard RNN.

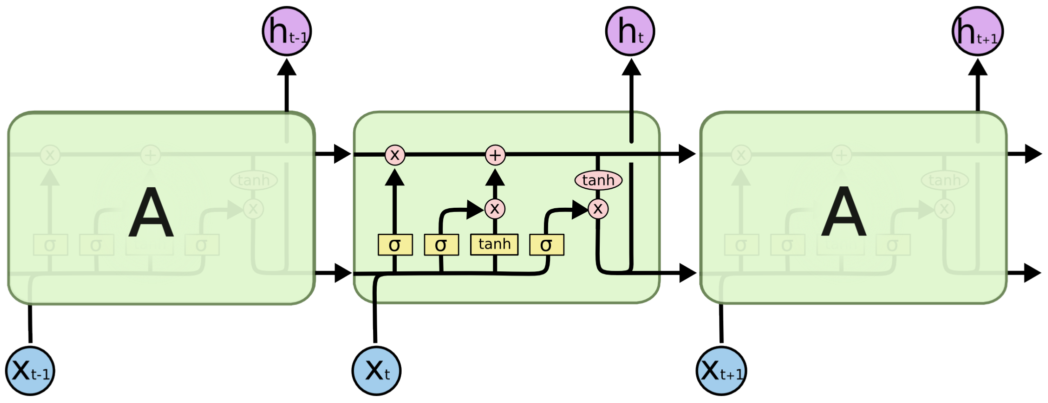
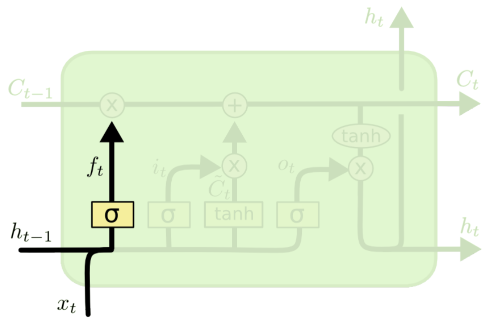


Figure 1.8: The repeating module in a LSTM.

From a step to the next one, the LSTM passes both the output and the state. The cell state runs straight down the entire chain, with some linear interactions: the LSTM has the ability to remove or add information to the cell state, by structures called gates. Gates optionally let information going

through.
An LSTM has three of these gates: an *input gate*, an *output gate* and a *forget gate*.

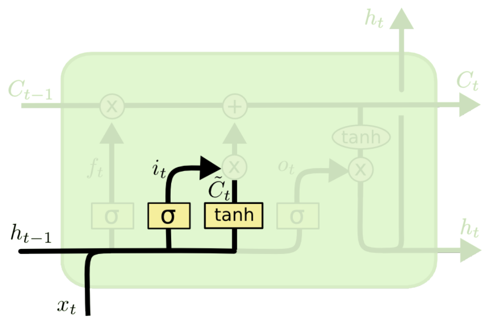
The first step the LSTM performs is to decide what information is going to be discarded from the cell state. This decision is made by the *forget gate* (Figure 1.9).



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 1.9: The forget gate of a LSTM.

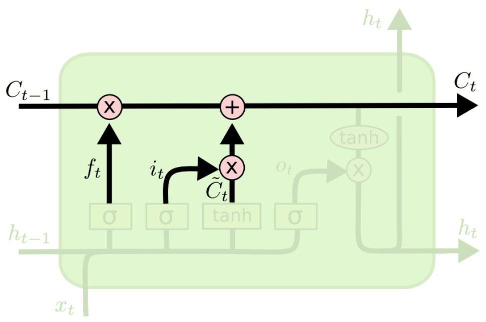
The next step is to decide what new information is going to be stored in the cell state. The *input gate* (Figure 1.10) decides which values are going to be updated, and a tanh layer creates a new vector of values that could be added to the state, $\tilde{C}_t$. The old cell state C_{t-1} is now going to be updated into the new cell state C_t (Figure 1.11): it is first multiplied by f_t, the output of the *forget gate*, and then added to $i_t * \tilde{C}_t$, the new (scaled) values.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

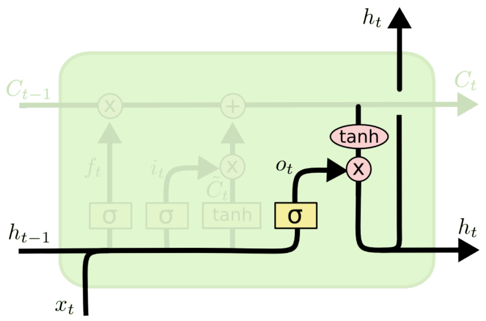
Figure 1.10: The input gate of a LSTM.

The last step is to produce the output (Figure 1.12). It is calculated by multiplying the output of the *output gate* and the current cell state.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 1.11: The update of the cell state of a LSTM.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 1.12: The output gate of a LSTM.

Chapter 2

Practice

2.1 Android Devices

Since late 2016, mobile operating systems that can be found on smartphones include Google's dominant Android and Apple's iOS as the only two big competitors, with both combined at about 99% market share. According to these statistics[1], devices running the Android operating system cover about 88% of the market. Such extensive distribution makes Android an ideal target for the development of a fall detection application.

In general, a fall detection application must:

- Read the signals from the built-in accelerometer.
- Interpret these signals:
  - Elaborate the signals according to an algorithm.
  - Based on the algorithm output, classify the read signals (and thus the event that just took place) as ADL or as fall.

Almost every smartphone running Android possess (at least) one built-in accelerometer. Reading the acceleration signals from it is widely supported by the Android API, albeit with its limitations (described later on in this chapter).

The interpretation of accelerometer signals, however, is rather difficult:

- Input signals are time-variant (unlike a still image)

[1] `https://www.gartner.com/newsroom/id/3516317`

- Information is limited (just three scalar values at each time instant)
- It is the history of the signals (the waveform) that describes the event, not the instant value

Besides, false negatives are to be avoided, but even false positives cannot be too high in measure.
In fact, automatic fall detection from sensor data is still an open problem. Some approaches have been developed; for example, the SisFall team (Sucerquia et al. (2017)) developed a threshold-based classifier, while our work will employ machine learning techniques (described in the next chapter).

2.1.1 Reading the Accelerometer signals

In Android, each reading of the accelerometer values comes in the form of an event.
Such event holds the following informations:

- Sensor type.
- Timestamp.
- Accuracy.
- Sensor reading values.

In our case the sensor type is always an accelerometer. Other sensor types might be available on the device, such as a gyroscope.
The timestamp value is expressed in nanoseconds, and represents the time passed since the activation of the sensor.
The accuracy is a parameter that describes the sampling rate. On Android the accelerometer does not have a fixed sampling rate, and the accuracy acts like a "suggestion", but ultimately the sampling rate is out of our control. This problem will be addressed in the following chapter.
The sensor reading values are stored in an array named `values`. Each value is a `float` representing the individual acceleration values in the three axis, expressed in m/s^2.

In the light of what just described, every event holds the same informations defined in Equation 1.2, in Chapter 1:

$$a[t] := [a_x[t], a_y[t], a_z[t]] \tag{1.2}$$

To receive notification of sensor reading events from Android, an application must:

- Define a class that implements `SensorEventListener`.
- Override in such class the method `onSensorChanged`.
- At run time, register such class as a listener for the accelerometer.

A sample code follows:

```
public class Activity [...] implements SensorEventListener {
    [...]
    public void onSensorChanged(SensorEvent event) {
        timestamp = event.timestamp;
        x = event.values[0];
        y = event.values[1];
        z = event.values[2];
    }
    [...]
    SensorManager mSensorManager = (SensorManager)
        ↪ getSystemService(Context.SENSOR_SERVICE);
    Sensor mAccelerometer = mSensorManager.getDefaultSensor(
        ↪ Sensor.TYPE_ACCELEROMETER);
    mSensorManager.registerListener(this, mAccelerometer,
        ↪ SensorManager.SENSOR_DELAY_FASTEST);
    [...]
}
```

`event.accuracy` values are not read here, as we chose to use the fastest available sampling rate.

2.2 TensorFlow

Developed by researchers at Google, TensorFlow (Abadi et al. (2015)) is an open source software framework for the purposes of machine learning: it is intended for tensorial numerical computation and is optimized for building and training neural networks.
It reached version 1.0 in February 2017[2], and is now used by a wide range of different companies. It has become one of the main framework for machine learning, thanks to the support of Google and the richness of functions implemented.

[2]`https://research.googleblog.com/2017/02/announcing-tensorflow-10.html`

TensorFlow performs three primary functions:

- **Tensorial Computation**.
  The central data structure in TensorFlow is the *tensor*, that is a set of primitive values shaped into a multidimensional array.
  A *computational graph* is a series of TensorFlow operations arranged into a graph of nodes. Each node has zero or more inputs and zero or more outputs, and represents a mathematical operation. Values that flow along normal edges in the graph (from outputs to inputs) represent the tensors (Figure 2.1).
  TensorFlow both builds and runs the computational graph. Building a graph means defining all its nodes and edges, while running means performing the actual computations.

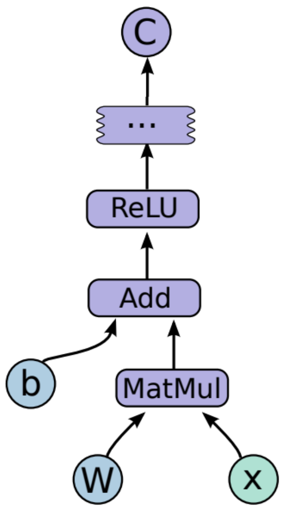


Figure 2.1: An example of a TensorFlow computational graph. The represented graph roughly corresponds to equation 1.9 in Chapter 1.

- **Automatic Gradient Computation**.
  Most numerical optimization algorithms require computing the gradient of a loss function with respect to a set of inputs (e.g. the Stochastic Gradient Descent seen in Chapter 1). Because this is such a common need in machine learning, TensorFlow has a built-in support for automatic gradient computation. Gradients are computed, like other tensors, by extending the TensorFlow graph (Figure 2.2).

- **Ability to run with hardware acceleration**.
  Learning algorithms can be computationally intensive to execute. For

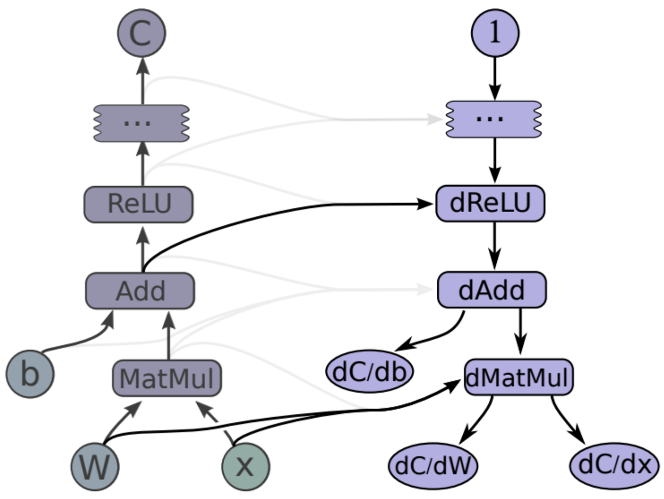


Figure 2.2: Gradients computed for graph in Figure 2.1.

this reason, TensorFlow provides an interface to leverage hardware acceleration (e.g. GPUs), if available on the device, to improve performance.

A machine learning technique will typically employ TensorFlow to build the graph describing the network architecture, and then running this graph for optimizing the parameters (i.e. training the network) thanks to the automatic gradient computation.
Once the parameters are learned, running the graph with a certain input will be computationally easy, and the output will represent the output of the network.

TensorFlow can run on a wide variety of different hardware platforms. We can here identify two of these platforms, and highlight what are the TensorFlow capabilities in each of them:

- **Workstation**.
  When running on a workstation, TensorFlow can perform all its functions. Specifically, it can both *train* the network and make *inferences* with it (i.e. give an input to the trained network and output a prediction). While training is a computationally intensive process, inference is not.

- **Smartphone**.
  When running on a smartphone, where the computing power is limited,

TensorFlow can only make inferences. At the moment, TensorFlow has two versions that can run on a smartphone: TensorFlow Mobile and TensorFlow Lite.

2.3 TensorFlow for Android

2.3.1 TensorFlow Mobile

Since May 17th, 2017 Google started the the "TensorFlow Mobile" project[3], aimed at an easy way to incorporate TensorFlow on mobile devices.

The "core" of TensorFlow Mobile is the TensorFlow Inference Interface, the module capable of making inferences. This module is available as a JCenter package, guaranteeing a stable and constantly updated release.

The Inference Interface needs a trained neural network as input. This trained network is also called a "frozen model" in the documentation, and contains all the relevant information (the graph, defining the structure of the network, and every tensor, containing network parameters). Irrelevant informations, that are not needed by the Inference Interface, are discarded in an automatic process called "pruning", to reduce the file size of the model.

The *frozen model* can then be used as asset inside the application.
To access this model and subsequently perform inference, the application must know the following:

- Path of the frozen model
- Name of the input and output tensors
- Shape of above tensors

In this snippet of code the necessary informations are highlighted in red:

```
import android.content.Context;
import android.content.res.AssetManager;
import org.tensorflow.contrib.android.
   ↪ TensorFlowInferenceInterface;

class TF_classifier {
```

[3]https://www.tensorflow.org/mobile/mobile_intro

```
   private static final String MODEL_FILE =
      ↪ "file:///android_asset/frozen_model.pb";
   private static final String INPUT_NODE = "Input";
   private static final String[] OUTPUT_NODES = {"Output"};
   private static final String OUTPUT_NODE = "Output";
   private static final long[] INPUT_SIZE = 1, 128, 3;
   private static final int OUTPUT_SIZE = 2;

   private TensorFlowInferenceInterface inferenceInterface;

   TF_classifier(final Context context) {
      AssetManager assetManager = context.getAssets();
      inferenceInterface = new TensorFlowInferenceInterface(
         ↪ assetManager, MODEL_FILE);
   }

}
```

Note that this class, `TF_classifier`, creates an instance of `TensorFlowInferenceInterface`. This interface has three main methods:

- `feed`, used to give the input to the trained network.
- `run`, to actually run the computational graph.
- `fetch`, to obtain the values of the output nodes.

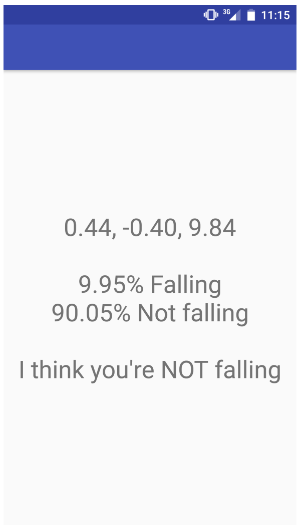
A simple method implementing this, `predictProbabilities`, is shown here:

```
float[] predictProbabilities(float[] inputData) {

   float[] results = new float[OUTPUT_SIZE];
   inferenceInterface.feed(INPUT_NODE, inputData, INPUT_SIZE);
   inferenceInterface.run(OUTPUT_NODES);
   inferenceInterface.fetch(OUTPUT_NODE, results);

   return result;
}
```

A screenshot from a running version of TensorFlow Mobile is shown in Figure 2.3.


11:15

0.44, -0.40, 9.84

9.95% Falling
90.05% Not falling

I think you're NOT falling

Figure 2.3: TensorFlow Mobile running on Android, inside a fall detection app. The output of the network is in form of complementary probabilities. The app and its functions will be described in Chapter 3.

2.3.2 TensorFlow Lite

Currently (starting from November 14th, 2017) Google is developing "TensorFlow Lite", an evolution of TensorFlow Mobile[4].
TensorFlow Lite will provide better performance and a smaller binary size on mobile platforms as well as the ability to leverage hardware acceleration if available on these platforms.
In addition, TensorFlow Lite has fewer dependencies so it can be built and hosted on simpler, more constrained device scenarios.

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

Chapter 3

Experiments

3.1 Objective

The objective of the following experiments is to develop an automatic fall detection technique (see Chapter 1), that will be implemented in an application for smartphones mainly aimed towards people with frailties.
This project is done in the scope of a collaboration between the University of Pavia and KeyPeople, srl. The role of the University of Pavia is to provide fall detection techniques, while KeyPeople will be in charge of the user experience and the distribution of the app.

We chose the Android operating system for ease of development (as seen in Chapter 2).
The app should be active 24/7, and it must perform fall detection and alert generation in soft real-time (i.e. within 1-2 seconds).
When active and carried by the user, the typical workflow of the app (Figure 3.1) is the following:

1. Detection of a (possible) fall
2. Generation of a local alert (e.g. an acoustic signal)
3. Wait for a possible manual reset (in the case of a false positive)
4. Generation of a remote alert (an automatic phone call, or the start of a safety procedure where a remote operator will take care of the situation. Other options are possible: for example, the activation of the smartphone microphones or camera)

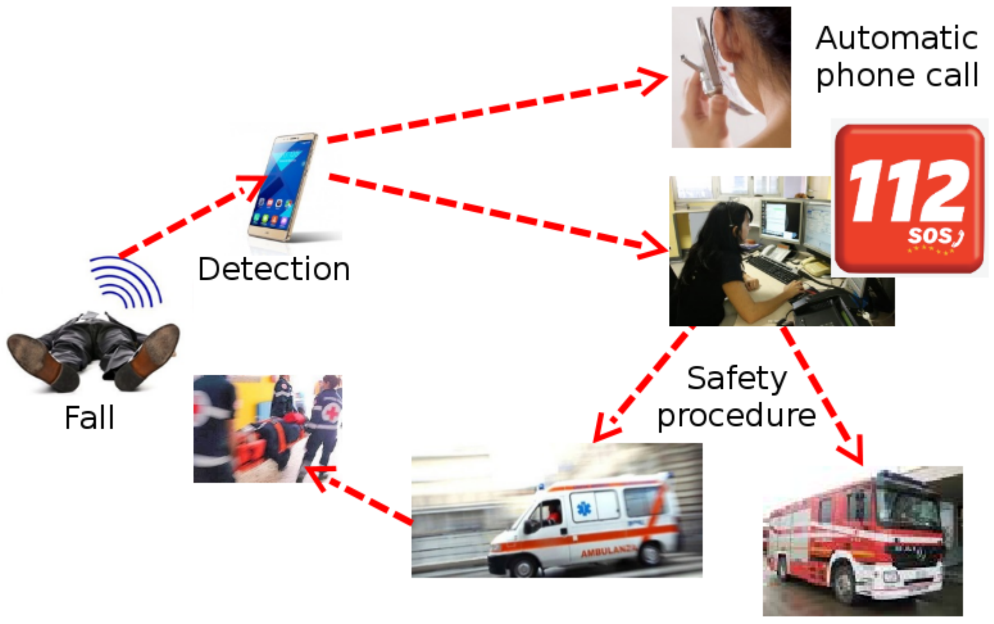


Figure 3.1: The typical workflow of the app (courtesy of KeyPeople, srl).

3.2 Fall Detection with Statistical Indicators

The SisFall dataset (Sucerquia et al. (2017)) is a viable candidate for developing and validating automatic fall detection techniques since it contains a great number of different activities, performed by different subjects of various ages, with accelerometers and gyroscopes signals recorded at a high frequency (200 Hz).

More precisely, as already discussed in Chapter 1, we can characterize the dataset as follows:

- It contains 19 Activities of Daily Living (ADL) and 15 falls, performed by 23 young adults and 14 elders.
- Each subject, after being instructed on how to perform a specific activity, repeatedly performed such activity from 1 to 5 times.
- Signals recorded for each of these repetitions were stored in a specific file.
- Each file (hence each activity) was labeled with the type of activity performed (i.e. the “code” in Tables 1.1 and 1.2 in Chapter 1)

As it can be seen in Figure 3.2, SisFall activities are in fact composite: for example a single activity may contain walking, stumbling, falling and immobility in a sequence.

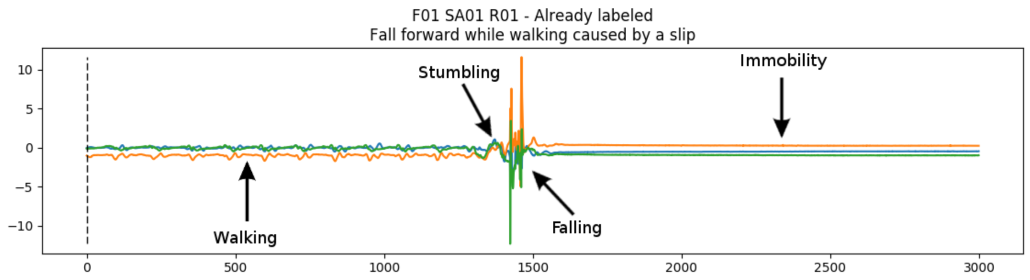


Figure 3.2: Example of a composite SisFall activity: the fall occurs after stumbling while walking. This screenshot is taken from the labeler program described later on.

In the original SisFall paper, the authors proposed a set of statistical indicators (also called "features") to classify each recorded sequence of signals as being either ADL or fall. In particular, as explained in Chapter 1, each of these features acts on a sliding window of signals and produces a scalar value that is compared with a predefined threshold.
As seen in Chapter 1, the best results were obtained using feature C_8. However, such feature relies on the assumption of a fixed orientation of the device hence of the accelerometers.

For this reason, we focused on feature C_9, which does not rely on the fixed orientation assumption.

$$C_9^{(k)} := \sqrt{\sigma^2(a_x^{(k)}) + \sigma^2(a_y^{(k)}) + \sigma^2(a_z^{(k)})} \tag{1.6}$$

We implemented a python procedure for computing the C_9 feature, and we tested it against the entire SisFall dataset, replicating substantially the same results (i.e. 92% accuracy) reported in the original paper. Also in agreement with the original paper, we observed that feature C_9 is slightly less accurate than feature C_8.

In a subsequent step, we decided to test feature C_9 with an app for Android smartphones. In the application developed (see Figure 3.3):

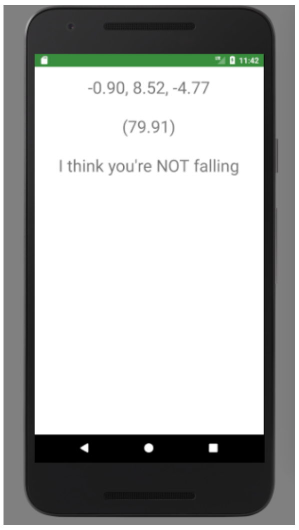
1. Accelerometer signals are obtained in real-time from the built-in sensors of the smartphone (as seen in Chapter 2).

2. Signals are cached in a sliding window ($N = 128$ and $s = 1$, see Chapter 1).

3. Feature C_9 is computed for each sliding window, hence at each time instant with a delay of 128 ticks.

4. The value of feature C_9 is compared with a predefined threshold and an event (i.e. an acoustic signal) is generated whenever such value is above threshold.

In this application we used the threshold value (187.14) that produced the best performances for feature C_9 over the whole SisFall dataset in our python implementation.


-0.90, 8.52, -4.77

(79.91)

I think you're NOT falling

Figure 3.3: Using feature C_9 in real-time within the Android application. The figures in the first row represent the accelerations on the three axis, while the value in the second row is that of feature C_9 computed over the previous 128 samples. The third row indicates whether this value is above or below the predefined threshold.

We did not perform a systematic test with this application since even simple episodic experiments showed that:

- The feature C_9 detection technique worked very differently on different devices, even with the same threshold (e.g. approximately the same action was easily above threshold on some devices and hardly so on others).

- In any case, feature C_9 proved to be sensitive not just to actual falls but simply to any sudden movement (i.e. with an abundance of false positives).

Another relevant observation made during these tests is that, as described in Chapter 2, the sampling frequency of an Android device is not constant by design (and reported in the official Android documentation). Such aspect could certainly influence the performances of feature C_9 classifications as well as with any other techniques.

Due to the above reasons we decided to adopt machine learning techniques for the next steps.

3.3 Machine Learning on SisFall

As the machine learning technique of choice, we considered that of LSTM networks (see Chapter 1). To do so, we adapted and further developed an LSTM implementation based on python and TensorFlow (Chevalier (2016)).

Unlike feature C_9 the training activity of an LSTM is supervised hence it requires an annotated dataset. More precisely, during the training phase, an LSTM is unrolled[1], so that its activation over an entire sequence of input signals contained in a sliding window is transformed into the activation of a much deeper feed-forward network (see Chapter 1).

However, this means that such supervised training requires a specific class label (i.e. ADL vs fall) for each sliding window. Dividing each sequence of signal in the SisFall dataset into sliding windows is not difficult but the same dataset does not contain a pointwise temporal annotation that could be directly applied to each such sliding window. Lacking a better alternative, which will be described below, we used the C_9 feature to generate labels (See Figure 3.4). Specifically, we labeled the values either below or above threshold as either ADL or fall, respectively, and used those labels for the supervised learning algorithm. Clearly, by doing this, we could only make the LSTM learn to reproduce the C_9 statistical indicator.

For precision, in these experiments, we used a two-layered LSTM architecture described in Chevalier (2016) (Figure). Such architecture is composed as follows:

1. A feed-forward network with one hidden layer.

2. Two stacked LSTM cells.

[1]This activity is performed automatically in TensorFlow.

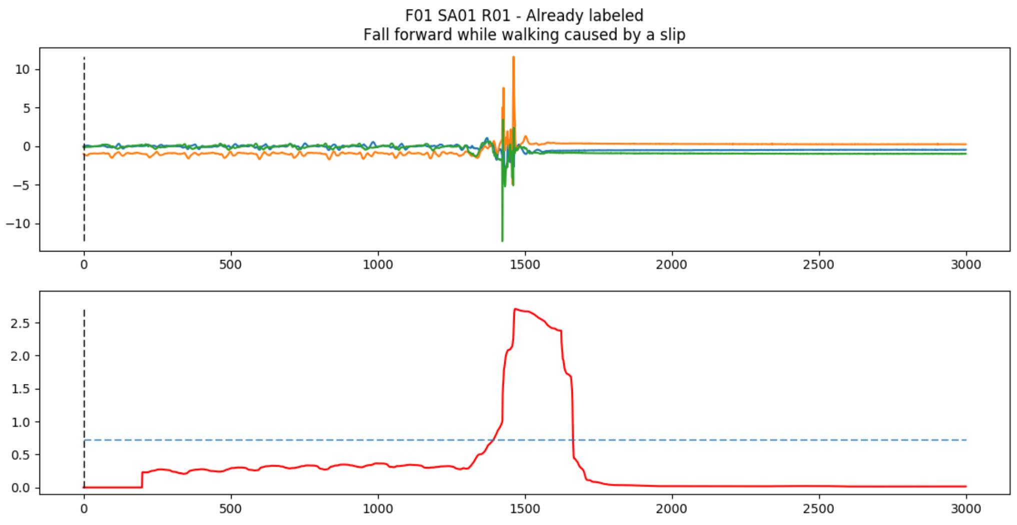


Figure 3.4: Above: signals from the accelerometers. Below: the values computed with feature C_9: values above threshold (dashed line) are labeled as fall. This screenshot is taken from the labeler program described later on.

3. A feed-forward network with one hidden layer.

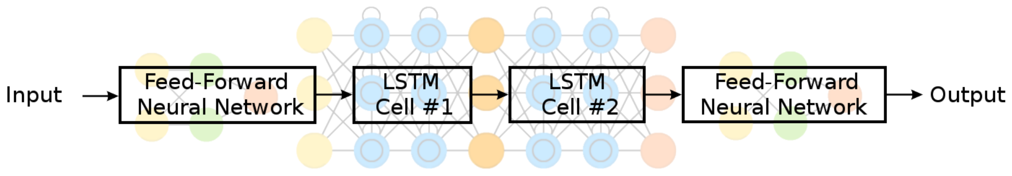


Figure 3.5: The architecture of the network. A feed-forward network, followed by two stacked LSTM cells, followed by another feed-forward network.

The rationale for choosing this architecture is because the author applied it successfully to Human Activity Recognition (HAR) on a completely different dataset (Anguita et al. (2013)). HAR is a different type of task in which the target is classifying human activities like, for instance, walking, going upstairs, going downstairs, sitting, standing and laying horizontally.

Such architecture was trained on the SisFall dataset with the addition of the automatically-produced labeling obtained from feature C_9 as described above. Initial experiments produced an overall accuracy of about 91% which, considering that the LSTM-based architecture was trying to learn a statistical indicator, is relatively low.

In a second experiment, we adopted a *data augmentation* technique by which we extended the SisFall dataset adding for each sequence the three rotations of 90 degrees over the Z axis. Data augmentation techniques like this one are typically used in machine learning to expand and enhance the dataset. In this case, we wanted to overcome the limitation due to the fixed orientation of the acquiring device which is intrinsic to SisFall. After re-training the network architecture on such augmented dataset, we obtained a more satisfactory accuracy of about 98%.

3.4 Beyond SisFall

In the light of what above, the SisFall dataset has two intrinsic characteristics that make it (as it is) sub-optimal for our purposes:

- **Only entire sequences are labeled**.
  The final application should be active 24/7 and be able to detect falls in soft real-time (i.e. within 1-2 seconds). To train LSTM-based architectures the dataset must be annotated with fine-grained temporal labeling, ideally one label per each time tick.

- **Specialized hardware with a fixed orientation was used for acquisition**.
  Working on an Android smartphones is substantially different: first of all no fixed orientation of the device can be assumed and, as seen in the previous chapter, it is not even possible to assume a fixed sampling frequency.

These reasons made us plan two distinct actions that are described below.

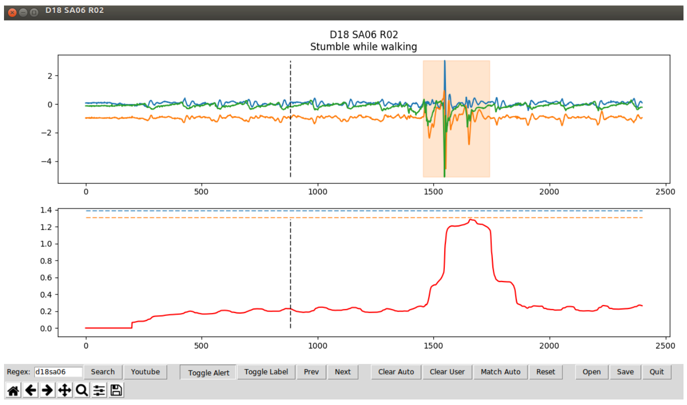
3.4.1 Fine-grained Temporal Labeling of SisFall

The only way to obtain a fine-grained labeling of the SisFall dataset is to manually annotate every sequence in it.

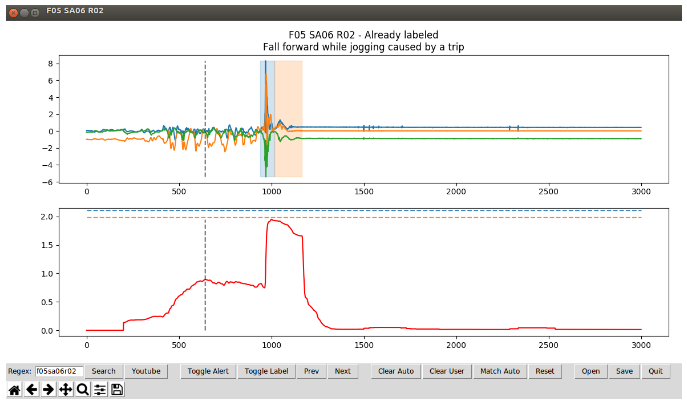
To do so, we developed a specific software tool for navigating through the original dataset and manually adding classifications associated to time intervals (See Figure 3.6).

Three categories are defined:

- **Warning** (in orange).
  Activities that are dangerous but still not representing a fall (for example: stumbling)



(a)



(b)

Figure 3.6: SisFall Data Labeler. Above: an example of the D19 activity (inconsequential stumbling); the annotated interval is in orange while the values of feature C_9 are reported in the lower panel as additional information to the human annotator. Below: an example of the F05 activity; the interval corresponding to the fall is marked in blue and its aftermath in orange.

- **Fall** (in blue).
  An effective fall, starting from the "point of no return" of losing equilibrium up to hitting the ground.

- **Aftermath** (in orange, immediately after an interval in blue).
  Activities (mostly involuntary) that follow the actual fall before final stabilization.

The typical workflow performed by the human annotator is the following:

1. A specific sequence is selected from the SisFall dataset and shown on screen.

2. Annotations of the three different kinds above are edited manually.
3. Annotations are saved in a separate file when the "save" button is pressed.

Doing as above allowed us, with the help of a few volunteers, to add a fine-grained temporal labeling to every sequence in the SisFall dataset. Note however that in this way the human annotators had to guess which was the condition of the human being performing each activity by just analyzing the sequence of accelerometer signals. Although with SisFall we had no alternatives, this is sub-optimal in that a more realistic annotation could be produced by referring to the actual action performed as it can be seen in a recorded video (not available in SisFall).

3.4.2 Acquiring an Integrated, Enhanced Dataset

To overcome the above shortcomings of SisFall, a new acquisition campaign has to be planned and the required set of tools must be realized.

In particular the new acquisitions should satisfy the following requirements:

- Accelerometers signals should be acquired with actual Android smartphones.
- Devices (i.e. smartphones) should be carried in different positions and orientations.
- The action performed by the human being should be recorded in a video and such video should be in sync with the acquisition of accelerometer signals.

The main objective of these requirements is having an heterogeneous dataset and make it possible to human annotators to work on activity videos disregarding accelerometers signals entirely.

3.5 Body Network of Android Devices

To satisfy the requirements described above, we decided to implement a software architecture comprising a "body network" composed of various Android devices carried on body, which is connected to a remote controller that governs the synchronized acquisition of signals and videos.

More precisely the architecture shown in Figure 3.7 has the following characteristics:

- Several acquiring devices can be governed simultaneously so that a single human composite activity will produce several different sequences of signals.
- The controller module will govern the actual acquisition performed by carried smartphones in order to ensure synchronism.
- The controller module, deployed on a tablet, will also be recording a video while smartphones will be recording accelerometer signals.

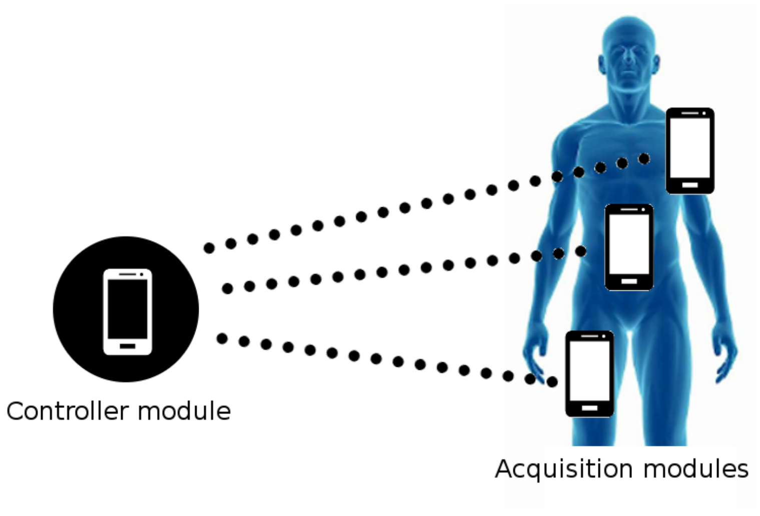


Figure 3.7: The overall architecture of the body network for the acquisition experiments. A controller module send commands to the acquisition modules, strapped on different parts of the body.

From a software standpoint, each acquisition module, when activated, opens a listening TCP/IP socket and executes commands received from the controller module via TCP protocol.
The recognized commands are:

- Connect to the controller module.
- Receive the username and the activity selected for performance from the controller module (it will be used to determine the name of the file).

- Receive the “start” signal to begin recording.
- Receive the “stop” signal to end recording.
- Save the recorded signal sequence in a file on the internal storage.
- Send the recorded signal sequence to the controller module.
- Close the connection.

At the beginning of each acquisition episode the human performer will be wearing the smartphones which will be properly activated. Then, the controller module will be started on the tablet and will connect to each acquisition module on smartphones.
From that point on, the entire recording activity is governed by the controller module in the sense that a human coordinator will be starting and stopping each acquisition using the controller module itself on the tablet. This means that during the acquisition of accelerometer signals the human coordinator will be pointing the video camera of the tablet towards the performer, so that the required video could be recorded. Eventually, the locally-acquired video and the remotely-obtained sequences of signals will be stored on the tablet.

For extra safety each sequence of signals acquired is also saved in the internal storage of each acquisition module. All files are named in a way that is unique to each episode and type of activity performed.

A few screenshots from the app are shown in Figure 3.8.

The actual software implementation of both the acquisition module and the controller module is by a unique app capable of fulfilling both roles, according to what the user selects at startup time.
Actually, the app also implements two further operating modes:

- **Standalone Mode** (Figure 3.9).
  This mode works on the smartphone alone, without remote TCP/IP connection and is intended for collecting data without a supervisor.
- **Web Server Mode** (Figure 3.10).
  In this mode the app starts a web server that also implements specific web pages with javascript provisions for the complete remote control of the application itself. In the intended usage, another user connects to the app via a web browser (e.g. on a notebook) and controls the

actions performed by the app. Each recorded sequence of signals is stored locally by the app and also sent over to the remote web browser.

The software implementation of the body network architecture will be completed by a software module to be executed on workstation for dataset post-processing. Such module will perform the following functions:

- Storage of the associated sets of video recordings and signal sequence files.
- Re-sampling of signal sequences with interpolation to obtain a constant sampling frequency.
- Manual video annotation by human experts, which will be performed by using a subtitle editor for simplicity.
- Production of additional files corresponding to each signal sequence containing the labels extracted from video annotations.

3.6 TensorFlow on Android

Apart from the collection of an enhanced and extended dataset, we also need to execute the detection procedures synthesized with machine learning techniques on the target Android devices. More precisely, we plan to perform the training of the LSTM-based architecture on a workstation and then apply such architecture after training on the Android device.

For doing so we adopted a recently released (previously under development) version of TensorFlow which is specifically conceived for Android (see Chapter 2).

The main aspect involved in such transfer on Android devices is the capability to produce “frozen” and “pruned” trained models from TensorFlow on workstation to TensorFlow Mobile. As described in Chapter 2, TensorFlow Mobile will not be used for training but just for making *inferences*, namely for detecting actual falls.

In this perspective, we can describe our complete architecture for automatic fall detection using machine learning techniques with the following steps:

1. Acquisition of a specific dataset, via the body network architecture.
2. Labeling of the recorded videos for each activity performed.

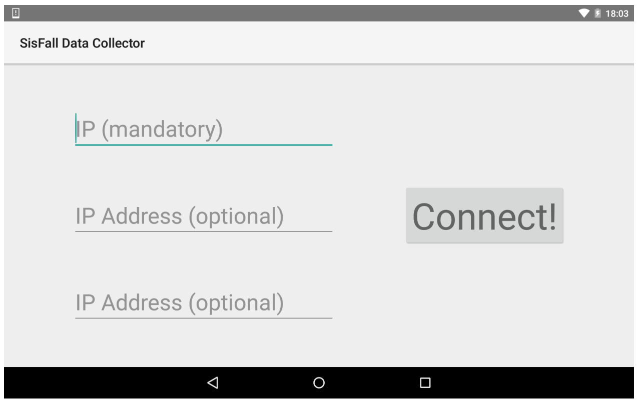
3. Transferring of video labeling to each corresponding sequence of accelerometer signals.

4. Training (on workstation) of the LSTM-based architecture using the full dataset of labeled sequences.

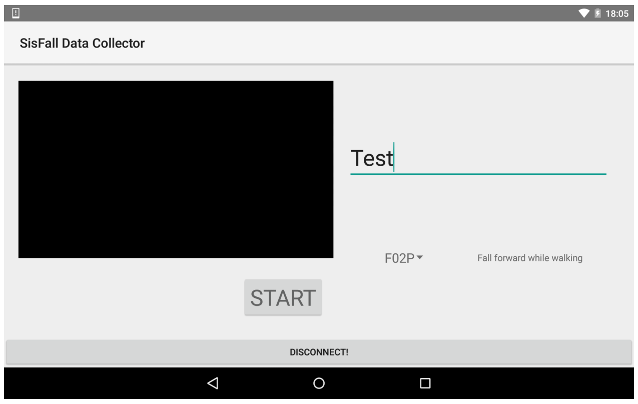
5. Freezing and pruning the trained model obtained.

6. Transferring this model on Android.

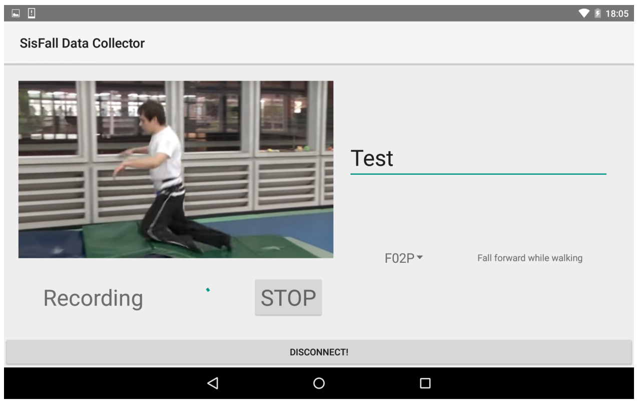
7. Perform systematic test of automatic fall detections directly with smartphones.



(a)

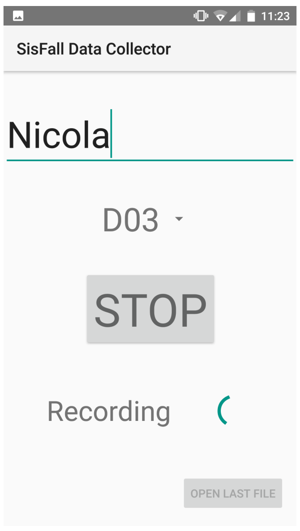


(b)

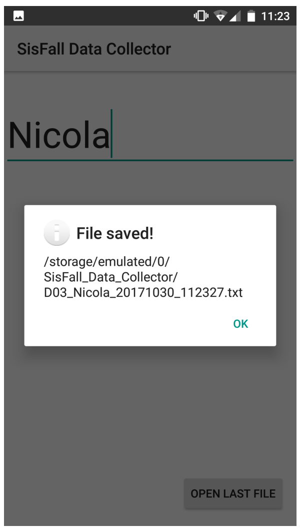


(c)

Figure 3.8: Body network software modules: (a) the controller module establishes the TCP/IP connections with up to three acquisition modules; (b) after connection, the controller module requires entering id data about the performer and the activity; (c) when the “start” button is pressed the controller module starts the acquisition of accelerometer signals by remote modules and starts recording the video.



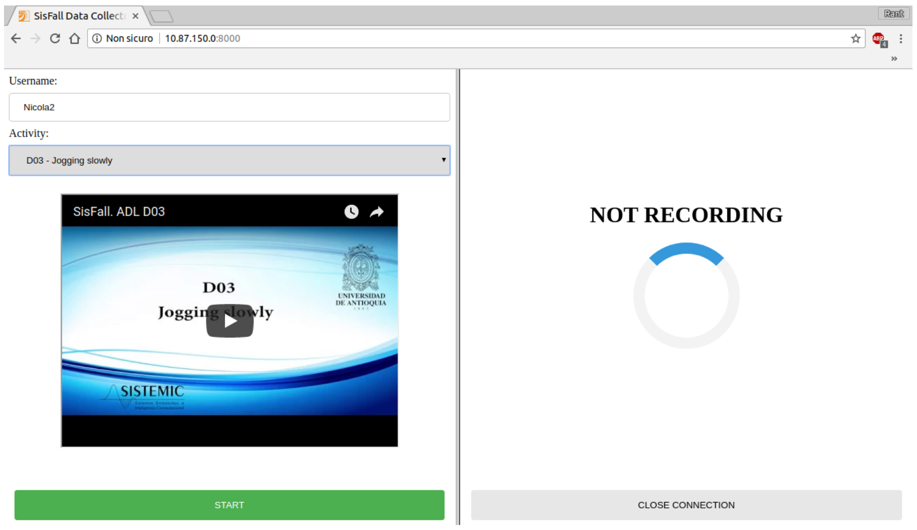
(a)



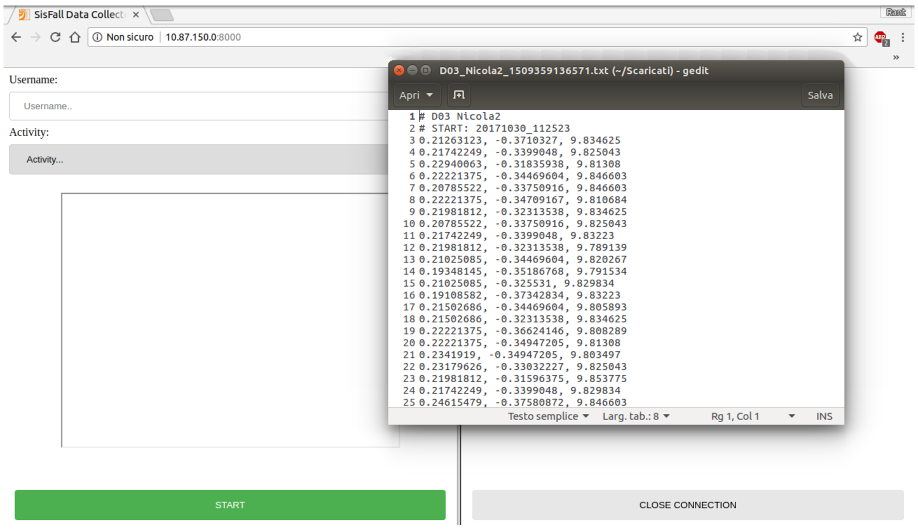
File saved!
/storage/emulated/0/
SisFall_Data_Collector/
D03_Nicola_20171030_112327.txt

(b)

Figure 3.9: Body network software modules: Standalone Mode. The user enters id data and starts/stops recording.



(a)



```
# D03 Nicola2
# START: 20171030_112523
0.21263123, -0.3710327, 9.834625
0.21742249, -0.3399048, 9.825043
0.22940063, -0.31835938, 9.81308
0.22221375, -0.34469604, 9.846603
0.20785522, -0.33750916, 9.846603
0.22221375, -0.34709167, 9.810684
0.21981812, -0.32313538, 9.834625
0.20785522, -0.33750916, 9.825043
0.21742249, -0.3399048, 9.83223
0.21981812, -0.32313538, 9.789139
0.21025085, -0.34469604, 9.820267
0.19348145, -0.35186768, 9.791534
0.21025085, -0.325531, 9.829834
0.19108582, -0.37342834, 9.83223
0.21502686, -0.34469604, 9.805893
0.21502686, -0.32313538, 9.834625
0.22221375, -0.36624146, 9.808289
0.22221375, -0.34947205, 9.81308
0.2341919, -0.34947205, 9.803497
0.23179626, -0.33032227, 9.825043
0.21981812, -0.31596375, 9.853775
0.21742249, -0.3399048, 9.829834
0.24615479, -0.37580872, 9.846603
```

(b)

Figure 3.10: Body network software modules: Web Server Mode: (a) the web interface to the application; (b) each recorded sequence is sent to the browser.

Chapter 4

Results and Future Developments

The general plan of this project is structured in three parts:

1. **Data collection**.
   We need to record a large amount of data using the body network architecture described in the previous chapter. These recordings will then be manually labeled in order to obtain a viable dataset.

2. **Training**.
   Our LSTM-based neural network architecture will be trained using the newly acquired dataset. A frozen model of the network will be saved.

3. **Testing on Mobile Devices**.
   The frozen model will be deployed on a smartphone, and used to run TensorFlow Mobile inside an application.

Due to time restrictions, actual activities were only limited to data collection and to some considerations about training.

4.1 Data Collection

Data collection is the first step of the process: to have a viable dataset, we need to collect the accelerometer signals and relative videos from a large number of people.
To collect data, we set up some appointments with volunteers in a safe environment (a large gym with mattresses). Volunteers will perform some chosen

Code	Activity
D01P	Walking
D02P	Jogging
D03P	Walking upstairs and downstairs
D04P	Sit in a chair, wait a moment and get up
D05P	While sit in a chair, try to get up and collapse in the chair
D06P	Standing, lay on a “bed” and get up on your feet
D07P	Pick an object from the ground
D08P	Stumble while walking but remain up still
D09P	Stumble while walking and use a table to avoid falling
D10P	Gently jump without falling
F01P	Fall forward while walking
F02P	Fall backward while walking
F03P	Fall laterally while walking
F04P	Vertical fall by fainting
F05P	Fall while walking, with use of hands in a table to dampen fall, caused by fainting
F06P	Fainting from a chair
F07P	Fall while trying to get up from a chair
F08P	Fall while trying to sit down in a chair

Table 4.1: Types of activities selected for our dataset, derived from the SisFall ones. We chose activities with a very different waveform from one another, and excluded some impractical ones (e.g. sitting inside a car).

activities derived from the SisFall activities (listed in Table 4.1), while wearing the body network described in the previous chapter.

The body network will be composed of three smartphones positioned in different parts of the body:

- The right shoulder, fixed with an armband.
- The waist, fixed with a belt.
- The front left pocket of the pants.

The three smartphone will record the accelerometer signals, and a tablet will act as a controller, sending commands to the smartphones and recording the video of the activity with the body network architecture described in Figure 3.7.

During a preliminary session we tested our system with one volunteer performing all the activities. Some pictures taken during this session are shown in Figure 4.1.

At the end of the session, both the accelerometer signals and the relative videos are stored in the tablet.
The following step is to annotate the video. This will be done manually with a software based off a subtitle editor. As described earlier, this procedure will be done by human beings, with the rationale that seeing the video of the performed activity will help a human annotator to easily determine the beginning of each sub-activities (e.g. walking, stumbling, falling). Since the video will be synchronized with the accelerometer signals, the labels from the video will be automatically translated to the labels of the signals.

These signals and their relative labels will then be organized in a yet to be defined format, to form our new dataset. The resulting dataset will then be used for the training phase.

4.2 Training

Our LSTM-based neural network architecture, described in the previous chapter, will be trained using the new dataset.
We set up a software infrastructure to perform the training, using python and TensorFlow on a workstation, based on the original work by Chevalier (2016).

To feed the dataset to our network, we decided to split every activity using a sliding widow (with N to be defined and $s = 1$). The label corresponding to this window will then be computed in the following manner:

- If the window contains at least one temporal label of a fall, the entire window is labeled as fall.
- If the window contains at least one temporal label of a warning, but no falls, the entire window is labeled as warning.
- If the window does not contain neither warnings nor falls, it is labeled as ADL.

The neural network will then have as input a list of windows and a list of labels (one label for each window).

While the network architecture is the one described in Chapter 3, we cannot exclude some changes and/or experimenting other architectures, based

on the results obtained. Also, a wide variety *hyperparameters* will probably change during the training phase. Hyperparameters are parameters independent from the network architecture, that must be decided a-priori. Since they greatly influence the outcome of the network, we will try various combinations of them to be able to get to the best accuracy. The initially chosen hyperparameters are listed in Table 4.2
Other changes during this phase are not foreseeable here, and will be decided based on the results.

Name	Value
Learning rate	0.0025
L2 constant	0.0015
Training epochs	300
Batch size	1500
LSTM size	32

Table 4.2: The hyperparameters as chosen by Chevalier (2016). The learning rate influences the gradient descent method; the L2 constant influences the regularization of the network; the number of training epochs is the number of times the entire dataset goes over the whole network; the batch size is how many samples go over the network at the same time; the LSTM size indicates how many cells are inside each layer.

After training the network, we will save the frozen model in a binary file, using an automatic procedure provided by TensorFlow, so we can deploy it on a smartphone. As described before, the frozen model contains all the informations about the trained network relevant to performing inference.

4.3 Testing on Mobile Devices

We will then deploy the frozen model to a Android smartphone, to perform the testing of our fall detection system in a real-case scenario.

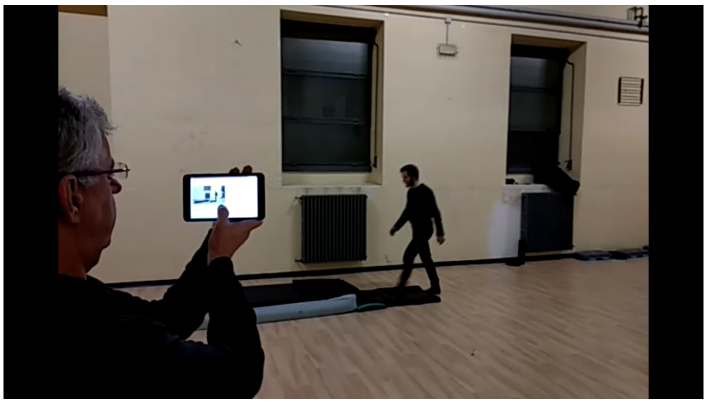
We already wrote an Android application that:

- Takes the frozen model as input.
- Reads signals from the accelerometer.
- Runs these signals through the frozen model using TensorFlow Mobile.

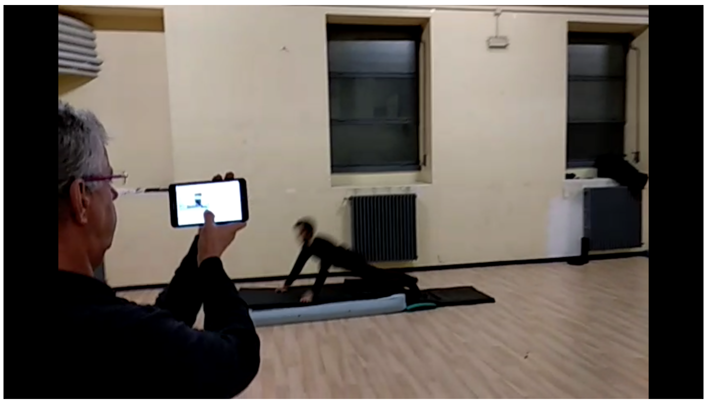
- Shows the output of the network to the user.

The app is absolutely similar to the one described in Chapter 2 and shown in Figure 2.3.

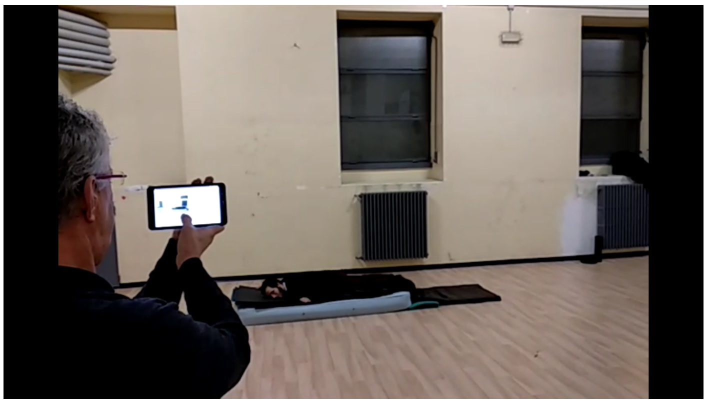
Once the training phase will be completed, we will simply transfer the frozen model from the workstation to the smartphone and test our system using the developed app.



(a)



(b)



(c)

Figure 4.1: A volunteer performing activity F01P during a preliminary testing session. The volunteer is carrying three smartphones; the tablet controlling them and recording the video can be seen.

Chapter 5

Conclusions

In this thesis we described the development of machine learning techniques applied to automatic fall detection.

After describing the background theory regarding the automatic fall detection techniques and the state of the art, specifically the SisFall dataset, we introduced machine learning, focusing in particular on recurrent neural networks and LSTM.

We summarized the tools used, and the reasons that led us to chose the Android operating system and the Tensorflow framework.
We explored what a smartphone application for fall detection should do, focusing on the Android operating system.
TensorFlow and its functions were then described: we explained how TensorFlow works in a smartphone platform, and how we can train a network on workstation and use the trained model on mobile devices.

We presented the experiments we performed, the transition from the SisFall dataset to a newer one, and the architecture we chose to use. We firstly replicated the SisFall results, then applied machine learning techniques to their dataset.
Going beyond, we described how a fine-grained labeling would lead to better results, and developed a plan to acquire a new, enhanced, dataset. We also developed a body network of sensors to collect the accelerometer recordings. Lastly, we implemented an Android app capable of using a trained neural network.

In the last chapter we discussed the results obtained, and outlined the steps to take in the immediate future.

From a scientific standpoint we learned that:

- Automatic fall detection is an open, difficult, problem.
- A common way to tackle this problem is to elaborate the signals from accelerometers.
- There have been some approaches in literature, with the SisFall one creating a dataset of recorded activities.
- We can record accelerometer signals coming from various smartphones using a body network architecture and create a new, enhanced, dataset.
- Using machine learning we could improve fall detection. Specifically, use TensorFlow to train a network (composed of LSTMs) specialized for treating signals that vary over time.
- The labeling of the dataset is a crucial aspect of machine learning problems.
- We can use a trained network directly on a smartphone, using TensorFlow Mobile, to make real-time predictions.

In the near future, other activities regarding this project could follow:

- Improvement of the body network architecture. Smartphones are bulky, and carrying more than three is uncomfortable. This could be improved by using a small and low-power sensor tile, and by increasing the numbers of sensors. Some openings in this direction are currently at work.
- Improvement of the neural network architecture with more sophisticate networks than LSTM. The main problem of the LSTM is that they need a cache of signals to be kept, since every output depends on the signals coming before it. A network capable of update its state without keeping a cache would greatly improve performances.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.*

Anguita, D., Ghio, A., Oneto, L., Parra, X., and Reyes-Ortiz, J. L. (2013). *A Public Domain Dataset for Human Activity Recognition Using Smartphones.*

Bengio, Y., Simard, P., and Frasconi, P. (1994). *Learning long-term dependencies with gradient descent is difficult.*

Chevalier, G. (2016). *LSTMs for Human Activity Recognition.*

Csáji, B. C. (2001). *Approximation with artificial neural networks.*

Cybenko, G. (1989). *Approximation by superpositions of a sigmoidal function.*

Domingos, P. (2012). *A few useful things to know about machine learning.*

Feng, P., Yu, M., Naqvi, S. M., and Chambers, J. A. (2014). *Deep learning for posture analysis in fall detection.*

Frank, K., Vera Nadales, M. J., Robertson, P., and Pfeifer, T. (2010). *Bayesian Recognition of Motion Related Activities with Inertial Sensors.*

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). *Learning to forget: Continual prediction with LSTM.*

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.*

Hochreiter, S. and Schmidhuber, J. (1997). *Long Short-term Memory.*

Medrano, C., Igual, R., Plaza, I., and Castro, M. (2014). *Detecting Falls as Novelties in Acceleration Patterns Acquired with Smartphones.*

Siegelmann, H. T. and Sontag, E. D. (1995). *On the computational power of neural nets.*

Sucerquia, A., Lopez, J., and Vargas-Bonilla, J. (2017). *SisFall: A Fall and Movement Dataset.*

Vavoulas, G., Pediaditis, M., Chatzaki, C., Spanakis, E., and Tsiknakis, M. (2016). *The MobiFall Dataset: Fall Detection and Classification with a Smartphone.*

Vilarinho, T., Farshchian, B., Gloppestad Bajer, D., Halvor Dahl, O., Egge, I., Steinsland Hegdal, S., Lønes, A., N. Slettevold, J., and Mathias Weggersen, S. (2015). *A Combined Smartphone and Smartwatch Fall Detection System.*