Lightweight Semantic Location and Activity Recognition on Android Smartphones with TensorFlow

BY

MARCO MELE

B.S. in Computer Engineering,

Politecnico di Torino, Turin, Italy, 2016

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2018

Chicago, Illinois

ACKNOWLEDGMENTS

The first and most important thank goes to my parents; their endless faith in me and their passion in providing me with all the essential tools for a successful life gave me what has been essential to achieving this milestone. With them, I want to thank all those who we as a family miss every day, for the opportunities they provided us all with through hard sacrifice.

I cannot fail to thank my closest friends, who have learned to stay close even when significantly distant; and to all those I shared this particular experience with, for our reciprocal, unfailing support for each other. I also want to give a special thank to those who have set a significant example of perseverance and accomplishment for me; to them, I owe a big share of the motivation that led me here.

Finally, this work is the result of the participation of two Academies that for almost two decades have chosen to give to some of their students the opportunity of a double-degree Master program. An acknowledgment goes to my home institution, the Polytechnic University of Turin, Italy, and my host, the University of Illinois at Chicago, Illinois, and to all their people who made this project possible.

MM

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF TABLES (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AP	Access Point
API	Application Programmer Interface
AQI	Air Quality Index
AR	Activity Recognition
ASIC	Application-Specific Integrated Circuit
BN	Bayesian Network
CAA	Clean Air Act
CART	Classification And Regression Tree
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CVG	Coriolis Vibratory Gyroscope
DL	Deep Learning
DT	Decision Tree
EDGE	Enhanced Data rates for GSM Evolution
FFT	Fast Fourier Transform
GNSS	Global Navigation Satellite System
GPRS	General Packet Radio Service
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communication
HAR	Human Activity Recognition
HCI	Human-Computer Interaction
HMM	Hidden Markov Model
HPC	High Performance Computing
HSPA	High Speed Packet Access
ICT	Information and Communication Technology

IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineers
IoT	Internet of Things
LBS	Location-Based Services
LR	Logistic Regression
LSA	Latent Semantic Analysis
LSTM	Long Short-Term Memory
MEMS	Micro-Electro-Mechanical Systems
ML	Machine Learning
NAAQS	National Ambient Air Quality Standards
NB	Naïve Bayes
NN	Neural Network
OCR	Optical Character Recognition
OS	Operating System
PCA	Principal Component Analysis
PLS	Partial Least Square
RF	Random Forest
RNN	Recurrent Neural Network
RSS	Radio Signal Strength
SDK	Software Development Kit
SL	Semantic Location
SLAR	Semantic Location and Activity Recognition
SVM	Support Vector Machine
TF	TensorFlow
TPU	Tensor Processing Unit
U.S. E.P.A.	United States Environmental Protection Agency
UIC	University of Illinois at Chicago
WHO	World Health Organization

SUMMARY

This work brings together several technology concepts that are rising exponentially in the plethora of applications that computing has today. Context-awareness is the parent concept that guides the applications we will discuss here. It relates to all those applications of technology that enhances whatever provided service with the knowledge that is strictly related to the user, the surrounding environment, and their properties and situations. Context-aware technologies are based on the principles that the technology adapts as needed, so that interaction with it varies depending on what is more suitable in a particular context. This concept is tightly related with ubiquitous computing, a term that encloses all those technologies that have pervasively made their way in countless aspects of our everyday life.

The main goal of this work is to be incorporated into a well-being oriented project, that wants to enhance the existing technologies related to air quality information, with personalized personal exposure to air pollutants. This detail is strictly related to the environment the user is in, and to the activities they are involved in. What we want from this work is to find a way to approach the problem of Semantic Location detection and Activity Recognition, addressing two of the main factors in pollutants exposure: involved activity and microenvironment.

We focus this work on exploiting the possibilities that mobile device can offer today, motivated by their pervasive presence in our daily life, their always increasing technological capabilities and their diffusion: more than seventy-five billion devices will be connected to the internet by the year 2025—this is ten times the Earth population today.

We use onboard sensors on common smartphones to collect motion data for activity recognition, and other sensors and similar information to determine indoor or outdoor positioning. We exploit the following smartphone sensors and components: motion sensors (accelerometer and gyroscope), microphone, magnetic sensor, radio signal details, light, and proximity sensors. This approach is supported and inspired by plenty of research literature and tailored to our specific needs. We work with Android devices and aim to build a prototype for a mobile application that can collect this data from the device and run a machine learning based model to run inference locally.

For this quite complex machine learning task, we study the possibilities that today's advancement in Neural Network and Deep Learning technologies have reached to attempt at a model that is able to carry out the job. We introduce a less known version of neural networks called Long Short-Term Memories that are uniquely intended to work with temporal data, and that today is employed by most of the major technology companies out there for their top tier products involving machine learning.

Finally, we rely on a solid machine learning programming framework called TensorFlow and intended from its creator Google itself to help scientist work with machine learning. This will allow us to quiet easily port the result of this work in a mobile environment.

CHAPTER 1

INTRODUCTION

There was a clear moment in the evolution of Information and Communication Technology (ICT) when we started referring to our mobile phones as *smartphones*. Mobile devices do more things that just serve as telephones, mostly because the *firmware* that runs on them today can be compared to the Operating System (OS) of a computer, thus being able to run user-developed applications for the most various tasks.

Along with this changes, mobile phones and the applications that run on it, have gained the property of being **context-aware**. For long, special-purpose devices have been called *location-aware*, as for them to provide functionalities, they were—usually continuously—aware of the device location; examples of such devices are navigational systems, like the Global Positioning System (GPS). But today's devices are more than just location-aware.

Context awareness is a more wide term, that refers to a device, or application, that is in any mean *aware of the context that surrounds the user*, and is particularly appropriate for smartphones, being one of the most important keywords under the big umbrella to which we refer as **Pervasive and Ubiquitous Computing** (Section 1.1).

The behavior of a context-aware system is not *static*; instead, it *adapts* accordingly to various pieces of information that constitute the context—location, actions, time, and countless more.

Dey [18] defines context as follows.

Definition 1.1 (Context)**.** *"[. . .] any information that can be used to characterize the situation of [. . .] a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

The most simple form of context-dependent behavior, although very basic, is probably the adapting of the content of the screen based on the orientation of the device—portrait or landscape; but the applications are countless: from a navigation application to a game, a keybaord (spell checking and prediction) to a personal assistant (generating content, recognizing the user who is asking, getting the right thing done). The scopes context awareness applies to are various and usually more complex—to cite some of them, we can think of:

- adapting the application's interface;
- limiting, or choosing, the data the user interacts with to what is relevant in the current context;
- propose new services, target advertising, adapt the way services are offered, and countless more applications.

In this work we study and build a framework for Semantic Location and Activity Recognition (SLAR) to help create a context-aware system for mobile devices. We will have an overview of the work we want to conduct in Section 1.7, after briefly going through some background in Sections 1.1 to 1.6 to support our intents.

1.1 Pervasive and Obiquitous Computing

Ubiquitous Computing—also called Pervasive Computing or Ambient Intelligence—is a critical concept in Computer Science that refers to the phenomenon of having computation, in its widest meaning, appear in several different places at any time, in a way that it becomes extensively "intrusive" in the user experience of everyday life.

Computation is no more limited within computers, smartphones or tablets; instead, it is embedded in wearables (e.g. smartwatches, glasses), domestic appliances—refrigerators, thermostats, sensors—and even cars, urban decor, and many other countless entities equipped with the smallest microprocessor. Those objects are then almost always connected to a network, usually the Internet; this is why we usually refer to all of those with the term Internet of Things (IoT). In 2015, the number of IoT devices connected to the internet was 15.4 billions; in 2025, it is expected to be more than 75 billions—that is, almost ten times the current Earth population.[1]

Ubiquitous computing and all its various names we just listed, along with several more, cannot be correctly defined or narrowed down to a precise concept; nevertheless, it is strictly related with other concepts in Computer Science, like **distributed computing**, Human-Computer Interaction (HCI) and **Artificial Intelligence (AI)**. All these concepts—Ambient Intelligence,

[1]From: `www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide`, visited June 2018.

IoT, and AI are considered among the most promising **emerging technologies** of the moment (June 2018) in ICT.[2]

Pervasive Computing inevitably arises many **privacy** concerns, the more that the "common" user, the one who did not major in CS, is in most cases completely unaware of how much *pervasive* this computing actually is, and of all the concequences of the case, especially the ones arising from the pairing of pervasive computing with AI and Machine Learning (ML); worth mentioning is also the fact that legislation, as complex as it is within all Computer Science, is usually steps behind in regulating user privacy, not being able to stay abreast with the pace at which technology evolves.

1.2 Activity Recognition

While Activity Recognition (AR) seems a quite straightforward term, its formal definition is hard to express for it is a more generic concept than it seems. One definition can be as follows:

Definition 1.2 (Activity Recognition)**.** *The science that aims at recognizing the actions of an agent through the observation of the agents' actions and environment.*[3,4]

[2]From: en.wikipedia.org/wiki/List_of_emerging_technologies#IT_and_communications, visited June 2018.

[3]From: en.wikipedia.org/wiki/Activity_recognition, visited June 2018.

[4]In Computer Science, we usually define an **agent** as anyone, or anything, that has an active role within a process.

This definition is vague as AR applies to many different fields, and a more strict definition would not catch its many-faceted nature. Its applications are indeed countless: surveillance, HCI, healthcare. In all these examples, AR strictly refers to human beings' activities; having this in mind we can formulate a more precise definition for Human Activity Recognition (HAR) given by Yang et al. [64]:

Definition 1.3 (Human Activity Recognition)**.** *The ability to interpret human body gesture or motion via sensors and determine human activity or action.*

This sure tells us more of what we actually aim to with HAR, and how we do it. Definition 1.3 introduces a key concept: *how* does one *detect* gestures and motion to interpret. Indeed, the means we dispose of to capture human activities define different approaches. Widely speaking, there are sensors that record human activities generating various types of data to be interpreted, and are the sensors and the data they produce that discriminate different branches of HAR.

Video-based This type of HAR is based on one or more cameras recording the environment in which the agent moves, generating a sequence of images that are then processed by the recognition algorithm. While it is simple to deploy as hardware, it is particularly more complex to work with the collected data, as it requires many steps of processing, interpretation and feature extraction.[5]

[5]*Feature extraction:* to be distinguished from *-selection*, the process of deriving *new, additional* data and values from the original set of collected data, usually aiming at reducing the *dimensionality* (i.e. the number of *features*,

Depth-based sensors With *depth sensor* here we refer to infrared sensors or cameras [32]. Depth sensors are deployed around the environment and emit an infrared signal to detect, through variation in measured depth, agents and their movements.

Wearable-based sensors This last technique uses one or more sensors attached to the human body in either generic or specific positions. Most commonly used sensors are motion detectors like accelerometers, gyroscopes, and magnetic sensors [5]. We will see in Section 2.1 that these sensors can either be special-purpose circuitry or general purpose devices like smartphones [26], the latter providing an even wider choice of sensors like ambient information and GPS modules.

All these methods have, like always, their *pros* and *cons*; the [5] survey on HAR shows that the last two methods, with depth and wearable sensors, are gaining a larger share in the HAR research fiels (as of November 2014). Video-based methods indeed do have particular drawbacks, like not being fully capable of capturing motions in a three-dimensional environment, and have more complex and resource-needing requirements to process imagery data [47]. Both RGB cameras and depth sensors have limitations on what the device *can* actually perceive; the most common issue is *occlusion*, like backgrounds, shadows, and light conditions that add noise to the motion detection [63].

also called *variables*, *covariates*, or *predictors*) and the redundancy in the dataset, or to better express an aspect of the data that carries the same or more information at a less cost, e.g. data variation or other statistical transformations. Common examples of feature extraction algorithms are PCA, PLS Regression, and LSA.

Wearable sensors overcome not only the complexity of the processing of camera footprints, but provide cheaper and less intrusive recording mechanisms with higher data variability—measure different phenomena at the same time—and accuracy, with higher sampling rates and location-independent—we will come back at this concept in Section 2.1.1.

As true as it is that video-based HAR raises higher privacy concerns, wearable sensors are not exempt from the matter, or at least more concerning than one might think—we overviewed this aspect in Section 1.1. On the other hand, the troubling problem of covering the human body with several, special-purpose measuring sensors, is now being overcome by using devices with embedded sensors that we already wear everyday without expressing too many concerns: smartphones, smartwatches, and sports equipement.

1.3 Semantic Location

Location is another key concept in Pervasive Computing, as it determines a crucial aspect of context-aware systems. Humans have always been concerned with expressing and representing location for centuries using maps, coordinates systems, and addresses. But what *semantic* location is more concerned about is defining a location not—or not only—by its *georeference*; instead, it focuses on classifying locations based on their "meaning". We might want, for instance, to classify places under categories like *home*, *restaurant*, *grocery store*, *university*, and similar, to give the location a deeper meaning than just its geographical position [39, 48].

Location-aware applications are a precise category within context-aware systems that, in the same way as other various aspects of Ubiquitous Computing, have made their way into everyday life, giving birth to what we call Location-Based Services (LBS) [13].

Having an insight of what type of a location the user is at, rather than just its position, is essential to provide both targeted and quality services. The difference between positioning and the semantic location is crucial both from humans and machines. To the extent of LBS, identifying a location as "40.411692 N, 16.689446 E" or "*320 N Morgan Street, Suite 600, Chicago, IL 60607*" has way less meaning than "home", "workplace", the restaurant we are at, or the hotel where we are staying. We will see in Sections 1.7 and 2.2 that here we are actually less concerned about giving a name to a location like "home" but rather distinguish between indoor and outdoor environments.

There are plenty of applications for LBS: emergency intervention, healthcare, recommendation systems, navigation, advertising, and a lot more. Just like we said applications and services vary depending on the user's context (Section 1.1), they vary depending on their semantic location, providing service differentiation, quick access, and information and action filtering.

1.4 Mobile Devices and Limitations

Of course by mobile devices we are talking **smartphones**: they fit in a pocket and can be operated with—usually—just one hand; they board a touchscreen display, virtual and physical keys, speakers, ports, and all needed for basic (and not) functionalities: antennae, microphone, sensors, battery, and an always more powerful computing unit: today's smartphones are not much less powerful in computation than mid-level laptops.

Smartphones run an Operating System, in charge of providing telephone functionalities and executing native and third-party applications, usually referred to as "*apps*". This modular structure has allowed, through the parallel developing of both the OS and the applications, the growth of the services and functionalities that a common smartphone is capable of providing: starting from calls and texts, to digital cameras and navigation, to browsing, access to any kind of online content, to the integration of virtual assistants.

The market-reaching speed with smartphones has been tremendously fast: in 2007, there were slightly more than 109 million devices that could be called smartphones; in 2016, it was 20 times more, over 2.1 billion. For 2020, it is expected more than 2.8 billion units sold.[6]

Mobile computing is crucial today, and relies on four fundamental principles:

Portability devices must have an easy mobility, users must be able to carry them around with no effort; at the same time, they must have sufficient computing capability, memory, and battery time.

Connectivity is crucial that devices keep their connections—telephone network, internet—alive as much as possible, despite their fast mobility between network nodes and an-

[6]From: `www.statista.com/statistics/330695/number-of-smartphone-users-worldwide`, visited June 2018.

tennas; a device moving along with a car or train can change its network Access Point (AP) every few minutes.

Interactivity devices must be able to interact with one another to provide functionalities, collaborating and transferring data.

Individuality they adapt to the individual use of its user.

But like most technologies, the growth of mobile computing on smartphones is limited by issues that the IT world is struggling to overcome:

(i) Bandwidth for Internet access is usually limited on mobile; in the last few years, old mobile networks like GPRS and EDGE have been replaced with faster technologies like HSPA, 3G, 4G, and shortly 5G, which is currently being tested worldwide.

(ii) Security is a big issue, enhanced on mobile as smartphones are very often connected to public Internet APs that do not guarantee an high level of security.

(iii) Power usage is definitely one of the most issues in mobile computing. The goal is to be able to keep the device going on battery as long as possible. This concern must be addressed not only by manufacturers but also, and especially, by OS and applications developers. The most consuming operations for a smartphone are wireless network connectivity (e.g. calls, browsing, and navigation), screen-on time, and high power computation (e.g. gaming).

This is not all, other issues arise like accessibility, health and safety, and have been for long discussed by the research community and the corporate world.

1.5 Long Short-Term Memories

Long Short-Term Memory (LSTM) are a—possible—building unit of a Recurrent Neural Network (RNN) (see Chapter 5); RNNs made of LSTM cells are usually called **LSTM Networks** or just **LSTMs**. This particular type of cells was introduced by Hochreiter and Schmidhuber [28] in 1997 and their work was followed and refined by plenty of literature work (see Section 2.4). LSTMs are currently largely used as they particularly suit a large variety of problems in classification and in processing and predicting time series.

What LSTMs do well is to overcome a recurrent problem in RNNs, called the **long-term dependency problem**. Put in few words, RNNs struggle to learn long-term dependency among data, i.e. they hardly keep and use a past information in a future time that is somewhat distant. LSTMs easly remember both short- and long-term dependencies among data for an arbitrary interval, with a extraordinary, "built-in" functionality that enables them to decide which pieces of information to keep and for how long, learning the impact on both near and far points in time—with time, we refer to the succession of data records which have a temporal nature. LSTMs were also designed to deal with the **vanishing gradient problem** giving them an advantage with respect to RNNs and other methods like Hidden Markov Models (HMMs).

LSTMs are today fundamental pieces of top-notch products and services from major companies like Google, Apple, and Amazon. We will see in Chapter 5 how LSTMs will suit our problem of Semantic Location and Activity Recognition, the benefits they apport and how they actually work in better detail (Section 5.4).

1.6 TensorFlow

TensorFlow™ (TF) is an open source library for High Performance Computing (HPC) developed by the Google Brain team at Google.[7,8] TensorFlow's paradigm is called **dataflow programming**: a program is a directed graph representing (mathematical) operations through which data *flows*, similar to the paradigm of **functional programming**.[9,10] TensorFlow runs both on single devices and on multiple CPUs or GPU and is multi-platform. The name derives from the term **tensor** with which the developers identify the data arrays involved in the computations.

In May 2017, Google announced its second-generation Tensor Processing Unit (TPU), an Application-Specific Integrated Circuit (ASIC) designed specific for ML with TensorFlow. TPUs have an high throughput of low-precision arithmetic that deliver up to 180 teraflops, organized into clusters of 64 TPUs each called **TPU pods**, for a total of 11.5 petaflops per

[7]`ai.google/research/teams/brain`.

[8]From: `www.tensorflow.org`, visited June 2018.

[9]From: `en.wikipedia.org/wiki/Dataflow_programming`, visited June 2018.

[10]Directed graph: *a set of points called* vertices *interconnected by links called* edges *s.t. edges can be traversed along a specific direction only, and are usually represented as arrows.*

TABLE I: TENSORFLOW SPECIFICATIONS.

Developers	Google Brain Team
Initial release	November 9, 2015
Latest stable release	1.10.0, August 8, 2018
Repository	`github.com/tensorflow/tensorflow`
Languages	Python, C++, and CUDA
Platforms	Linux, macOS, Microsoft Windows, Android, and website
Licence	Apache 2.0 Open Source Licence
Website	`tensorflow.org`

pod [29].[11] In February 2018, TensorFlow became part of the Google Cloud Platform suite [30].[12] Today, TF is used by more than 40 major companies other than Google itself, including AMD, NVIDIA, Uber, Qualcomm, ARM and more.[13]

1.6.1 TensorFlow for Mobile

TF was designed to integrate perfectly with mobile OSs like Android and iOS, helping developers with complex tasks like speech and image recognition, Optical Character Recognition (OCR), translation and voice synthesis. In May 2017, in occasion of Google's annual

[11]Petaflop: *a unit of computing speed equal to a million million* (10^{12}) *of FLOating-point OPerations per Second (FLOPS).*

[12]`cloud.google.com`.

[13]From: `www.tensorflow.org`, visited June 2018.

keynote Google I/O, the company presented TensorFlow Lite [60], a more lightweight solution for mobile devices providing fast inference and small binary size.[14]

TensorFlow Lite comes with three pre-trained models but only a subset of all the operators available in TensorFlow; this is the reason why this lightweight version does not meet the needs of this work, for our ML model needs operators available only in the regular mobile version. We will see throughout this document that this will not lead to performance degradation.

1.7 Motivations and overview

Now that we have given some background knowledge on some concepts, we can go through an overview of what this work is about. In order to motivate the objectives, we need to introduce a larger project that this work aims at supporting.

The goal is to monitor one's **Personalized Exposure** to air pollutants combining the latest technologies in Earth observation and smartphones. The project is carried out by the Department of Mathematics, Statistics, and Computer Science and the Department of Civil and Materials Engineering of the University of Illinois at Chicago (UIC), partnered by the AirNow Program of the United States Environmental Protection Agency (U.S. E.P.A.).

According to the **World Health Organization (WHO)**, about 7 million people died in 2012 only for air pollution-related diseases [3], one of which only in China [4]. By mandate of the Clean Air Act (CAA), the U.S. E.P.A. is in charge of defining National Ambient Air Quality

[14]From: `www.tensorflow.org/mobile/tflite`, visited June 2018.

Standards (NAAQS) for pollutants. The NAAQS define six pollutant measurements: Carbon Monoxide (CO), Lead (Pb), Nitrogen Dioxide (NO_2), Ozone (o_3), Particulate Matter (PM_{10} and $PM_{2.5}$), and Sulfur Dioxide (SO_2).

In 1998, the U.S. E.P.A. initiated the **AirNow** program, providing real-time information on localized air quality data, with forecasts and an Air Quality Index (AQI).[15] The aim is to let the public adjust their living habits accordingly to potentially harmful air conditions to reduce their personal exposure—one might not want to go running in downtown Manhattan in a week with high levels of PM_{10}, or camp near a field subject to high concentrations of lead.

The AQI ranges from *Good* (0–50), when *"air quality is considered satisfactory, and air pollution poses little or no risk"*, to *Hazardous* (301–500), with *"health warnings of emergency conditions, the entire population is more likely to be affected."*.

Pollutants concentration estimates have been continuously improved with high-resolution satellite imagery, but we are not close already to a personal exposure index. Personal pollutant intake is not only about the AQI in the area people live in, but strongly depends on other major factors:

(i) The person's **microenvironment**, e.g. *indoor*, *outdoor*, or *in vehicle*;

(ii) The **activity** they are involved in, e.g. *running*, *walking*, *standing* or *biking*;

(iii) The individual **physiology**, i.e. age, gender, and health conditions.

[15]See: `www.airnow.gov`.

With this intent, the **MY-AIR** (**M**onitor **Y**our **A**ir-pollution **I**ntake and **R**isk) project aims at providing users with a personalized exposure index leveraging satellite data with Semantic Location and Activity Recognition (SLAR) models running on people's smartphones.

In this very part of the work we aim at designing Semantic Location and Activity Recognition (SLAR) module for a mobile application, which is able to classify user activities and environment, meeting the limitations of mobile devices in terms of computational capabilities, power consumption and data protection. Combining Semantic Location and Activity Recognition we will be able to identify these nine categories:

(i) indoor biking
(ii) indoor running
(iii) indoor stationary
(iv) indoor walking
(v) in vehicle
(vi) outdoor biking
(vii) outdoor running
(viii) outdoor stationary
(ix) outdoor walking

1.8 Environment Setup and Tools

This is an overview of the most important tools for the major parts of this work. More details will be given in each dedicated section.

Android application Android Studio and Android Software Development Kit (SDK) with API level 26, and smartphone with Android 8.0 "Oreo"

Data preprocessing Python with SciKitLearn, Pandas, and Numpy

Model training Keras framework with TensorFlow backend and CUDA, running on a GPU NVIDIA GeForce GTX 1060M

CHAPTER 2

RELATED WORK AND STATE-OF-THE-ART

In Section 1.7 we have gone through an overview of what this work is about. Before starting with the details of this implementation, we want to take a look of what the research community has reached so far in similar work, along with commercial products with similar purposes.

As the work embraces many aspects that are not necessarily always addressed together, we will go through the related literature per different sectors, addressing the following main aspects separately and, whenever possible, together as well:

(i) Human Activity Recognition
   - (a) special-purpose wearable sensors
   - (b) location-oriented, special-purpose wearables
   - (c) Human Activity Recognition on mobile devices

(ii) Semantic Location
   - indoor/outdoor detection

(iii) combined Semantic Location and Activity Recognition

(iv) Long Short-Term Memories

2.1 Human Activity Recognition

Activity Recognition (AR), in its most wide and general meaning, has been extensively approached by the research community, due to the width of contexts it can be applied to. How we briefly addressed in Section 1.2, AR is a name with a broader meaning that the one we will refer to, as it also embraces the video-based task in another area of pervasive computing called **Computer Vision**.

The specific setting we will be referring to here on—unless explicitly specified—is a **sensor-based AR**, where the word *sensor* identifies a *sensing device*. These sensors are usually either embedded in the environment (e.g. IoT, *smart houses* or, more widely, *Ambient Intelligence*) or in other devices with a different main purpose; those usually would be devices that we usually carry on ourselves most of the times: this would be the case for smartphones, smartwatches, and any other kind of *"intelligent"* carry-on or wearable appliance [26].

Simple Activity Recognition has been out there for more than a while now, yet accurate classification of human activities remains a non trivial task and is still a challenging research area [33]. The task of recognizing human activities strongly depends on the goals of the classification. The different types of targets for the classification can be as much as we can think of: we perform a variety of activities during the day, from driving to watching a movie, from typing to cooking and countless more. If one wanted to carefully classify each action we take during a day, it would be quite an hard task, so one usually decides on a set of actions is really interested in and tries to detect them.

From the selection of the set of activities we intent to infer, arise different problems, harder in some settings more than in others. For instance, one might want to detect when the user is watching television, something we commonly do while performing other activities, like cooking, or having a conversation with someone else in the room [33], which is also a good example of **multi-agent activity**, i.e. an activity to whom more than one agent take part—remember that we defined the agent in the context of Definition 1.2, Section 1.2. In the work we carry out we will be dealing with mutually exclusive activities (see Section 1.7), thus concurrent activities will not be a issue we shall take care of.

The largest majority of research work in Activity Recognition prformed during the late 1990s and early 2000s was mostly under the category of video- or image-based AR [68]. Video-based classification suffers of high costs if deployed outside the scope of scientific research, as it requires the equipment with multiple, good resolution cameras and an unordinary need of computational capacity.

Sensor-based AR has firstly relied on special-purpose, weared sensors to collect motion data in specific positions on the human body. With *special-purpose* sensor we indicate any type of sensor which only serves that particular function or was appositely designed, *ad hoc*, for a specific task; special-purpose sensors are usually not integrate within more complex devices (like a smartphone sensor), instead it usually is an independent unit that most of the times is bigger than an equivalent, embedded sensor.

A lot of work has been carried out on this type of AR, especially in health care; yet, wearable, large sized sensors to be strapped, fasten to an arm or chest are not something one would desire to live with all day long. Since the exponential spreading of Micro-Electro-Mechanical Systems (MEMS), research efforts in AR with wearable sensors have largely increased [68], given the ease of data collection from miniature-sized inertial sensors. Then, since the advent of MEMS–equipped smartphones, many have invested in mining the data that these sensors continuously produce, firstly for pure functionality purposes, then by app developers and data analytics companies with the goal of improving context-aware services and applications with the collected data.

2.1.1 Special-purpose wearable sensors

Plenty of work has focused on AR from motion data generated by special-purpose sensors fastened to the subject's body. This type of Activity Recognition relies on the use of wired or wireless motion sensors applied in a specific position of the body. Most of the work conducted with this settings is less recent and, for these sensors not being practical to wear in everyday life, these approaches have an application scope which is mostly academic or medical, aiming at the monitoring and well-being of patients [35] and elderly [14, 16, 52].

Yang et al. [64] set up a distributed sensor network over the human body for monitoring and recognition; a large number of sensors simultaneously recording data from different positions introduces a significant complexity overhead in feature extraction, data processing, and classification algorithms: a well-known problem in computational learning usually referred to as the **Curse of Dimensionality** [31]. Here, data is highly processed with Fast Fourier Transform (FFT) and after feature selection, the work showed how the number of sensors could be easily reduced from 8 to 2 while still achieving a precision of 94 %, reducing the cost of both implementation and computation. Yet, the results of the work are strictly dependent on the position of the sensors on the body.

Najafi et al. [45] in 2003 presented one of the first AR works in medical assistance for elderly with non-invasive technology, using one single special-purpose motion sensor strapped to the chest. Their work consisted in detecting the patient walk time and posture transitions, i.e. from sitting to standing or lying. Data analysis was performed with signal processing and transform techniques, of high power computation requirements, that led to accurate results in activities and posture transition classification.

2.1.2 Location-oriented, sensor-based Activity Recognition

Zhu and Sheng [66] proposes a new approach to recognize indoor human activities with wearable motion sensor data combined with location data in indoor environments. The experimental setup involves a motion sensor strapped to the subject's body and an optical capture system to detect the subject's location within a room or a more generic indoor space. The location data is then used to help the inference of the human activities with a two-step algorithm that exploits two, differently grained classifications with Neural Networks to infer basic activities; lastly, the activity is fed to a Hidden Markov Model (HMM) to model activity sequentiality constraints.

2.1.3 Human Activity Recognition on Mobile Devices

We anticipated in Sections 1.1 and 1.4 how mobile devices are a great tool for AR as they provide many onboard motion sensors providing various type of data. Since the rise of more sophisticated smartphones with good precision embedded sensors, the research community has begun to exploit these devices. Onboard orientation-aware sensors (e.g. gyroscope and accelerometer) are shown to improve classification performances of more than 10 % with respect to special-purpose circuitry [16].

One of the most interesting studies conducted by Dernbach et al. [16] in 2012 shows how smartphone's motion sensors are a great alternative to on-body sensors. In this study, the researchers exploit tri-axial accelerometer and gyroscope—we will describe these sensors better in Chapter 3. This work points out that smartphone motion sensors provide a great advantage with respect to on-body sensors in terms of **orientation-aware** data; this means we are able to obtain motion data independently of the sensor's orientation and relative movement, albeit we must carefully correct the data in order to always meet a fixed coordinate system; Section 3.2 shows how accelerometer data is corrected for mobile devices and what the Android APIs provide to perform this task.

Dernbach et al. [16] also show how the classification is improved by implementing a **sliding window classification**. An activity is composed of a sequence of movements that vary over time, therefore to observe a time window of motion data provides a lot more information than just a simple record at a precise time. Not only sliding windows of data are shown to be optimal for sensor data, but it is empirically proven that classification accuracy is stable across **different window lengths** [8]. They then perform some feature extraction on the classification window (e.g. mean, variance, minimum, and maximum values) to help simple classifiers like Naïve Bayes (NB), Bayesian Network (BN), and Decision Tree (DT).

As pointed out before, one of the most common applications for HAR is healthcare and well-begin. An example of early work with smartphone sensors was conducted by Lane et al. [35] in 2011. The work consisted in developing a smartphone application that tracks sleep patterns, social interactions and activities. For the AR task they made use of combined values from GPS, accelerometer and microphone, processed with frequency analysis, and with classifiers like NB and HMM. The Android application disposed of an *ad hoc* library to run inference.

Not only research is interested in Activity Recognition but of course big companies that provide products and services. We are going to take a look to the most common frameworks for mobile OSs in the market.

Apple iOS Core Motion

Apple's Core Motion[1] is an Activity Recognition framework that provides the means to access already processed motion-oriented sensor data from onboard sensors. The framework also provides higher-level data from *virtual sensors* like steps and other environment-related events.

Google Android AR APIs

Google's Activity Recognition APIs for Android[2] use short bursts of sensor data collected periodically to infer user activities through pre-trained ML algorithms. The framework pro-

[1] `developer.apple.com/documentation/coremotion`.

[2] `developers.google.com/location-context/activity-recognition`.

vides both on-demand activity information and event-triggered notifications called `intents` for specifically registered requests.

It is obvious that Google's AR APIs make an easy-to-use, effortless framework for user activity, but it might actually not fit everywhere; let us take a look to some key advantages and drawbacks.

- It is an "out-of-the-box" tool, with all the *cons* of the case: there are no implementation details explicitly documented; most of the times are covered by corporate secrecy for highly valued commercial products.
- Its performances too are not explicitly stated and are hardly guessable, as they strongly depend on the target activities the developer wants to infer. Nevertheless, differences in functionality and performances across different API levels[3]are also unknown and might be significant, since the developer would not have much control over them. As of May 2018, only the 4.6 % of Android devices received the latest OS version (Android 8+ "Oreo"), and almost 40 % of the devices had a version less recent than the latest five API levels (Android 6+ "Marshmellow"), which are the ones that provide means to control application and service permissions.[4]

[3]Each version of the Android OS has a specific API level (or version). Methods may vary across different API levels, and some might not be available in more—or less—recent levels.

[4]Data as of May 2018; now second-to-last, as a new version has been released in August 2018; see Table IV.

- There is no control from the developer's perspective on the sensors and data used for the inference, and on whether the data "leaves" the device; some experiments helped deduct that the framework does make use of GPS and location data, along with other usage information.
- The framework requires that the end user grants location information to the entire OS and its depending services, that in Android fall under the name of Google Play Services; these authorizations allow a pervasive "intrusion" in the smartphone data that concerned users might not want to share, not only for privacy concerns, but also for the sake of power consumtion. We will talk again about Android permissions in Chapter 3.

As we said, Google's AR APIs performances are not of public domain. Zhong et al. [65] and us tried to infer them for common tasks, and we reached the same conclusions they did—some information is reported in Table II:

(i) Activity Recognition is characterized by a particular delay: correct recognition takes from 3 up to 30 seconds, with an average of 18 seconds.

(ii) The *stationary* class is way less precise than one might think of: it actually is particularly subject to short jerks that lead to wrong predictions even if the devices remains still. The inaccuracy is also due to the fact that the prediction is often wrong when the device is held in hand while being used, and perturbations from typing and swiping induce to misclassification when the actual class should be *stationary*. We will see in Chapter 7 that we will be able to build a model that keeps on the *stationary* class with higher

TABLE II: CONFUSION MATRIX OF MEASURED PERFORMANCE FOR GOOGLE ACTIVITY RECOGNITION API.

Class: Actual (down) v. Predicted (across)	**Stationary**	**On Foot**	**Biking**	**Vehicle**	**Unknown**
Stationary	52 %	20 %	0	10 %	18 %
Walking	0	85 %	0	0	15 %
Running	0	80 %	10 %	0	10 %
Biking	0	0	68 %	0	32 %
Vehicle	5 %	2 %	4 %	64.5 %	24.5 %

confidence while the phone is being used, with a confidence grater than 90 % threshold more for than the 80 % of the times.

(iii) The APIs provide the class *on foot* as the most confident class for both *walking* and *running*; the actual walking or running class is listed as second-most probable activity, with much lower confidence levels. Tests performed by Zhong et al. [65] report that *walking* and *running* are actually discerned correctly about 66 % and 53 % of the times, respectively.

(iv) The *vehicle* accuracy is also low because the recognition results produced while the vehicle is stopped—e.g. bus stop, traffic light—are often incorrect; the *unknown* class is usually given in its stead. This also happens fot *biking* for the same reasons, where the class *unknown* in Table II also includes results that were labeled as *tilting*.

2.2 Semantic Location Recognition

The research community has delivered plenty of results in Semantic Location (SL) to provide diverse and targeted functionality to the user. The main objective of SL recognition are Location-Based Services (LBS): in the era of data mining, all top firms providing online services have continuously improved their products using user-generated data about location: Facebook, Google, Apple, as any app on our devices, or website we visit—the reader can name it—can, and usually does track our exact location at any time, hopefully after being granted, with more or less user awareness, the proper permissions, and possibly to the extent allowed by the law (e.g. *Privacy of customer infromation* – 47 U.S. Code § 222 [2]).

Semantic Location has been extensively studied in relation to navigation; several efforts have been made, for example, in **indoor navigation** [36, 37, 9], which is a challenging task as GPS technologies are barely usable in indoor environments. Semantic Location tasks also aim at recognizing the context of the user location: context-aware systems infer the type of location to provide differentiated services (cf. Section 1.3), e.g. home, workplace, or stores [39, 48]. These works strongly rely on external data on urban environments like maps, bus routes and stops, corporate addresses, and more. This is the reason why the most omniscient and pervasive ubiquitous system now is very likely owned by the Maps division of Google, as it detains all kind of information on routes, public transportation (including, but not limited to, routes, stops and time schedules), shops, institutions, and much, much more.

The reader can now easly imagine in how many way one can define Semantic Locations, somewhat like what we said for Activity Recognition in Section 2.1. But what we really seek here is a precise semantic distinction between indoor and outdoor.[5]

2.2.1 Indoor/Outdoor Detection

Not many works focus on *bare* indoor/outdoor detection, where with *bare* we mean that we seek for a model that is not (necessarily) tied to a location description: we do not have particular interest on which type of indoor environment the user is in, rather, we are only interested in whether they are currently performing an activity inside a building or outside on the street. The research community presents two worth citing contributions to indoor detection.

The first, from Ravindranath et al. [51] and Wang et al. [61] is strictly dependent on GPS availability and signal strength. GPS availability is much less reliable in an indoor environment (cf. Section 3.3) and this inaccuracy of the system can be leveraged to infer indoor or outdoor positioning. This methods not only suffers from the various number of situations in which indoor signal might be not particularly shielded, as well as outdoor GPS precision be lower than usual in some areas; GPS is also the **most power-consuming sensor** in mobile devices, making it not suitable for tasks that might run all day long [42]. Furthermore, GPS availability on mobile devices strongly depends on the user enabling the location services and granting the relative permission to the application (recall Section 2.1.2).

[5]We are not considering here in-vehicle detection, as we will defer this task to the AR process.

We will see how we can infer indoor/outdoor positioning with way less power-consuming onboard smartphone sensors. These sensores are already constantly operating inside the device, so there is no extra cost in adoperating them. They also are not on the user to control.

The first approach to **lightweight I/O detection** comes from Li et al. [38] in 2014, a work called **IODetector** exploits only low-power sensors available on mobile devices to detect indoor-to-outdoor transitions and vice-versa. They make use of the data coming from **light sensor**, **Radio Signal Strength (RSS)** and **magnetic field sensor** (see Chapter 3 for more details on mobile sensors). This work gives us an insight on what are the possibilities in terms of mobile sensors that can be leveraged for this task, and how they commonly behave in indoor and outdoor environments—knowledge that we will embrace almost in full in our setup.

Yet, this first work approaches the location detection by analyzing the sensors behavior during transitions, and achieve not extremely high accuracies introducing a *stateful* algorithm, which implicates the dependency of a result from previous classifications.

The IODetector weakness due to its **hard-coded thresholds** for sensor values has been then overcome by Radu et al. [49] in 2014 with the **Poster** project. What they did was changing the type of problem from *triggering* to *classification*. They also extended the *sensometry* of IODetector with **surrounding sound** amplitude, **battery temperature** values, and **proximity sensor**.

Radu et al. [49] built up an articulate model combining supervised and semi-supervised learning techniques enhanced with co-training to achieve a performance of 92 %. We will leverage the knowledge on the potential of these additional indicators to build our integrated SLAR model that will lead to a more accurate result and higher noise resistance with easier training.

2.3 Combined Semantic Location and Activity Recognition

Although exploiting ad hoc wearable sensors, Raj et al. [50] in 2008 present the first—and potentially unique—other work in simultaneously detecting both the human activity and the corresponding semantic location. This work focused on both wearable sensors and GPS data to infer motion types and semantic location—indoor, outdoor, and in a vehicle, trying to keep the number of sensors contained. Working with special types of installed sensors implies that they are worn all day, all over the body, with a sufficient power supply that needs too to be carried by the user. This work runs a well-contextualized inference that includes usage of satellite images to define buildings contours for indoor and outdoor detection. The joint inference of activity, location, and trajectory is achieved with a Dynamic Bayesian Network.

2.4 Long Short-Term Memories

In the last years, a lot of work has been carried out on Long Short-Term Memories (LSTMs), a particular implementation of Recurrent Neural Network (RNN) showing unique results in time series data prediction and classification. We went through a brief overview on LSTMs in Section 1.5, and we will explain later in better detail how they work, motivated by the work of researchers in AR.

Zhu et al. [67], Liu et al. [40], Wang et al. [62], and Veeriah et al. [59] all propose literature that shows the benefits of LSTMs in time series data, for Activity Recognition as well. Zhu et al. [67] and Liu et al. [40] have studied LSTMs applications to human skeleton visual data for action recognition, reaching a 90 % accuracy with Deep LSTM models with Dropout for co-occurrences of movements of joints in skeleton data.[6,7]

Liu et al. [40] set up a spatiotemporal implementation of LSTMs for 3D skeleton data, defining a new type of **gates** for combined spatial and temporal aspects, reaching accuracies from 93 % to 97 % on different datasets.[8]

Convolutional Neural Networks (CNNs) have demonstrated an elevated potential in prediction and classification tasks, but they do incur in some limitations, especially when applied to datasets that lead to **gradient vanish**, and/or when the problem setting involves time series, sequential data. In such cases, like a video processing task, common NNs do not catch the features *evolution* over sequential time steps; Baccouche et al. [7] in 2011 comments on how NNs ignore this evolution even when temporal variations are embedded in the data through features extraction techniques like segmentation or delta records.

In their previous work in 2010, Baccouche et al. [6] show how on a video-based action recognition task, LSTMs are exploited to overcome the problem of **exponential error decay**

[6]Deep model: *commonly refers to a Neural Network (NN) with multiple hidden layer; see Section 5.3.*

[7]Dropout: *a layer in NNs that "drops" some data in a controlled way, to control bias and variance in the model; see Section 5.4.*

[8]Gate: *a building block of the LSTM cell; see Section 5.4.*

of standard RNNs, and their particular appropriateness for time series data for their ability to maintain and evaluate the context for more than just one single record.

These works mark a track for our, leading us towards the choise of applying LSTMs for sensor-generated, time-series data to conjunctively infer Semantic Location and Human Activities.

CHAPTER 3

MOBILE SENSORS OVERVIEW

In some way, today's smartphones are way beyond computers. When we introduced the concept of *context-awareness* in Section 1.1, we saw that mobile devices sense the environment around them with more senses than we do: light, radio waves, temperature, sound, vision, movement, orientation; at this point, they also understand human speech and conversations.

The choice of the word *senses* is not random: smartphones achieve these goals by means of **sensors**, Micro-Electro-Mechanical Systems (MEMS) of very small sizes inside the device. Sensors are fundamental to allow a comfortable user experience, and to give applications the opportunity of providing insightful services. Some of these sensors have more hidden functionalities, like the magnetic field sensor; others, we interact more directly with, like a camera or a microphone [43]. For all the notions in this Chapter, except where otherwise cited, please refer to the Android Developer Guides.[1]

Android sensors provide high-precision measurements at an high rate and with averagely low power consumption. High-precision sensors allow developer to write applications of various levels of complexity, including high quality games that make use of the device movements instantly. Table III reports a list of the sensor types currently **supported** by the

[1]Android Developer Guide Sensor Overview available at: `developer.android.com/guide/topics/sensors/sensors_overview`.

Android platform (as of June 2017)—supported does not imply that all Android devices board them, see below.

Android sensors are *virtual devices* providing the raw data from the actual physical sensors that the device boards, and are divided in three main categories:

motion capture movements, mainly acceleration and rotation;

environment capture information about the surrounding environment, like light, pressure, and temperature;

position describe the physical position of the devices; includes orientation and magnetic field.

As for the implementation, we also have two distinct types of sensors:

hardware-based are physical components of the device that directly measure physical quantities, e.g. acceleration;

software-based also called virtual or synthetic sensors, are not actual components but they are an abstract representation of a sensors that in truth provide an elaboration of the data coming from hardware-based sensors; e.g. gravity, which is deduced from acceleration.

Not all devices share the same set of sensors, and not all smartphone manufacturers mount the same sensor make and build on their devices. The Android Sensor Framework (see Chapter 4) allows an application to dynamically query for all available sensor on a device, their manufacturer, values range, resolution and power requirements.

TABLE III: LIST OF SENSOR TYPES SUPPORTED BY THE ANDROID PLATFORM.

Sensor	Type	Description	Unit of measure	Available since (API Level)
Accelerometer	Hardware	Acceleration force applied to the device on three axes, incuding gravity.	m/s^2	3
Ambient temperature	Hardware	Measures the ambient room temperature.	°C	14
Gravity	Device dependent	Gravity force only applied to the device on three axes.	m/s^2	9
Gyroscope	Hardware	Rate of rotation of the device around three axes.	rad/s	9
Light	Hardware	Ambient illumination level.	lx	3
Linear acceleration	Device dependent	Acceleration force applied to the device on three axes, excluding gravity.	m/s^2	9
Magnetic field	Hardware	Ambient geomagnetic field along three axes.	µT	3
Orientation	Software	Degree of rotation of the device around three axes.	° (deg)	3
Pressure	Hardware	Ambient air pressure.	mbar	9
Proximity	Hardware	Proximity of an objec to the device screen.	cm	3
Relative humidity	Hardware	Relative ambient humidity.	%	14

We will begin now an overview of the sensors that are more of interest for this work. We will see in better detail how they work, for what task they can be useful, the use they have had in related literature, and whether we are going to make use of them with the proper motivations.

3.1 Introduction to motion sensors

Orientation and motion sensors, also called *inertial* sensors, are strictly related to the concept of **coordinate system**. To understand the value readings of these sensors, we need to relate them with the correct coordinate system. There are indeed two different coordinate systems, both represented by three axes x, y, and z, defined as follows.

Global coordinate system Is the coordinate system of the Earth, the one we are more familiar with. The x axis points towards East, normal to the true North, and tangent to the Earth's surface; the y axis points towards the magnetic North, that is approximately the true North; the z axis points towards the atmosphere, normal to the Earth's surface.

Device orientation system Is the non-inertial coordinate system of the device.[2] Smartphones' orientation system is defined with the smartphone laying on a table, in portrait (i.e. vertical) mode, with the screen pointing upwards. The x axis is pointing to the

[2]Non-inertial coordinate system: *or* non-inertial frame of reference, *is a coordinate system whose physics vary depending on a force (i.e. acceleration) with respect to an inertial one. An inertial frame of reference is either stationary or moves of a constant speed along a straight line with respect to another inertial one, and has no fictitious forces (e.g. centrifugal effect).*

right of the device horizontally; the y axis is pointing out horizontally upwards from the top of the device; the z axis is pointing out the screen towards the sky.

3.2 Accelerometer and Gyroscope

An **accelerometer** is a devices that measures the *proper acceleration* of a body in its coordinate system—in this case, the device's—which is not a fixed coordinate system, instead the body's frame of reference defined by its instantaneous rest position. This means that an accelerometer resting on the Earth's surface would only measure the force of gravity $g \approx 9.81\,\mathrm{m/s^2}$, along a vertical axis pointing towards the center of the Earth (as in Figure 4 (a)), and 0 along all axes when in free fall towards the center of the Earth. Ideally, an accelerometer is composed of a mass connected to springs. When the mass is subject of acceleration it moves from its rest position, and the distance from the original point along the three axes gives the acceleration.

A **gyroscope** is a device that measures orientation and angular velocity. A common gyroscope looks like a disk free of rotating along any axis, but MEMS gyroscopes are usually implemented with other technologies, sometimes similar to the accelerometer explained above. The gyroscope mass is usually vibrating along one axis and, when the device is rotated, it moves along a different trajectory, based on the theory behind the Foucault pendulum, under the action of the *Coriolis effect*. The variation is sensed by capacitors producing different electrical intensities. The angular velocity increases positively when rotating along an axis, according to the *right-hand rule*. When the device is steady, the gyroscope should measure approximately 0 along all dimensions, like in Figure 4 (b). This type of gyroscope is called *vibrating structure gyroscope*, and standardized for consumer electronics by the IEEE under the name of Coriolis Vibratory Gyroscope (CVG).

Modern accelerometers and gyroscopes embedded in small devices are MEMS and are basically always present in smartphones and tablets. Their main function is to detect portrait and landscape mode, but their applications are countless, including complex tasks like games and camera image stabilization. Higher level chips or applications can process accelerometer data to provide enhanced information like steps or tilting.

Accelerometers and gyroscopes are exploited in basically any sensor-based Activity Recognition setup. As anticipated in Section 2.1, literature presents mostly works with ad-hoc sensors strapped to the human body [14, 52, 64, 45, 66], but some work uses mobile devices' accelerometer sensors as well [16, 35]. They are an obvious choice for AR on mobile, constitute trivial detectors of *still* and *motion*, and produce **distinct intensity values and patterns** for different activities. MEMS accelerometer and gyroscope are both lightweight sensors, with **low power consumption**, and produce data continuously.

Figures 1 to 5 report an example plot of accelerometer and gyroscope time series collected during the five different activities. Accelerometer and gyroscope respectively have always the same axis scale for sake of ease of comparison. All plots show clear distinct patterns for each activity. As one can expect, *stationary* condition is the most straight of all others (Figure 4), but it is not difficult to spot significant differences in values oscillations in both frequency and amplitude among the different activities: *running* (Figure 3 (a)) shows a more chaotic pattern and larger values in amplitude than *walking* (Figure 5 (a)), while the chart for *in vehicle* shows smoother transitions (Figure 2 (a)).

Correction of accelerometer coordinate system

The main concern about accelerometers is that the acceleration values are in the device's coordinate system (Section 3.1), thus values are tight to the device orientation, biasing the classification. The AR training process can be "fooled" by the device orientation during data collection, a problem that has been pointed out by previous work in AR on mobile devices [57].

Android Sensor APIs provide means to correct the coordinate system. The developer has to compute a **rotation matrix** R derived from instantaneous accelerometer and magnetic sensor data—magnetic field data carry information on the device's orientation with respect to the Earth coordinates; see Section 3.4. Once we have the rotation matrix, we can compute the true acceleration as:

$$a^i_{\text{true}} = R_i \times a^i_{\text{device}}\,, \tag{3.1}$$

where a^i_{true} will be the acceleration in Earth's inertial frame of reference at time instant i, R_i is the rotation matrix derived at time i, and a^i_{device} is the acceleration sensed at time i. Both the accelerations a are arrays of three elements containing the three-axial values of acceleration, in the form $[a_x, a_y, a_z]$, thus R is a 3×3 matrix.

Ideally, once the accelerometer values have been corrected according to the Earth's frame of reference, the acceleration by the gravity g acts mostly on the z axis; in fact, we can observe in Figures 1 (a) to 5 (a) that z axis values usually unfold around the value of g. This can be perceived precisely from Figure 4 (a).

3.3 Positioning Systems

While still in the aspect of motion, the other smartphone module that is natural to think about is the GPS. As seen in Section 2.2, previous indoor/outdoor detection and activity recognition systems were GPS-based, as it is a state-of-the-art, extensively developed and perfected technology in positioning.

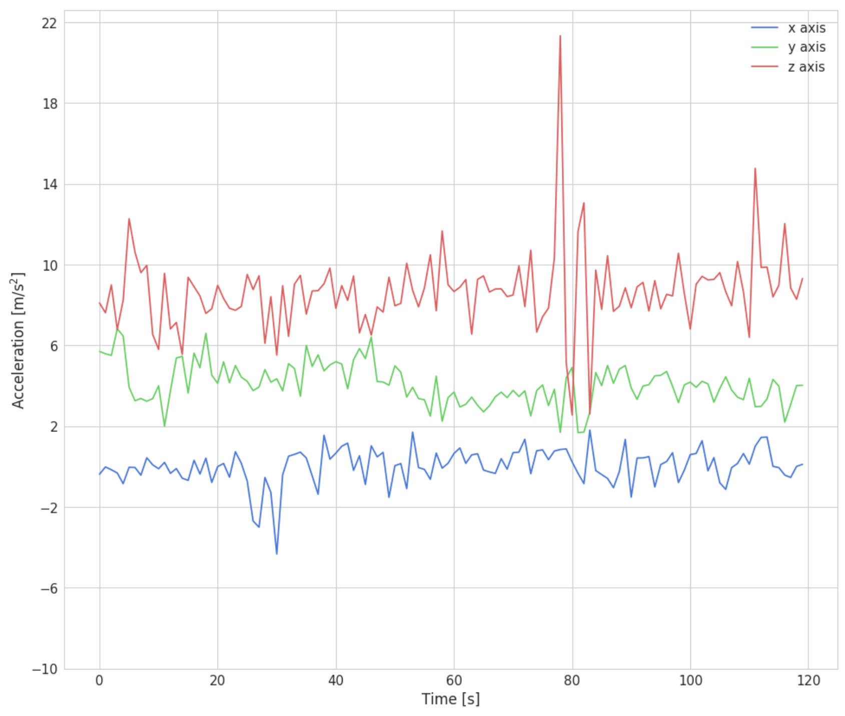
The Global Positioning System was born as a project of the U.S. Army and is currently under control of the United States Government, although similar technologies exist worldwide; as of December 2016, there are other two Global Navigation Satellite Systems (GNSSs), the Russian GLONASS and the European Union's "Galileo".

For its original military purposes, GPS has been developed with high precision standards and liability, and has today accuracies to the order of microseconds and millimeters.[3] Civilian accessible technologies are declared to be accurate to 4 meters RMS by the U.S. Government.[4]

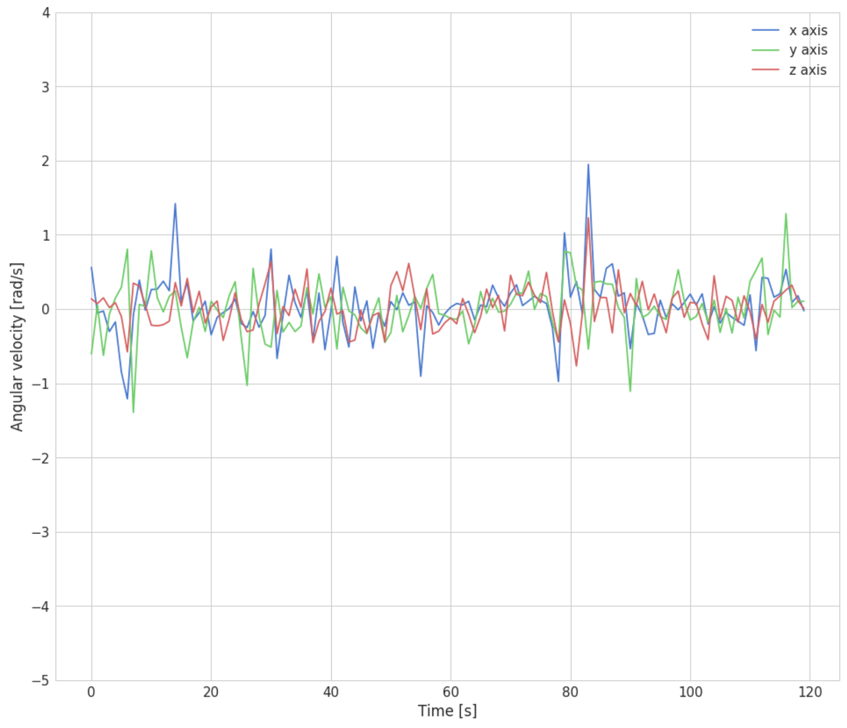
Today's smartphones are equipped with GPS technologies for navigation features. This module not only provides the device position in the world, but also its speed. To obtain the device speed and location, Android Location Services offer two location providers with different granularity.

[3]From: `www.gps.gov`, visited June 2018.

[4]RMS: *Root Mean Square, a measure of the magnitude of the variation of a quantity, a.k.a.* quadratic mean.

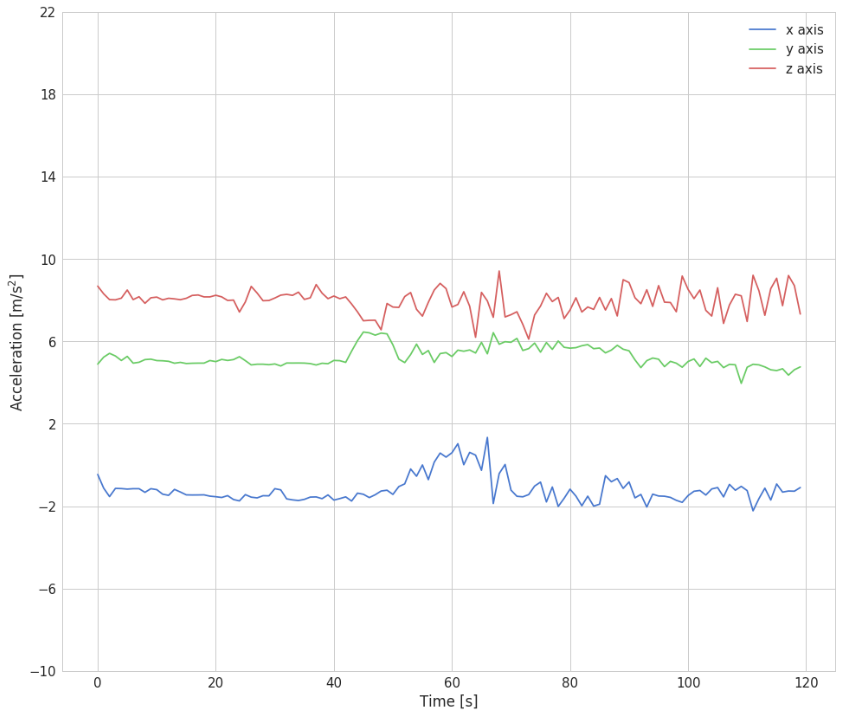


(a) Accelerometer data for activity *biking*.

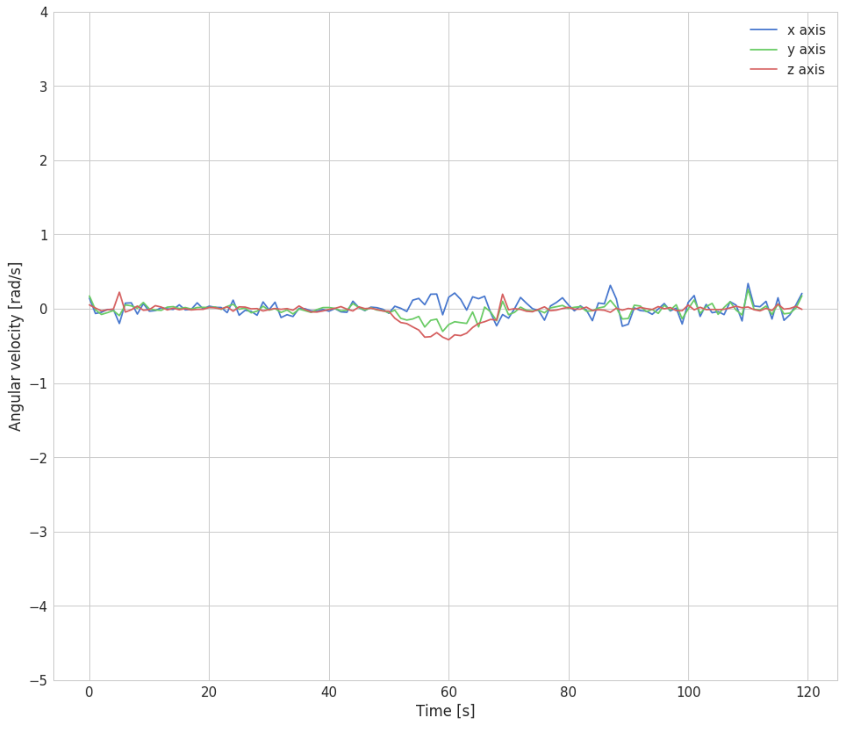


(b) Gyroscope data for activity *biking*.

Figure 1: Motion sensor data for activity *biking*.

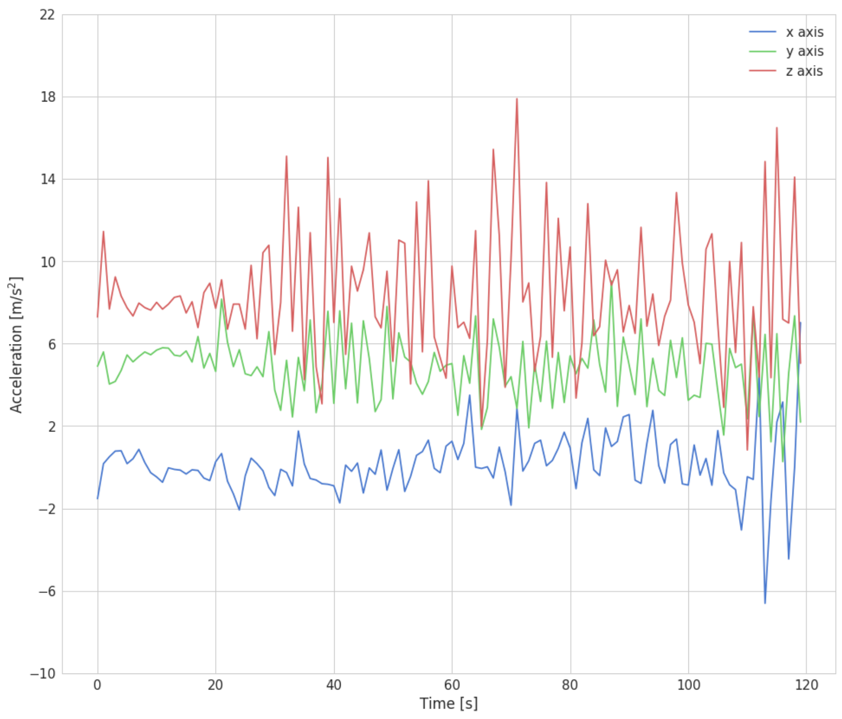


(a) Accelerometer data *in vehicle*.

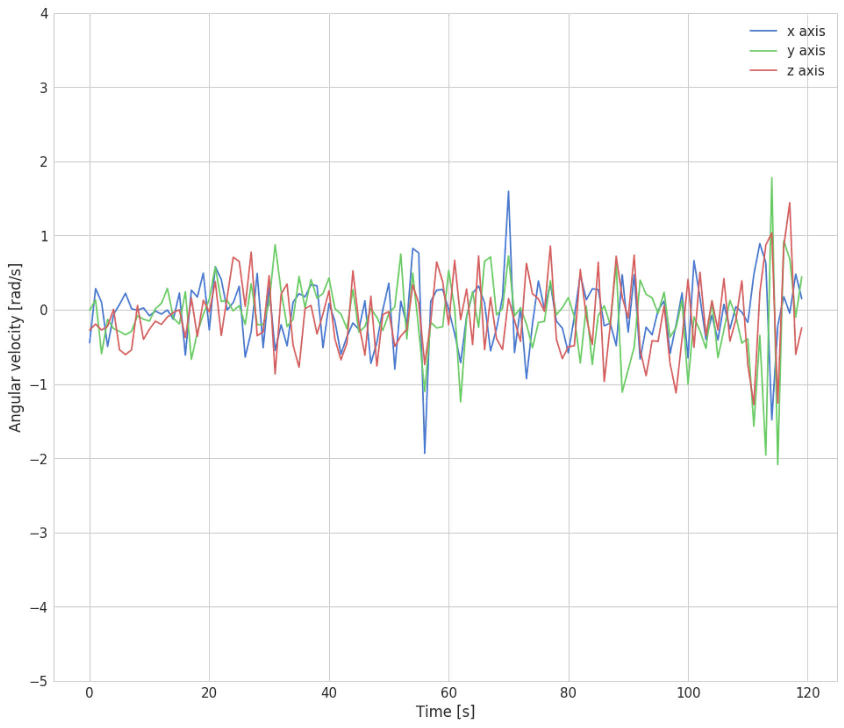


(b) Gyroscope data *in vehicle*.

Figure 2: Motion sensor data *in vehicle*.

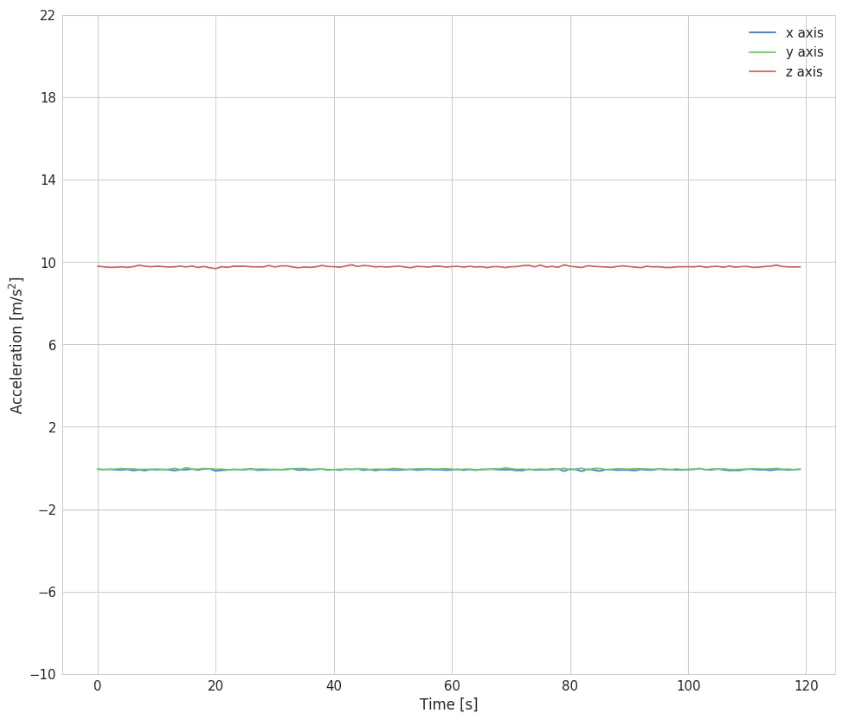


(a) Accelerometer data for activity *running*.

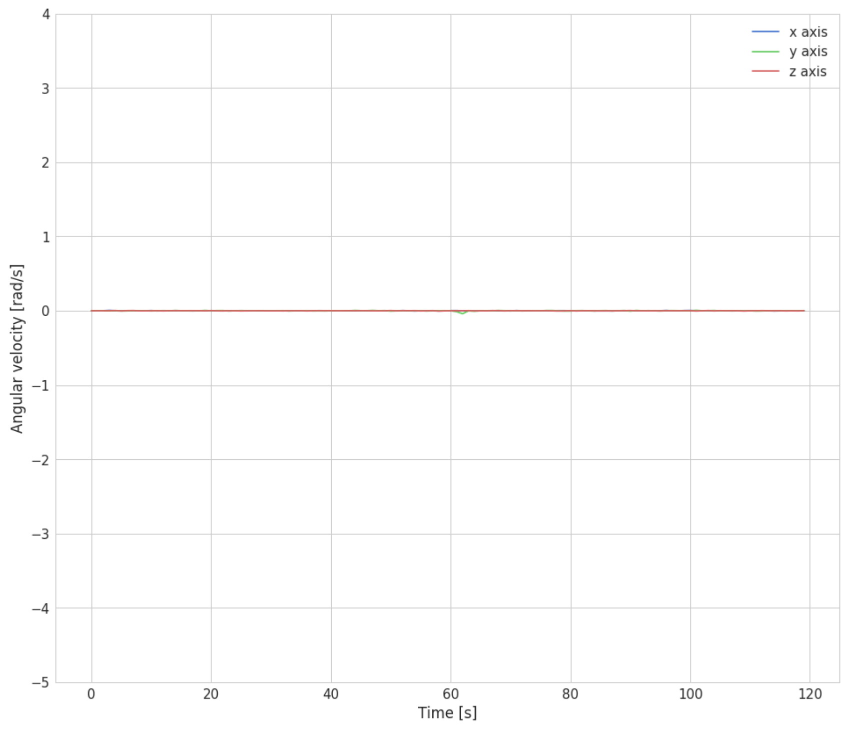


(b) Gyroscope data for activity *running*.

Figure 3: Motion sensor data for activity *running*.

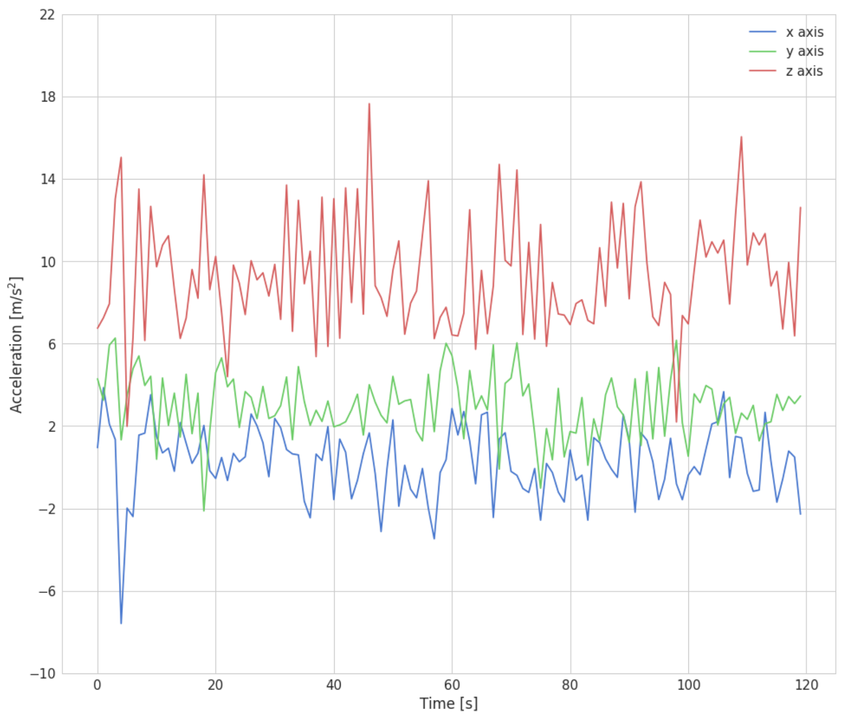


(a) Accelerometer data for activity *stationary*.

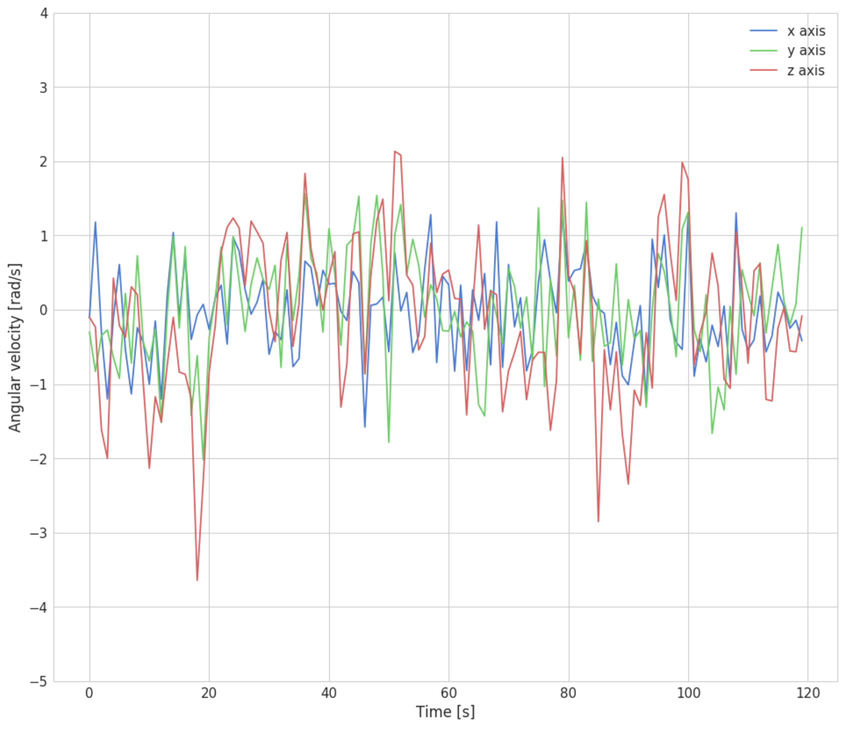


(b) Gyroscope data for activity *stationary*.

Figure 4: Motion sensor data for activity *stationary*.



(a) Accelerometer data for activity *walking*.



(b) Gyroscope data for activity *walking*.

Figure 5: Motion sensor data for activity *walking*.

The *coarse* one is the **Network**, that provides positioning and speed based on the time spent on moving from one operator cellular tower to another, based on the information from the network service provider. This kind of information is rarely used because is extremely low accurate: it is indeed meaningful only when the device moves among the radius of operation of different tower cells, i.e. for tens to hundreds of meters, otherwise the reported speed is always zero, as sustained by collected data.

More accurate, or *fine-grained* location is provided by the GPS. Speed is inferred by the rate of change of position, which is derived from satellite triangulation, at the bases of the GPS technology. We have said before that GPS is highly accurate, but this is an half truth. Although there probably is no better positioning system outdoors, GPS suffers a lot indoors, when the satellites are not in a *"line-of-sight"* with the device: buildings' concrete and electronic appliances shield and interfere with the GPS signal. From the data collected for this work, GPS-generated information is available for less than the 50 % of the time spent indoors, and even when the GPS signal is available, the reported speed is almost always zero even when the device is moving significantly. Thus, the Location Services provided speed does not make a reliable indicator for our model.

We have seen previously in Section 2.2.1 that works like the one by Radu et al. [49] have exploited the uncertainty from GPS data to infer indoor or outdoor location. The other side of the medal is that using the phone's GPS with no insightful information is not always worth in terms of efficiency.

The GPS is an extremely **power consuming module**—with good confidence, we can say it is the most power consuming component in the device, let alone the screen. This power inefficiency is due to the fact that the satellite communication channel has high latency, and this slow communication costs energy, especially because for the position to be determined the phone has to "talk" to three different satellites. On Android, the GPS also has the drawback of preventing the phone to go in *idle* state when not in use [42]. The GPS running all day long on a device would very likely be draining the battery of a common smartphone in a bunch of hours. These notions are usually known to the average user that suffers from its smartphone's short battery life. We will not make use of the speed our model as this will lift us from graving on the phone's battery, whilst the model will suffer from less than a 0.5 % in accuracy overall.

3.4 Geomagnetic Sensor

We pointed out in Section 2.2.1 that we would be taking most of our predictors for indoor/outdoor detection from the works conducted by Li et al. [38] first and Radu et al. [49] later. The first indicator we analyze is the **geomagnetic field**.

A MEMS magnetic field sensor is small *magnetometer*. They ideally operate measuring a physical phenomenon called **Lorentz force**, a combination of both the magnetic and electric field that affect a charge. When a charge moves at non-null speed in the presence of both a magnetic and an electric field, it is subject to said force, proportional to its speed and the intensities of both fields.

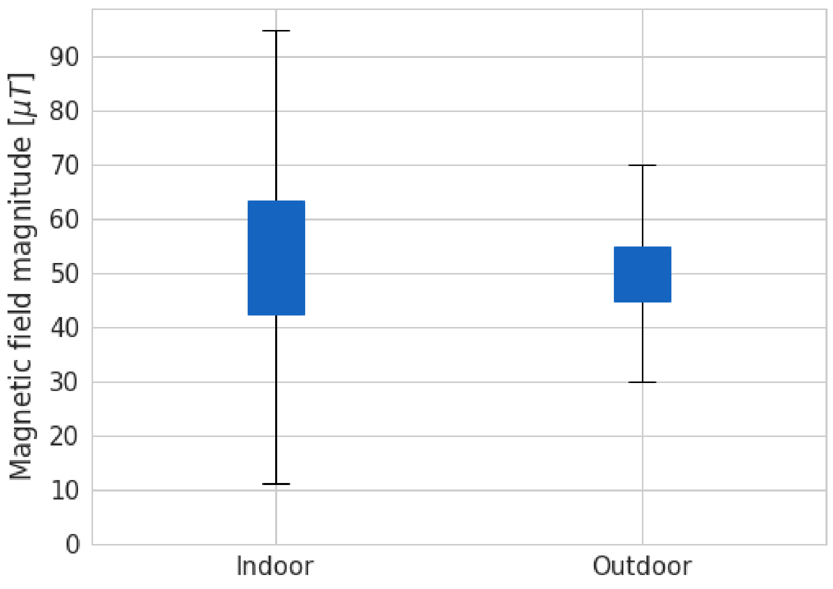


Figure 6: Value distribution for geomagnetic field magnitude.

MEMS magnetic field sensors are present on mobile devices to implement a **compass** and support navigation. They measure the ambient magnetic field intensity due to the perceived Earth's magnetic field plus disturbances, along three axes, in microTeslas (μT).

Measuring the intensity of the surrounding magnetic field can make a good indicator for indoor/outdoor detection. It is in fact the disturbances that the device senses that can help us understand if the device is indoors. When we are surrounded by steel and concrete building skeletons, and electrical appliances, the Earth's magnetic field that the sensor measure vary. This disturbance in indoor environments can be as intense as to be enough, in some proper setting, to define an actual "*fingerprint*" of the place [15].

Magnetic field is not a constant, not even among "undistorted" outdoor environments around the globe—e.g., magnetic field at the equator and at the poles differ, but the distortion introduced by proximity to electric appliances inside our buildings can be as strong as more than one order of magnitude [38]. Since we are not interested in the direction of these disturbances, and for sake of dimensionality, we take in as a descriptor only its magnitude, or L^2-norm, as follows:

$$|B| = \sqrt{B_x^2 + B_y^2 + B_z^2}\,. \tag{3.2}$$

The boxplots in Figure 6 show how the magnetic field magnitude in indoor environments has a wider range of values with respect to outdoor.

3.5 Cellular Radio

Li et al. [38] suggest that the phone's cellular Radio Signal Strength (RSS) is an interesting indicator to exploit for indoor/outdoor detection. Although this is not properly a sensor, the cellular radio module provides costless information, as it is inevitably always running. The signal strength information comes indeed from the cellular antenna that provides the basic connectivity functionalities: calls, texts and internet access. It actually is not an extremely power-efficient module, as long-range radio connectivity requires some amount of power to provide a good service, yet reading its related information does not affect the power consumption it would have either way.

The RSS can be exploited in a similar way as for geomagnetic field; it is in fact leveraging its limitations that we can infer indoor and outdoor positioning. When we switch context entering a building, the RSS drops, for the same reasons for which the magnetic field gets

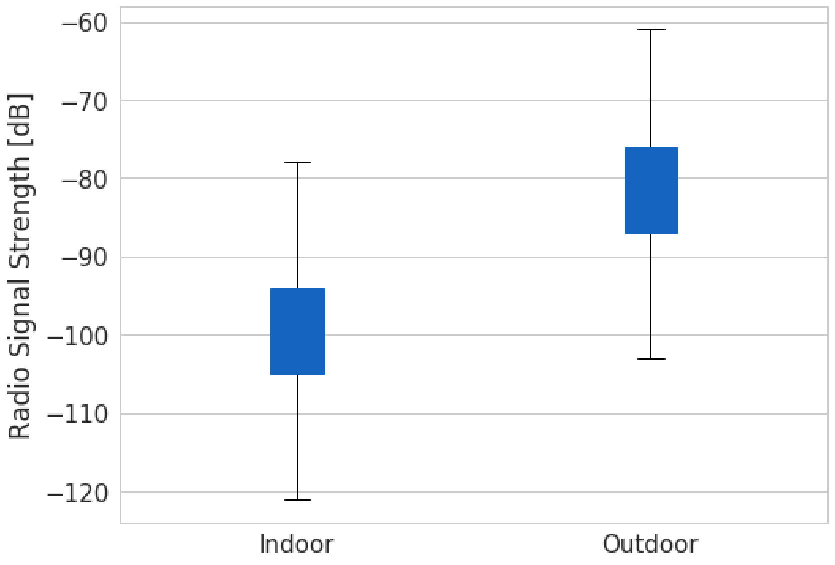


Figure 7: Radio Signal Strength (RSS) indoor attenuation in decibel.

disturbed: concrete, steel and windows of the building shield the signal, and electronic appliances interfere with it—e.g. television and other radio transmissions, cordless landline phones.

The boxplots in Figure 7 show the values of RSS in indoor and outdoor. The signal attenuation median value, in decibel (dB), is higher (in magnitude) of around 20 dB when indoors.

Of course the RSS attenuation is not a perfect indicator only by itself; as shown by Chung et al. [15] and Radu et al. [49], it suffers from outliers generated by different “anomalous” conditions: the cell tower availability depends on the location; rural areas as well as highly crowded urban areas suffer a less strong signal even outdoors. Weather condition too influences the signal strength. A device can also have good connectivity when inside but near an open window; finally, connectivity also somehow depends on the device's antenna technology.

3.6 Light and Proximity Sensors

An **ambient light sensor** is an always present component of smartphones and some other type of display-equipped technology; it is indeed mainly exploited to adjust the screen brightness level according to the ambient light. Light sensors are implemented with *photodiodes* or *phototransistors*, and are extremely power efficient. An ambient light sensors reading is the light intensity perceived in luminance (lx).

A **proximity sensor** can detect the presence of an object in accordance to its proximity. The distance is usually measured with the emission of an electromagnetic field or infrared rays. The perturbation of the field, or the difference in time for the mirrored radiation to return back at the sensor, is translated into distance. Proximity sensors have many applications, from parking sensors to automatic gates. On smartphones, the proximity sensor is usually in charge of detecting whether the phone is in a pocket, or held to the ear during a phone call, thus to prevent unwanted taps on the screen. A common smartphone proximity sensor reading can either be a distance (in millimeters or centimeters) or one of a range of values (with a maximum and a minimum value, and at least two—"far" and "near"), depending on the manufacturer.

Ambient light and proximity sensors provide very straightforward examples of context-related information. Li et al. [38] suggest that ambient light is a very strong descriptor for indoor/outdoor detection. The ambient light reading is crucial because indoor environments are very differently lit with respect to outdoors—this is, after we discriminate according to the time of day.

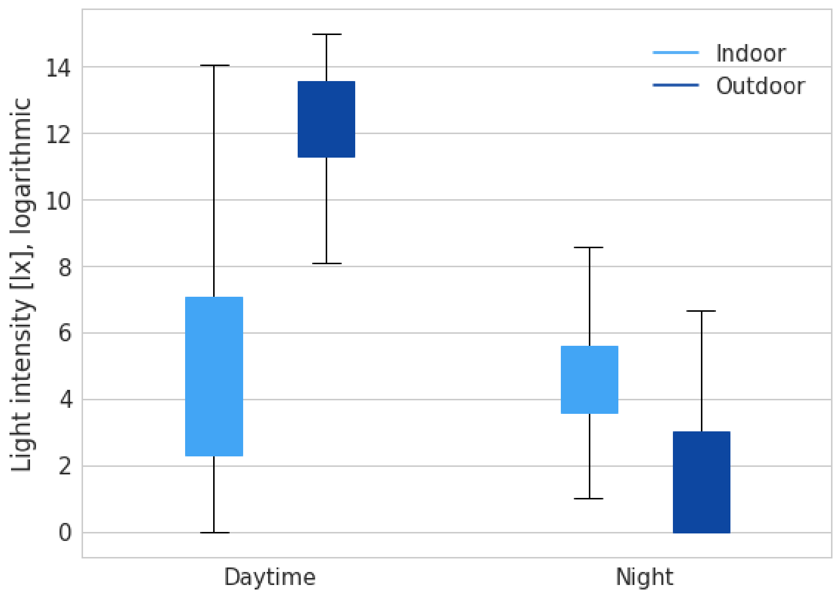


Figure 8: Ambient light indoor and outdoor during daytime and night.

In **daytime**, indoor ambients have way less luminance than outdoor; this is true even if the room seems more *lit* than outside, because the light intensity is different: sunlight frequencies generate higher luminance than light bulbs [38], thus a cloudy sky will still produce an higher luminance than a well-lit room, usually hundreds of times higher. Moreover, the sensed ambient luminance level is almost independent on whether the sensor is pointed at the light source [38]. On the left hand side of Figure 8, the two boxlopts show how outdoor light in daytime is of several orders of magnitude brighter than indoor—note that the luminance scale is logarithmic.

At **night**, everything is reversed: outside the darkness produces light intensity values inferior to the unit, while the house is usually lit. This of course arises one of the concerns about ambient light as an indicator: for the most part of the time in which we are asleep, with the lights shut, indoor ambient light is usually the same than outdoors—sometimes it can even be lower, e.g. when the room has blinds while outdoors there is significant light pollution. We can see from Figure 8 how night ambient light values outdoors drop below the values from indoor at night.

The other concern arises from our tendency of keeping the phone in a **pocket** or bag. There the light intensity is of course null independently of ambient light. Here is where the proximity sensor comes at hand. The approach is to detect if the light sensor is covered through the proximity sensor—they are one next to the other on smartphones. When the proximity sensor tells that the device is being covered, the last value of ambient light is held without being updated, until a new value is produced with the device front uncovered.

3.7 Microphone

The last descriptor we introduce is once again suggested by Radu et al. [49] and is again not properly a sensor. The **microphone** is a fundamental component in each telecommunication device, as it is necessary to record audio during phone calls. It is today used to provide also more complex functionalities, the most notable being voice input for commands (for both accessibility and *hands-free* usage) and personal assistant interaction. This makes the microphone a *de facto* environment sensor, as it provides insightful information about the surrounding; it is basically the phone's ear.

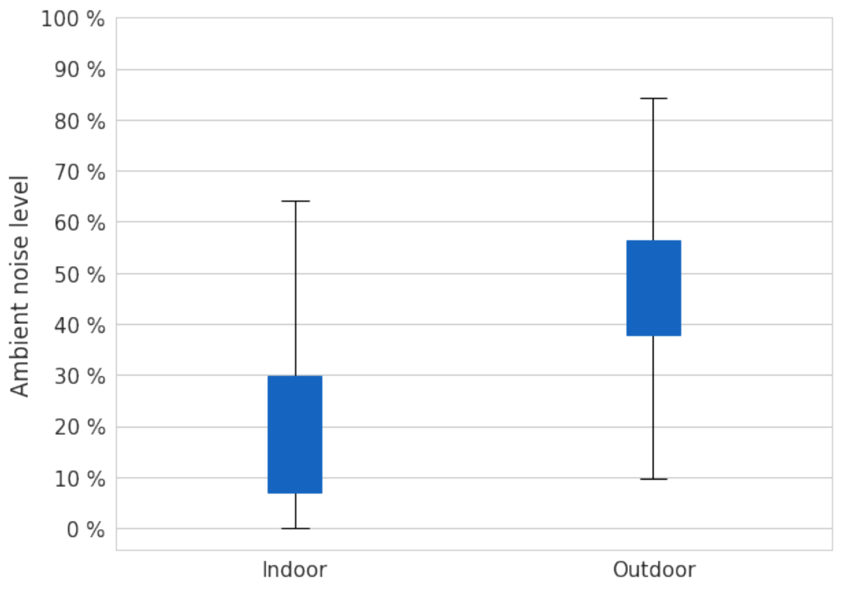


Figure 9: Ambient noise levels for indoor and outdoor.

The reason why the microphone is a useful descriptor for us is because we can exploit the ambient noise level to distinguish between indoor and outdoor. Outside the ambient noise is, on average, much higher: there is traffic, wind, construction works, chattering. Indoor environments do have noise, but is usually less intense and less constant [49]. Figure 9 shows the distribution of noise levels captured indoors and outdoors.

Alas, the microphone per se is not a particularly low-power sensor, nor it is always enabled—although newer smartphones do have always-running microphone-enabled speech recognition, used to "wake up" the personal assistant; this feature is usually supported by a low-power co-processor to minimize the power consumption. Nevertheless, we do not need to be worried about the microphone power usage, as we do not need to record audio segments. What we need instead is just an instantaneous burst of microphone feedback to capture the maximum amplitude, and we do not need to process nor save the audio file.

CHAPTER 4

DATA COLLECTION

Before moving on creating the SLAR model we need some data. We have gone through all the indicators we intend to use in Chapter 3. As we have mentioned, smartphones are equiped with all the sensors we need, but each manufacturer mounts on their devices sensors of different make and models, but this is not all. Different OSs provide different means of reading sensor values, resulting in very heterogeneous data.

Since our target are Android devices, we need data produed by smartphones running such system. For a wider application, one might also want to analyze the diversity of sensor data among different Android devices as well. The first step then is to collect "first-hand" sensor readings in all the scenarios we listed in Section 1.7.

The data collection process requires the development of an Android application, as we want to read these values in the same way the final application will when running the inference. We need then to start building the "scheleton" of this application in the first place. Before going into the details of how all data is collected, we might want to go through some basic knowledge of the Android system and development first.

4.1 Introduction to Android

Android is a mobile Operating System (OS) ideated by the former Android Inc., a company born in Palo Alto, CA in October 2003 and acquired by Google Inc. in 2005, that is now con-

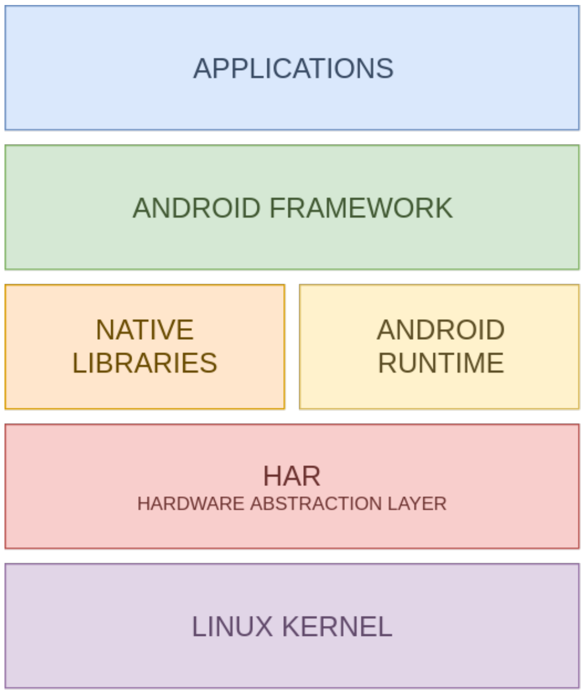


Figure 10: Android Open Source Project stack.

tinuing the development. The system is on top of the **Linux kernel** (see Figure 10), and was born with the expectation of being **flexible** and **upgradeable** [11].[1] It is primarly designed for touchscreen-equipped devices like smartphones and tablets. Today, this category extends, but is not limited, to televisions (Android TV), cars (Android Auto), and wearable technologies like smartwatches (Wear OS by Google, formerly Android Wear). In September 2008, HTC delivers the first Android smartphone as we know them today.

[1]Figure 10 data from: `source.android.com`, visited July 2018.

The promise of a fully upgradeable system as been held, through several major distribution upgrades; today Android's latest stable release is 9.0 "Pie", freshly released on August 5, 2018 for Google's "homemade" devices Pixels, and new major upgrades are usually delivered in the third quarter of each year.

Table IV reports an overview of all delivered Android major releases, codenamed, with their API level and market share among all Android devices.[2] This information is precious for developers; as we will explain shortly, it is fundamental for developers to set a target API level when developing an application, in order to reach the desired share of devices.

The shares reported in Table IV may not be much significant without talking real numbers. In its annual keynote held in May 2017, Google announced that the number of monthly active Android devices—including, but not limited to, smartphones, tablets, TVs, cars and watches—went **over 2 billion** units [46], outselling all competitors every year since 2012. As of 2015, Android was **the most installed of all operating systems**, including desktops, outselling all Windows, Mac OS X (desktop) and iOS (mobile) combined, and in 2017 took away from Microsoft Windows the title of most used OS for internet browsing.

[2]Table IV data from: `developers.android.com/about/dashboards`, visited July 2018.

TABLE IV: LIST OF ANDROID MAJOR RELEASES.

Version	**Code name**	**Release date**	**API level**	**Market share**
2.3	Gingerbread	Q1/2011	10	0.3 %
4.0	Ice Cream Sandwich	Q4/2011	15	0.4 %
4.1	Jelly Bean	Q2/2012	16	1.5 %
4.2		Q4/2012	17	2.2 %
4.3		Q3/2013	18	0.6 %
4.4	KitKat	Q4/2015	19	10.3 %
5.0	Lollipop	Q3/2014	21	4.8 %
5.1		Q1/2015	22	17.6 %
6.0	Marshmellow	Q4/2015	23	25.5 %
7.0	Nougat	Q3/2016	24	22.9 %
7.1		Q4/2016	25	8.2 %
8.0	Oreo	Q3/2017	26	4.9 %
8.1		Q4/2017	27	0.8 %
9.0	Pie	Q3/2018	28	N/A

4.2 Android application development

Like most of the competing mobile OSs, Android is based on running applications (or more commonly *apps* for short) available through an official repository called Google Play Store. In February 2017, the Play Store contained more than 2.7 million applications, for a total number of downloads of more than 65 billion already in May 2016.

Applications are intended to extend the basic functionalities of the device. Mobile telephones, old models that did basically only provide radio telephone and text functionalities, were no more than special-purpose devices; now, with the infinite scopes that apps can have, we can definitely affirm that a smartphone is no less a general-purpose device than computers.

Application development tools are delivered by the Android developers team through the **Android Software Development Kit (SDK)**, intended to extend the Java programming language—sometimes with C/C++ support—so it can be used to develop application that Android systems can run. The basic native Java and Android libraries, together with more particular runtime libraries, form the third level in the stack in Figure 10. Android programming language support has been recently extended from Java and C++ to **Kotlin**, a programming language developed by JetBrains. In 2015, Google and IntelliJ delivered Android Studio, an Integrated Development Environment (IDE) exclusively for Android development.

New SDK releases usually comes with a new API level, as shown in Table IV. We said earlier that this is relevant to the development: old features are improved or replaced with new ones, usually perfectioned in performances and security, and new features are introduced, even in the aspect of sensor support, as shown in Table III, Chapter 3.

4.3 Reading sensor data

Chapter 3 provides a thorough overview on Android sensors, specifically all the ones that we intend to exploit to generate our feature set. We have seen how android sensors are divided into three main categories: *motion*, *environment*, and *position*; we also saw that some of them are *hardware-based* and others are *software-based*.

The Android sensor framework provides different means of accessing sensor values. The framework is based on the `android.hardware` package that provides the APIs to manage each sensor. A `SensorManager` class is used to extract instances of `Sensor` objects representing different sensors available on the device. The `Sensor` class provides not only the sensor readings, but all type of information related to the sensor—e.g. make, model, type, values range. The following sample code is an example of how to create an instance of the accelerometer sensor.

```
private SensorManager sensorManager;
sensorManager = getSystemService(Context.SENSOR_SERVICE);

private Sensor sensor;
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
if(sensor == null)
  System.err.println("Accelerometer not present on this device");
```

For most of the sensors, the data collection methodology is the following. The developer creates an instance of the `SensorEventListener` interface; this object allows to *register* a new sensor *listener* for a specific sensor instance. Once a listener is registered for a sensor, two methods are invoked on an *event* basis: `onAccuracyChanged` and `onSensorChanged`, when the sensor accuracy and values change, respectively. The following code snap shows a common implementation of the `SensorEventListener` interface.

```
public class SensorActivity
        extends Activity
        implements SensorEventListener {
  private SensorManager sensorManager;
  private Sensor sensor;

  public final void onCreate(Bundle savedInstance) {
    super.onCreate(savedInstance);
    sensorManager = getSystemService(Context.SENSOR_SERVICE);
    sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    sensorManager.registerListener(this, sensor);
  }

  public static void onAccuracyChanged(Sensor sensor, int accuracy) {}

  public static void onSensorChanged(SensorEvent event) {
    float accelX = event.values[0];
    float accelY = event.values[1];
    float accelZ = event.values[2];
  }
}
```

This approach applies to all sensors whose values can only be accessed by event-woken methods; from our set, we have: accelerometer, gyroscope, geomagnetic sensor, light sensor, and proximity sensor.

An Android application should declare all the hardware feature it is using, to be compliant with the most recent Google Play Store security policies. Hardware features are declared in a AndroidManifest.xml file as follows.

```
<uses-feature android:name = "android.hardware.sensor.accelerometer"
              android:required = "true" />
```

The data from a `SensorEventListener` class cannot be retrieved on demand, instead the values must be collected and, if required, stored whenever it is "produced" by the framework. Since we want to take a *snap* of all sensor values at once at a given point in time—e.g. every 250 ms—we want to produce a record with the most recent sensor readings at that time. To achieve this, we set up a thread synchronization scheme. A system `Task` thread starts periodically at a given interval, acquiring one or more locks protecting the variables holding sensor values. The thread then builds up a record with all data, a timestamp and some other details; it then writes the record on a file in system storage—and in case on the screen too. Finally, the thread releases the locks.

When a `SensorEvent` is triggered by a sensor, the `onSensorChanged()` method is invoked and passed the `SensorEvent` object. The method checks for which sensor has triggered the event with the `event.sensor.getType()` method, acquires the lock protecting the related variable(s), and stores the newly produced value(s) for such sensor; then releases the lock. The following code is a real example on how this is achieved.

Recall from Section 3.6 how we intend to update the ambient light value only when the phone is not in a pocket or bag through the information from the proximity sensor; we use the following code to show how this other goal is achieved as well.

Note that `event.values` is a `float[]` array of at least three values. One-valued sensor data are stored in `event.values[0]`, three-axial sensor data are stored in the first three element as shown in the previous code snap.

```
private ReentrantLock lightLock = new ReentrantLock(true);
private ReentrantLock proximityLock = new ReentrantLock(true);
private float lightValue;
private boolean inPocketMode;
/* ... */
public void onSensorChanged(SensorEvent event) {
  synchronized (this) {
    if (event.sensor.getType == Sensor.TYPE_LIGHT) {
      if (lightLock.tryLock()) {
        /* only update if out or pocket and not on call */
        if (proximityLock.tryLock()) {
          if (!inPocketMode)
            lightValue = event.values[0];
          proximityLock.unlock();
        }
        lightLock.unlock();
      }
    }
    /* manage other sensors ... */
    }
  }
```

Other data instead can simply be accessed *on demand*. To retrieve the Radio Signal Strength in decibel, the Android framework provides a `TelephonyManager` class to access details about the cell phone radio. The class provides the method `getAllCellInfo()` to retrieve a `List<CellInfo>` of all the supported cell types—GSM, LTE, CDMA, and WCDMA. Under normal circumstances, the phone is connected to one single cell of a precise type, that can be identified with the `isRegistered()` method. The `CellInfo.getCellSignalStrength().getDbm()` provides the RSS value in decibel. The following is an example of how to read this value.

```
import android.telephony.*;

private float rssValue;
private TelephonyManager telephonyManager =
                     getSystemService(TELEPHONY_SERVICE);
try {
  for (CellInfo cellInfo : telephonyManager.getAllCellInfo())
    if (cellInfo.isRegistered())
      rssValue = cellInfo.getCellSignalStrength().getDbm();
} catch (SecurityException se) {}
```

We still need to be able to get the ambient noise amplitude level. We can access the device's default microphone—this means either the built-in microphone or an headset/speaker—at any moment through the `MediaRecorder` class. As we anticipated from Section 3.7, we really do not need to save nor process an audio file; we can infact discard any recording as long as we catch the current noise level. We will not either leave the microphone running at all times. The `MediaRecorder` class is then instantiated, selecting the default audio source and redirect the output on `/dev/null`.[3] A `MediaRecorder` class has the `start()` and `pause()` methods to access the microphone and the `getMaxAmplitude()` method to get the maximum amplitude in decibel of the current burst of audio being recorded. The following snap of code is a sample usage.

```
import android.media.MediaRecorder;

private MediaRecorder mediaRecorder;
private float micAmplitude;
try {
  mediaRecorder = new MediaRecorder();
  mediaRecorder.setAudioSource(MediaRecorder.AudioSource.DEFAULT);
  mediaRecorder.setOutputFile("/dev/null");

  mediaRecorder.prepare();
  mediaRecorder.start();
  micAmplitude = mediaRecorder.getMaxAmplitude();
  mediaRecorder.pause();
} catch (IOException ioe) {
} catch (IllegalStateException ise) {
} catch (SecurityException se) {}
```

[3]Null device: *is a device file discarding all data written on it as a successful write operation. It corresponds to* `/dev/null` *in Linux distributions.*

Of course when we collect training data we want to label it at the moment. We will have to tell the application which of the nine actions listed in Section 1.7. The application's graphic user interface will display radio boxes for activity selection and a checkbox for indoor/outdoor labeling.

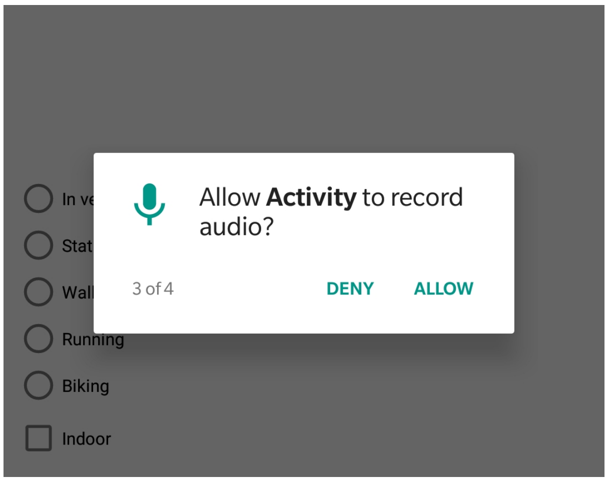
4.4 Android permissions

Some of the code snaps in Section 4.3 are surrounded by `try-catch` blocks handling a `SecurityException`. This is a direct consequence of the Android permissions management framework, introduced with Android 6.0 "Marshmellow" (API level 23). Some sensitive operations within the OS are protected by permissions—e.g. access to camera, microphone, file system, and position. An application must declare all the permissions it will ever need in the AndroidManifest.xml file, as follows:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

The application must then explicitly require each necessary permission to the user at least the first the permission in used. The user is prompted with a pop-up request like in Figure 11, where they can either grant it or deny. This allows the user to have full control over what the application has the ability to access. Permissions can be revoked at any time by the user, and some Android distribution show statistics on when the application accesses a permission-protected feature.

The following sample code proceeds with the request of all the permissions listed with the previous declaration example. The `onRequestPermissionResult` method is invoked



Allow **Activity** to record audio?

3 of 4 DENY ALLOW

Figure 11: An Android application requesting a permission.

each time the user takes an action on a group of requested permissions like in Figure 11. Permissions must be checked each time the application starts, as they could have been revoked by the user after the previous usage, or the application data holding the permission results could have been erased. Table V lists all the permissions used by this application.

```java
private void requestAllPermission() {
  try {
    requestPermissions(Arrays.stream(getPackageManager()
        .getPackageInfo(getPackageName(), PackageManager.
           GET_PERMISSIONS)
        .requestPermissions)
      .filter(p -> checkSelfPermission(p) != PackageManager.
         PERMISSION_GRANTED)
      .collect(Collectors.joining(",")).split(","), REQUEST_CODE);
  } catch (PackageManager.NameNotFoundException e) {}
}
@Override
private void onRequestPermissionResult(int requestCode, @NonNull String
    [] permission, @NonNull int[] grantResult) {
  for (int i = 0; i < grantResult.length; i ++)
    if (grantResult[i] == PackageManager.PERMISSION_GRANTED)
      Log.d("PERMISSIONS", "Permission " + permission[i] + " granted.")
         ;
}
```

TABLE V: LIST OF PERMISSIONS USED BY THE APPLICATION.

Permission	Motivation
`ACCESS_COARSE_LOCATION`	Access the `TelephoneManager` to collect the RSS. This permission is required as this class provides means of accessing network-provided, coarse-grained location, as explained in Section 3.3.
`READ_EXTERNAL_STORAGE`	Read previous data and setting storage files from file system.
`RECORD_AUDIO`	Required to access the device's default microphone.
`WRITE_EXTERNAL_STORAGE`	Write sensor data and classification result on file system.

CHAPTER 5

CHOOSING THE CLASSIFICATION MODEL

Once we have implemented our data collection application, what is left before the classification task is to generate a dataset with sufficient samples for all the classes of interest. Running the application described in Chapter 4, we are left with a collection of records with the following format:

`timestamp` The date and time at which the record was generated; this information is not only important to reconstruct sequences, but the hour of day is used as feature to support the light predictor (cf. Section 3.6).

`activity` The label for the activity assigned to this record during data collection.

`acc_x`, `acc_y`, `acc_z` The values of the three-axial accelerometer.

`gyro_x`, `gyro_y`, `gyro_z` The values of the three-axial gyroscope.

`indoor` The label for indoor or outdoor assigned to this record during data collection.

`lux` The luminance value from the light sensor.

`mag_x`, `mag_y`, `mag_z` The values of the three-axial geomagnetic sensor.

`cellStrenght` The value of the Radio Signal Strength (RSS).

`mic` The maximum amplitude recorded by the microphone.

We are now ready to go on and find a suitable model. The **classification** is the problem of assigning each instance of data (also called records or *points*) in the dataset (called *population*) to one in a set of *classes* or categories. For sake of clearity, this is the set of classes we already introduced in Section 1.7:

(i) indoor biking

(ii) indoor running

(iii) indoor stationary

(iv) indoor walking

(v) in vehicle

(vi) outdoor biking

(vii) outdoor running

(viii) outdoor stationary

(ix) outdoor walking

5.1 Introduction to Machine Learning

Back in 1959, the computer scientist Arthur L. Samuel coined the term **Machine Learning (ML)** to indicate the *"automated detection of meaning in data"* [55] during its pioneering studies in computer game theory [1]. Since then, ML has shown a continuous evolution in every field of computation, addressing more and more different problems of various complexity, but at the ending the goal is the same: *automated learning from data*. Machine Learning is the fundation of Artificial Intelligence (AI). An *agent* is said to be *intelligent* if it has the ability to learn. What **learning** refers to is the ability to progressively improve the performance of a task execution from the data.

Definition 5.1 (Learning, *machine-*, Mitchell [44]). *A computer program is said to* learn *from an* experience *E with respect to a* task *T with* performance measure *P if, its performance at task T, as measured by P, improves with the experience E.*

Machine Learning is used in both supervised- and unsupervised learning tasks, and has today countless applications: search engines, spam filtering, human language understanding, self-driving cars, fraud detection, decision support, and more. ML is today the answer to all those computational problems that are too difficult, if not impossible, to define in a conventional way by a human programmer or designer [55].

One common and relevant problem where ML fits is learning a **classification model**, which is what we need to do for our SLAR task. Exactly as we humans experience the learning process, in the same way a machine needs to be thought, and the way to do this is through labeled data. Let $\mathcal{X} = \mathbb{R}^d$ be the set of data, called **instance space**, where d is the **dimensionality**, i.e. the number of features or descriptors. Then, as we said, the algorithm will need a set of **training data** $\mathcal{S} \subset \mathcal{X}$ with their correct classes. What is left to do now is to find the right learning algorithm for our needs.

There are different ways to classify elements. Most of them are based on discerning the data points based on the values that their *features* (or descriptors, variables, or attributes) assume in a specific instance. Other methods, assign points to a category based on the *similarity* between the point and the ones already in the category. A model that achieve the task of classification is called **classifier**. Finally, before going through some of the most common classifiers and their positive and negative aspects (riassumed in Table VI), we need to keep in mind that there is not a *best* ML model, because as stated by Definition 5.1, we always have to define a performance in relation to a specific task, i.e. all models are not absolutely better than all the others. This concept is expressed by the *"No Free Lunch Theorem"*.

Theorem 5.1 (No Free Lunch, Wolpert, 1996)**.** *Every classification algorithm has the same error rate in classifying unseen data, averaged over all possible data generation distributions.*

5.2 Common classification models overview and limitations

Classification algorithms have been around for a lot of time and there are several well-known of them; the first work on classification goes back to the late 1930s with Fisher [19, 20]. All models have their strengths and weaknesses, which usually strongly depend on the dataset more than on the model itself. An algorithm might be a bad choice for a particular problem with one dataset, and at the same time the best fit for another dataset. This is why, even if we will now go through a brief exploration of what is "good and bad" in some classifier, it sometimes really is a matter of trial and error. Table VI provides a short introduction on the most common classification models, while we move slightly more in depth on how they work.

TABLE VI: OVERVIEW OF CLASSIFICATION MODELS.

Model	Advantages	Drawbacks
Decision Tree (DT)	Tree based models, are easy to interpret if trees are not too deep. Good fit for categorical features with linear decision boundaries. Bagging and boosting can reduce overfit and variance. Provides importance measure for features (e.g. Gini index, entropy)	Very easy to overfit, performs poorly with non-linearly divisible feature space. Variance is not reduced when features are correlated. Large boosted trees lead easily to overfitting.
Logistic Regression (LR)	Logistic Regression (LR) is the extension of Linear Regression with a qualitative response, is a basic model that works well on linear decision boundaries and provides a probability value for the outcome. Model usually have low variance.	Usually suffers from high bias, not suitable for data with high variance and outliers, is highly dependent on the training data.
Naïve Bayes (NB)	Is probably the easiest model available. Extremely easy and fast to build and can fairly handle high dimensionality.	Is based on the assumption of independence of the features and gets worse as the dependency among features gets stronger. Suffers from multicollinearity.
Neural Network (NN) and Deep Learning (DL)	Are quite always the best choice when dealing with non-linear decision boundaries and a large feature set (high dimensionality). Due to their not-so-easy implementation, there are a lot of open source library to help implementation.	Requires features to be reduced to numerical and cannot handle missing data. Require more time and computation to learn the model and the result is non meant to be human readible, becomes a *black box*. Not easy to train for the high number of parameters to tune.

Continues on next page

Continues from previous page

Model	Advantages	Drawbacks
Random Forest (RF)	Random Forest (RF) is an ensamble method with multiple DTs. Improves bagging when features are correlated and reduces variance in DTs.	Same as DTs, plus less easy to interpret visually.
Support Vector Machine (SVM)	Performs similarly to LR, better with non linear boundaries through a careful choice of the kernel. Handles high-dimensional data.	Can be subject to overfit and disturbance from outliers depending on the kernel and margin chosed.

Ends from previous page

5.2.1 Decision Trees

Decision Trees (DTs) are probably the most easy model to understand, because they can be drawn and visualized so easily that, given a graphic representation of a grown tree, we could easily classify a new instance just looking at the tree without any computational support. Decision Trees are created discriminating data points based on their features in a simple way: at each level of the tree, a *split*—the path from a parent node towards one of its child nodes—represents a choice among a value (categorical features) or a range of values (numerical features) for one specific feature, or combination of them. The split attribute is chosen on a "best fit" basis, where *best* is defined according to different policies (e.g. entropy, Gini index).

DTs suffer from high bias. To reduce overfitting, the most two important methods are *early stopping*, i.e. halting the tree growth after a predetermined number of maximum levels, or

TABLE VII: OVERVIEW OF CLASSIFIERS PERFORMANCES

Classification Model	Dataset and approach			
	SL only	AR only	SLAR post-combined	SLAR pre-combined
Decision Tree (DT)	97 %	86 %	83 %	77 %
Deep Learning (DL)	97 %	89 %	86 %	94 %
Logistic Regression (LR)	92 %	72 %	66 %	80 %
Naïve Bayes (NB)	84 %	80 %	68 %	83 %
Random Forest (RF)	91 %	75 %	68 %	74 %

when the split gain reaches a certain threshold, and *pruning*—as in cutting of branches—the tree after a full growth. We usually look for a short tree with possibly not too many branches at each level.

Decision Trees are a perfect fit when the data can be separated according to splits parallel to the axis. Imagine a dataset with two numerical features x and y. If the data is separable according to multiple splits like $x \leqslant x_0$, then a DT will perform just fine, but things get tougher when the separation boundary assumes forms like $x \leqslant x^3 - x^2$ or worse.

5.2.1.1 Random Forest (RF)

To overcome some of the limitations of Decision Trees, statistical learning proposes an *ensemble method* called **Random Forest (RF)**. Ensemble methods are combinations of simpler algorithm, and a Random Forests are, like the name suggests, collection of Decision Trees. The single DTs are trained with different parameters one from the other, and the result of the classification of an instance with a Random Forest is usually the **mode** of the classes output by each single tree in it. The learning process differs from the *bagging* algorithm—random selection of a subset of points—because the random subset is defined on the features set at each split of the tree growth. This method can lead to an increase in variance keeping bounded the bias that DTs have due to their tendency to overfit on the training data, but this improvement is not strictly guaranteed and introduce a complexity in the model understanding that was typical of DTs.

5.2.2 Logistic Regression (LR)

The **Logistic Regression (LR)** is a model that derives the odds of the probabilities of an event from a function that is a linear combination of—assumed—independent predictors. LR is by definition a binary decision method, but its multinomial extension can be used for categorical predictions instead. Logistic Regression has multiple common application fields, most notable are medical and social scences, where it is used to evaluate risks for medical or financial conditions given some parameters for individuals [12].

At the basis of multinomial LR, stands the following equation:

$$\begin{aligned} f(k,i) &= \beta_{0,k} + \beta_{1,k}x_{1,i} + \cdots + \beta_{M,k}x_{M,i} \\ &= \beta_{0,k} + \sum_{j=1}^{M} \beta_{j,k}x_{j,i} \end{aligned} \tag{5.1}$$

describing the probability of the category k for the i-th record, where the values $\beta_{m,k}$ are called **regression coefficients**, each associated with the m-th predictor $x_{m,i}$.

Logistic Regression is not a particularly complex model, and allows some degree of understanding of the problem by reading the learned regression coefficients for each predictor from Equation (5.1) as weights of such variable in the final outcome. But it does not always fit the problem. In the first place, LR makes some assumptions; the most important one is the case-specificity of the data—each independent predictor assumes a specific value for each case. Like most of the classification models, the *independence* expressed in the previous statement is not to be read as statistical independence of the predictor (differently from, for example, the Naïve Bayes classifier), yet *collinearity* should be significantly low among the predictors. Logistic Regression usually suffers from an high bias, and thus does not perform well on data with high variance and with a significant presence of outliers.

5.2.3 Support Vector Machines (SVMs)

A **Support Vector Machine (SVM)** is a classification model based on separation boundaries; it is commonly adopted for several tasks like text and image classification, hand writing recognition, outliers detection and biology applications. When learning an Support Vector Machine (SVM), we usually map data points into a space in which they are likely to be separable; then the SVM tries to figure out a separation boundary able to keep samples of different classes apart with a certain gap.

If we have an n-dimensional dataset, than this task corresponds to finding an $(n-1)$-dimensional **hyperplane**; if a separating hyperplane exists, then there probably exist more than one. In such a case, we might want to find the one such that the separation *margin* between the two or more classes is as wide as possible; we call this the **maximum-margin hyperplane**.

If a separating hyperplane cannot be found for a specific set of data, than the common procedure is to define a **kernel function** $\ker(x, y)$ that remaps the feature space into a (usually much) higher dimension space, in the hope of *"spacing"* the points enough to find an hyperplane through them. Another technique, can consist in define a **soft margin** that allows certain data points to *cross* the decision boundary set by the hyperplane, falling into a category different than the one they belong to.

In many cases, if the data is hardly separable, finding such hyperplane can become computationally hard, and the model becomes less and less understandable and scalable, while also reaching overfit. Depending on the data, the kernel function and the type of margin (hard or soft), the SVM can particularly suffer from outliers and noise.

5.3 Introduction to Deep Learning

Back in Section 5.1, we introduced the concepts of Artificial Intelligence (AI) and Machine Learning (ML), and what we refer to with the words *intelligence* and *learning* in computing. Scientists began to wonder whether machines would ever come to *think* long before we ever built one. Not extremely surprisingly, the most successful path towards what we call today Machine Learning, was born from a mathematical attempt of emulating the human brain—more specifically, the **biological network of neurons**: we call them **Artificial Neural Networks**, or more shortly just Neural Networks (NNs). Since then, progresses in biological

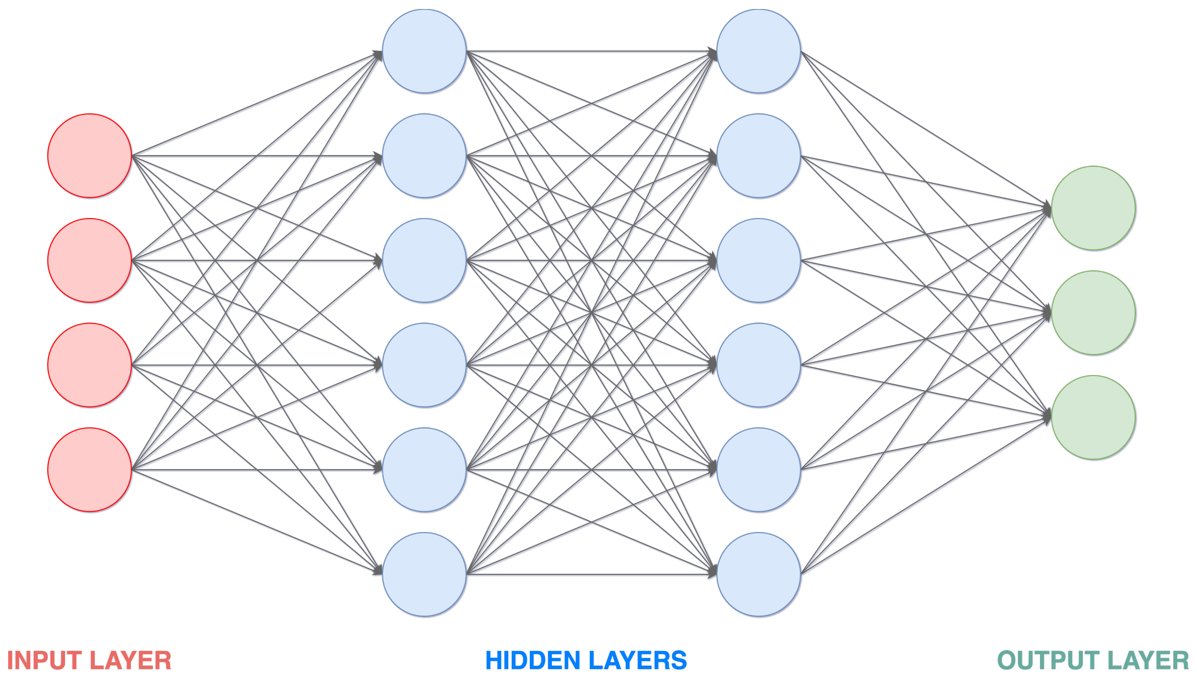


Figure 12: Representation of a simple Neural Network.

neural networks have helped the progress in artificial ones and, surprisingly, the other way around as well.

A Neural Network is an interconnection of nodes named **neurons** through which input data *flows* up to terminal nodes, and can be represented in a network like the one in Figure 12. In one way or another, depending on the *learning process*, the algorithm assigns a value to each neuron representing a weight. Neurons are organized in sequential levels called **layers**, each level performs a certain *function* on data coming from the previous level, which is combined of matricial products between data and weights and summed up across all units. The first

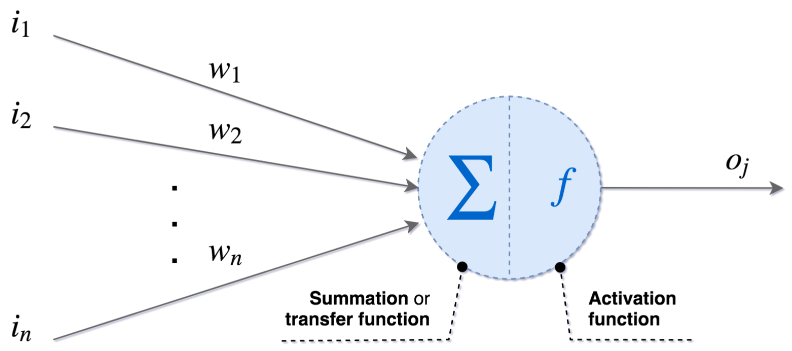


Figure 13: Anatomy of a simple neuron.

layer of a NN is called **input layer** and usually assumes the form of a vector $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ (recall the symbols introduced in Section 5.1); the last layer is called **output layer** and usually assumes the form of a vector $\mathbf{y} \in \mathbb{R}^c$, where c is the number of categories in a classification task. All the other intermediate layers are called **hidden layers**, and are vectors of the form $\mathbf{h_i} \in \mathbb{R}^{\dim(h_i)}$.

Each neuron is actually made up of different component. We will see in Section 5.4 how complicate a neuron can get, but for now let us see the basic and inevitable pieces of a simple cell. As we see from Figure 13, the neuron receives a vector $\mathbf{i} \in \mathbb{R}^n$ of inputs values from the previous layer of dimension n, with each one of these values associated with a weight w. The first operation within the neuron is this the summation of the products of each value and its weight or, in other words, the dot product $\mathbf{i} \cdot \mathbf{w}$. Then, the next step is to apply to the result of the previous computation, the function proper of this layer. We call this function

activation function f, which is usually associated with an activation **threshold**. The result of this function o_j is the either fed to the next layer, or is one output value of the network if this is the output layer. The neurons can also receive as an additional input a **bias** value that takes part in the first summation.

Neural Networks have continuously evolved over time becoming more and more complex as computation hardware got faster and faster. Soon Graphics Processing Units (GPUs) became a requirement for high standard computation, allowing NNs to grow in size and complexity so to achieve better results. For size growth we do not only refer to the layers' dimension, but also to the number of hidden layers. We started to refer to NNs with more hidden layers as **Deep Learning (DL)**.

Many common ML algorithms suffer from what in statistics is referred to as the **Curse of Dimensionality** [31], a well known problematic condition arising when the data has an high number of dimensions. Increasing number of variables and their dimensions leads to an exponential increase in data size and complexity, and this complexity is not only due to large processing tasks. One of the most important consequences of high dimensionality is that the number of possible combinations for the input $\mathbf{x}$ increases while the size of the training data set does not necessarily do so. This leads to a set $\mathcal{X}$ much bigger in size than the knowledge base set of labeled data $\mathcal{S}$, making it harder to be accurate on an enormous variety of possible future unseen cases learning on an infinitesimal portion of it.

Different layers of a Neural Network implement different functions to achieve the expected result, and a multi-layer *deep network* with large layers allows to build up more complex functions. This layered architecture is referred to as **feature hierarchy**, building up levels of different complexity and abstraction. Deep Learning performs an **automatic feature extraction**, meaning that relevant information is extracted by the algorithm without the need of human intervention.

5.3.1 Feedforward learning

Feedforward NNs are a fundamental concept in Deep Learnings. We said before that layers of a NN implement a function, so overall a classifier is a model that implements something like $\mathbf{y} = f^*(\mathbf{x})$, as a *map* of the input to a category. A **Feedforward** network, more precisely, implements a function of the form $\mathbf{y} = f^*(\mathbf{x}; \theta)$, where θ are the optimal parameters defining the best approximation, that have to be learned.

The term *feedforward* recalls the concept of the flow of information from the input layer $\mathbf{x}$ through the inner layers $\mathbf{h_i}$ and out from the output layer $\mathbf{y}$. More importantly, *forward* emphasizes the fact that the information "travels" exclusively in such direction, and never backwards.

We have introduced before the **weights** that we see in Figure 13. These are the very values that the algorithm has to learn in order to achieve a more and more precise result. The first step is to initialize these weights; then, data is run through the network several time, and each time the model produces a *guess* for the result, compares it with the expected one, and based on the achieved error, the weights are updated so to get as close as possible to the expected result. Every iteration, then, is a different state of the network and is a new model drawn from the previous ones. Let us imagine to overview the network at the most abstract level possible. If we would try to summarize the steps for training it, it would be something like this:

(i) the network is initialized;

(ii) data flows into the input layer;

(iii) produce a guess: `guess = input * weights`;

(iv) see what big an error the network has done: `error = guess - ground_truth`;

(v) compute the needed adjustment based on the weight's contribution to the error:

`adjustment = (weight's contribution to error)* error`;

(vi) apply adjustment and repeat from Item iii.

As shown by Figures 12 and 13, all network nodes receive the output of all the nodes in the previous layer. This mean that the features are continuously combined with each other, with different coefficients, combining them together in different proportions. The deeper the network, the more this phenomenon is accentuated. Combinations that are more relevant, then, will have higher importance than the less useful ones, producing the effect of automatic feature extraction that we mentioned earlier.

5.3.2 Gradient descent

The progressive adjustment of weights to meet the desired output that we mentioned in Section 5.3.1 is an **optimization problem**. Optimization problem is a set of tasks that aim at optimizing a certain "goal function" so to reach an *optimal* result. One of the approaches to optimization problems is called **gradient descent** and is extremely recurrent not only in Neural Networks.

The term *gradient* has a meaning strictly related to the slope of a function. If we think of a function that somehow describes the error we make in the network, then we want to find the lower point of this function and work with it. This means, we want to find a *minimum* of the error. The *descent* basically means that we "walk" on our function's plot going downwards, trying to go as down as possible to reach such minimum. Calculus teach us that to find minimum points of a function we introduce derivatives, and thus said gradient, which is a derivative representation of multi-variable functions.

Let us go back a moment to Figure 13. Each time data flows from on layer to another, it is repeatedly mapped, or transformed, by a new function. The network then is nothing more than a big chain of nested functions, something in the form of

$$f_n(f_{n-1}(\dots f_2(f_1(x)))). \tag{5.2}$$

Recalling the **chain rule** of calculus,

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}, \tag{5.3}$$

we can express the relation between the error e made by the network and each weight w, by mean of the activation function a, as

$$\frac{de}{dw} = \frac{de}{da} \cdot \frac{da}{dw} \tag{5.4}$$

so that we are able to determining how changes in the weights affect changes in the activation function and, thus, the error.

We said in Section 5.3.1 that in feedforward network the information flows only from the input layer towards the output layer. During the training, this forward propagation produces a cost J. The **back-propagation** algorithm makes this cost *flow back* through the network layers allowing the gradient computation based on the chain rule in Equation (5.3),[1] that we should generalize Equation (5.3) to the non-scalar case, where $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, $g\colon \mathbb{R}^m \to \mathbb{R}^n$, $f\colon \mathbb{R}^n \to \mathbb{R}$, $\mathbf{y} = g(\mathbf{x})$, and $z = f(\mathbf{y}) = g(f(\mathbf{x}))$; then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}\,. \tag{5.5}$$

Or, to use the proper gradient notation

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{x}}{\partial \mathbf{y}}\right)^{\top} \nabla_{\mathbf{y}} z\,, \tag{5.6}$$

[1]Back-propagation is only used to compute this gradient, but the gradient is not used to learn the model. Instead, another learning algorithm does so, called **stochastic gradient descent**.

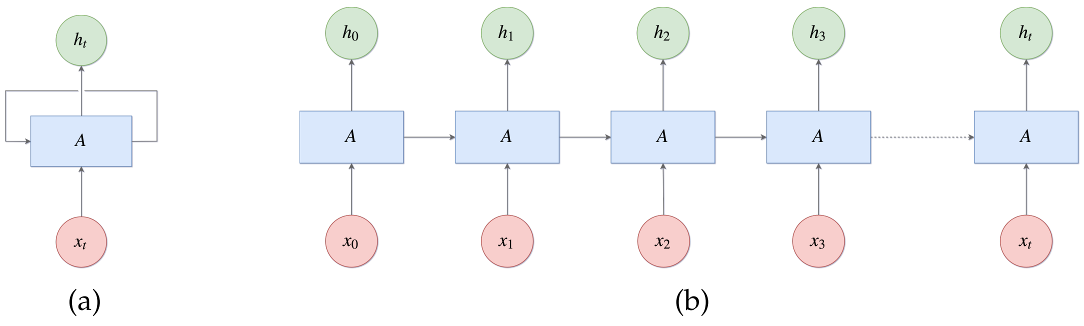


Figure 14: A simple Recurrent Neural Network cell represented with a feedback loop (a) and unfolded (b).

with $\partial \mathbf{x} / \partial \mathbf{y}$ being the Jacobian of g. Going deeper in details on the calculus and performance of these computation is beyond the scope of this work.

5.3.3 Recurrent Neural Networks

We talked previously about how Neural Networks try to emulate the human brain behaviors, even for difficult tasks. But one strong characteristic of the human mind is that its understanding of what is going on strongly depends on past experience, sometimes from a very recent past, sometimes longer. Being able to use previous information for future situation is what makes our thinking process so articulate. For example, while reading a document, our understanding of its parts is based on our understanding of previous parts; more strongly, the understanding of a sentence is based, work by word, by all those who came before. If scrambling the words in a sentence ends up making no sense for us, so should be for a NN, whenever it matters. This means that as our thoughts are persistent in our minds, some pieces of information should sometimes persist in within the model as well.

Recurrent Neural Networks (RNNs) were introduced in 1985 by Rumelhart et al. [53] for a better fit in processing sequential data. The concept of having information persist inside the network is reflected in *"loops"* as shown in Figure 14 (a). The information does not only flow through the cell from one layer to another, but also loops back in. A time-division visualization is shown in Figure 14 (b): each result goes into the same cell at the next time iteration $t+1$.

This *chained* architecture in Figure 14 (b) suggests their particular fit to sequential data. Today, RNNs are widely used for applications that enclose a notion of time and sequentiality, like speech recognition and translation.

5.4 Long–Short Term Memories for time series data

This loop architecture of RNN cells is not always enough to recreate the process of keeping past information to understand new one. Specifically, the problem is in the word *past*. Let us think about these two sentences, where we want to predict the last word:

(i) *«I lived in France and can speak good. . .* French.*»*

(ii) *«I lived in France before moving to Spain, so I can speak good. . .* French.*»*

In both cases we expect to come up with *French*, and in the first sentence it is pretty clear, something a RNN could achieve by looking at the context. But, in the second sentence, the relevant contextual information is farther away, and is interleaved with similar and confusing pieces of other information. This abstract exampe makes us think that what we have to recall in order to understand something in the present, could be in different *pasts*. There are things we need to *keep in mind* for a longer time than others, and here most RNN models fail. This problem goes under the name of **Long-Term Dependency Problem**, and has been first pointed out by Hochreiter [27] and Bengio et al. [10], and RNNs struggle from it.

We saw already in Section 2.4 that the literature proposes us a good solution to this problem, called LSTM. A **Long Short-Term Memory (LSTM) Network** is a *"gated"* variant of a RNN, where the term **gate** replaces the term *node* or **cell** for its increased complexity. This variety of RNNs is particularly suited for time series data, as supported by various research literature and related work (Zhu et al. [67], Liu et al. [40], Wang et al. [62], Veeriah et al. [59]).

Long Short-Term Memories (LSTMs) were introduced around 1997 by Hochreiter and Schmidhuber [28] and then extensively studied and refined by a large number of members of the research community, expecially studying LSTM performances on tasks that were before not suitable for common RNNs.[2]

LSTMs were appositely studied to overcome the long-term dependency problem, embedding the ability of holding on a piece of information for a much longer time as a standard behavior. We saw in Section 5.3.3 and Figures 13 and 14 (a) that a common RNN cell applies a single function f to its input values, with a feedback loop and an output value. The LSTM architecture still follows the chain model from Figure 14 (b), but the internal structure of a single node gets more complicated, and the node gets the name of **gate**.

[2]A non-comprehensive list of research work exploring applications of LSTMs to tasks where RNNs did not perform well: Baccouche et al. [6, 7], Bengio et al. [10], Graves et al. [25], Schmidhuber et al. [54], Gers et al. [22, 23], Liwicki et al. [41]. See Section 2.4 for more details.

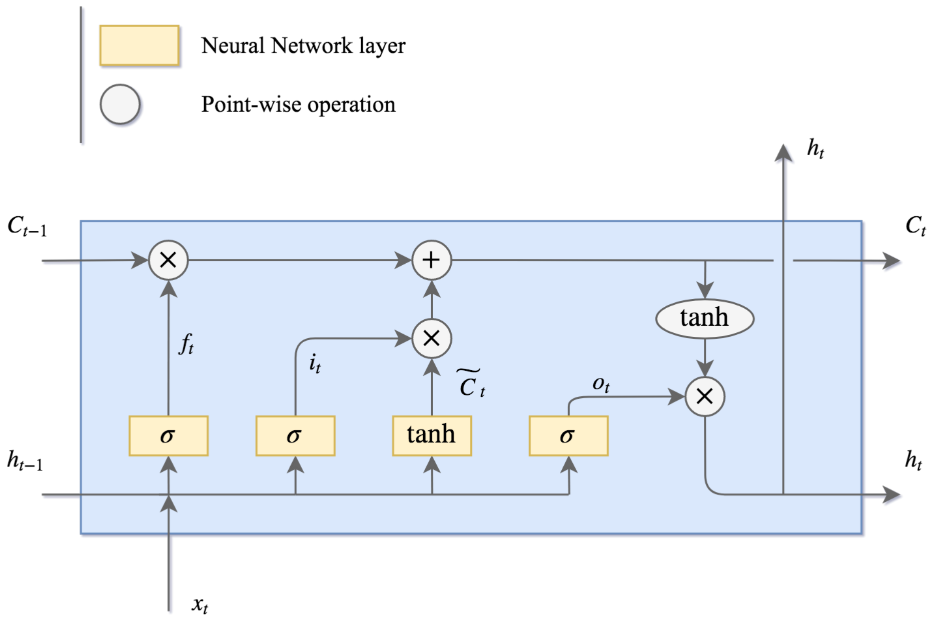
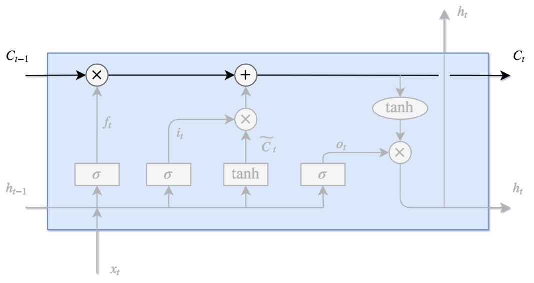


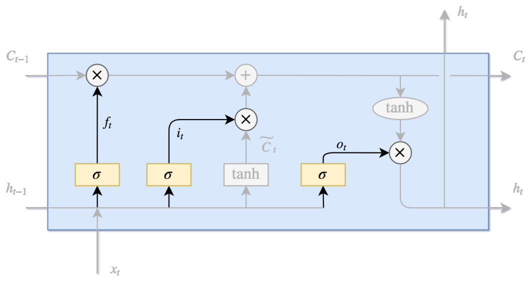
Figure 15: Basic architecture of a LSTM gate.

Figure 15 describes the architecture of a LSTM cell. Arrows represent flows of vectors, from the output of one node to the input of other nodes; the loopback structure from Figure 14 (a) is obtained by connecting the C_t output to the C_{t-1} input of the next step, same for h_t. gray rounded nodes are point-wise operations (sum, product, and point-wise applied functions); yellow rectangular shapes are NN layers performing a mapping function as those describes in Section 5.3 and shown in Figure 13; x_t and h_t are the cell input from the previous layer and input to the next one, respectively.

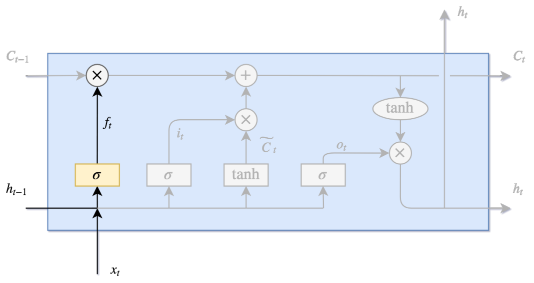
Let us get deeper in the LSTM gate anatomy to understand what all the different "pieces" in Figure 16 are for.



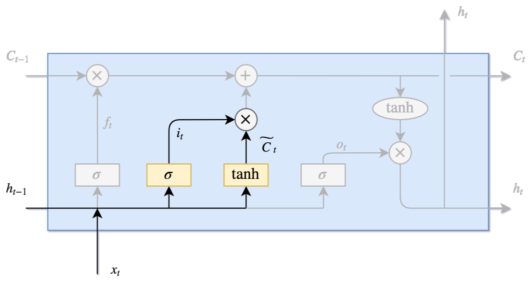
(a) Cell state.



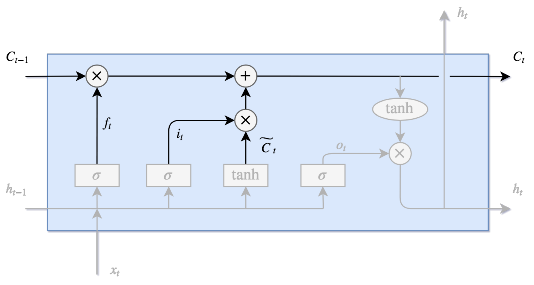
(b) *Sigmoid* layers.



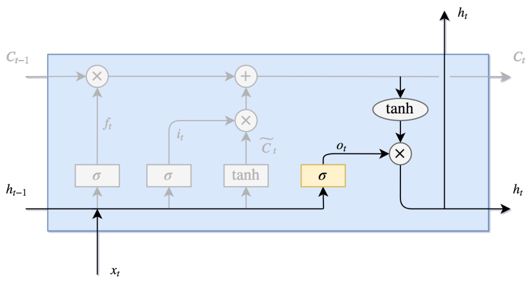
(c) Forget gate.



(d) Input and tanh gates.



(e) Status update.



(f) Output gate.

Figure 16: Different components of a simple LSTM gate.

- The uppermost flow highlighted in Figure 16 (a), running from C_{t-1} to C_t is called the **cell state**, or sometimes just **cell**. The cell state goes through some point-wise operations—sums and products—that are the means by weach its modifications are regulated, adding to and removing information from it. The other gates will regulate which information to add to the state and which to no longer carry along.
- The layers in charge of letting the information through or not are called **sigmoid layers**, represented in Figure 16 (b) as yellow rectangles with the σ symbol. They are sigmoid NN layers followed by products. A *sigmoid* layer implements a function $g\colon \mathbb{R} \to [0,1]$ indicating the *portion* of information to let through, from 0 (none) to 1 (all). There are three of these gates in the cell, that are in charge of different operations.
- We said the LSTM cell has to decide whether and what information to retain or discard. The decision of what to discard is made by the first sigmoid layer in Figure 16 (c) that we call **forget gate**. It collects the information from the previous output of the cell h_{t-1} and the current input x_t and outputs a number in $[0,1]$ related to each component of C_{t-1}. Each of these 0 to 1 values will be multiplied point-wise to the values in C_{t-1}, so values of 0 will get rid of the corresponding value in C_{t-1}, while a 1 will keep it. In this way, the previous cell state is *preprocessed* so whatever information is not relevant anymore is discarded before proceeding with this iteration.
- After we "forget" what we no longer need, we take care of what new to add to the cell state. This operation is carried out by the two different gates in Figure 16 (d).

  - First, another sigmoid layer implements the **input gate**. This decides which value will be updated with the new values from the input.
  - Then, the tanh layer generates a vector $\widetilde{C}_t$ of *candidates* for the new values. These values will be part of the *status update*. To multiply together these two values means scaling the new candidates by how much we intend to update each value.

- We need to proceed with updating the previous cell state C_{t-1} into C_t. We proceed by applying the *forget* step f_t to C_{t-1}, then add the new input generated in Figure 16 (d). This is done by the connections in Figure 16 (e), implementing the formula

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \,. \tag{5.7}$$

- Finally, we process the **output** with the blocks in Figure 16 (f). The cell state is scaled between -1 and 1 by the tanh operator—this is not the tanh layer from Figure 16 (d), but just a point-wise operation. Then, it is multiplied by the output of the sigmoid gate o_t, as in the formula

$$h_t = o_t * \tanh(C_t) \,. \tag{5.8}$$

Long Short-Term Memories have been proven to be particularly suited for complex tasks with sequential data. Their most important and widespread applications are:

- speech recognition;
- grammar learning;

- handwriting recognition;
- human action recognition;
- anomaly detection;
- business model management.

State-of-the-Art LSTMs are currently underneath most high-end products from top-notch companies. Google uses them for speech recognition, smart assistant and translation; Apple and Amazon use them for their relative smart assistant too; Microsoft uses them in their AI products.

5.4.1 Dropout

There are several bagging methods for training Machine Learning algorithms—we mentioned it in Table VI for Decision Trees, but this procedure gets computationally expensive for more complex models like Deep Learning, due to the time requirements to run the model training multiple times [24].

Srivastava et al. [56] in 2004 introduced a regularization method called **dropout** that with basically no computational overhead, provides a similar effect to bagging for complex and ensemble methods. The idea under dropout is to remove some of the (non-output) units from a network—this is done as easily as multiplying a unit's output by zero.

What happens at training time is that each time an example is input to the network, each input unit is either included or not, with probability p_{x_i}, and each hidden unit is either included or not, with probability p_{h_i}, where p_{x_i} and p_{h_i} are hyperparameters defined before-hand and not correlated with neither the current value of the unit, the current input, nor the outcome for the other units. Usual values are $p_{x_i} = 0.8$ and $p_{h_i} = 0.5$.

5.4.2 Sliding window classification

With a model built to be learning from data evolution over time like Long Short-Term Memories, it is a natural response to classify **batches** of sequential records, so that an input vector to the model represents a short *burst* of motion data.

As shown in Chapter 2, the research literature suggests that a *sliding window* classification can improve performances [8, 58, 17]. By sliding window we mean that, given a longer series of motion data of length t, we feed the model with a window of $w < t$ records at a time. Then, we *slide* the window forward in time of $s < w$ records, so each windows overlaps with the previous one for $w - s$ records. This helps preserve the time evolution of the data among multiple classification instances.

Classification on a window of records can bring to more **stable classification results**: the impact of slow perturbations. For example, a slow and small movement like moving the phone on a desk or typing on it, does not induce an error on the classification that should remain as *stationary*. In the same way, hitting a pothole while biking or driving should not impact the classification.

Studies have shown that classification accuracy is stable across different window lengths; yet, a sufficiently long window is required to capture a possibly complete *cycle* for different activity [8]; for example, it take us up to around one to two seconds to take a complete step, so this is the minimum window length we might want to use.

CHAPTER 6

LEARNING THE CLASSIFICATION MODEL

We introduced the concept of *learning* in Section 5.1, and following the Definition 5.1 we know that we need to take the necessary step of letting the model chosen in Section 5.4 learn from the data collected as in Chapter 4. Each model has its way of learning; that means that as an instance of the input is fed to the algorithm, it executes different operations proper of the model to get the information it needs.

We saw in Sections 5.3.1 and 5.3.2 how learning works for Neural Networks and Deep Learning, and now its time to actually implement this process on a machine. In the following sectons we will go through some of the necessary steps for training the model and the technologies that we dispose of to accomplish this task.

6.1 Environment setup

In the last years, the Python programming language has became a must in data science tasks. Python is an **interpreted** language firstly released in 1991 with the aim of emphasize readability. Most of the things the programmer shall care about with compiled programming languages, like data types and memory management, are hidden or abstracted in Python, so that the user focuses more on the problem solving than on the programming itself.

Around Python, developers have created a great system of integrations that allows easy access to functional tools. For data science settings, Python comes with a variety of packages to simplify operations. In this work, we adopt mostly four fundamental packages:[1]

NumPy provides a solid ground for scientific computing. Sometimes *strongly-typed* construct are necessary in computation, and this package provides a C-based implementation of data types and operations.

Pandas provides high-performance data structures and analytics tools, incuding time series data.

MatPlotLib a useful library for plotting quality figures from data, used throughout this work for most of the figures.

SciKit-Learn a powerful Machine Learning library for Python with support for a lot of ML algorithms, data preprocessing, and full interoperability with NumPy.

6.1.1 The TensorFlow framework

A great advantage with Python is that it has been enriched with its TensorFlow framework that we introduced in Section 1.6. TensorFlow™ is an open source library intended for High Performance Computing (HPC).[2] TensorFlow is a complex library that handles **distributed**

[1]Numpy: `numpy.org`; Pandas: `pandas.pydata.org`; MatPlotLib: `matplotlib.org`; SciKit-Learn: `scikit-learn.org`.

[2]TensorFlow is an open source project powered by Google and publicly available at `tensorflow.org` and `github.com/tensorflow`. See Section 1.6 and Table I for a wider overview.

numerical computations [21]. Its paradigm is based on a workflow graph, which makes it a functional programming language.

TensorFlow is able to exploit multi-Graphics Processing Unit (GPU) enabled servers and workstations with up to thousands of computational units, making it possible to perform training and inference with large Machine Learning and Deep Learning models [21]. From an abstract point of view, TensorFlow gets the computation flow defined in Python, and runs it in an efficient way, since its underlying code is written in C++. TensorFlow allows automated and controlable parallelization across different, single- or multi-core Central Processing Units (CPUs) GPUs, even distributed across several servers. TensorFlow has basically no complexity limits; it can train models with millions of parameters with billion of records with millions of features; its so empowered natures comes from its developends as the very underlying structure of some of the most powerful Google products like Google Cloud Speech and Google Search itself [21].

TensorFlow is available for all major desktop and mobile platforms. For Python, it comes with the TF.Learn API `tensorflow.contrib.learn` that has full compatibility with the SciKit-Learn library introduced before, allowing an easy data manipulation. Sometimes writing a TensorFlow model from scratch might not be a very easy task. Here, some powerful, higher-level libraries come at hand. The most commonly used is Keras , that provides TensorFlow as one of its backend engines.

TABLE VIII: KERAS SPECIFICATIONS.

Developers	François Chollet (founder) and various developers
Initial release	March 27, 2015
Latest stable release	2.2.0, June 7, 2018
Repository	`github.com/keras-team/keras`
Languages	Python
Platforms	Cross-platform
Licence	MIT
Website	`keras.io`

6.1.2 The Keras API

Keras is a set of APIs for Python that allow to handle Neural Networks models from a very high level.[3]

Keras can adopt different *backends* to actually run mathematical operations on top of the defined models; at the moment it supports TensorFlow, CNTK and Theano. Keras provides easy modeling of Convolutional Neural Networks and Recurrent Neural Networks and supports both CPU and GPU computation.

[3]`keras.io`.

6.2 Data preprocessing and feature scaling

One of the fundamental steps in data mining is **data preprocessing**. This steps is meant not only to allow the training process in the first place, but also to significantly improve the model performances that are affected by improper data.

Raw data indeed usually suffers from some problems that worsen learning and prediction, e.g. conceptually wrong data records or forbidden values, and sometimes even make these operations impossible, like missing values. Data preprocessing is a collection of tools that allows to take control over the *data quality* and thus is a must-take step before moving on with the actual data processing. Data preprocessing operations are grouped in five families: data cleansing, data selection, feature normalization, feature transformation, and feature selection and extraction.

Data cleansing This is the process of identifying "bad" records that might have been collected by mistake or happen to contain improper values, for example for a sensor misreading. In our case, the data cleansing process consists in removing sensor misreadings for the accelerometer and gyroscope—recall the dataset structure from the beginning of Chapter 5. When the magnitude of the acceleration $||\mathbf{A}||_2^2 = A_x^2 + A_y^2 + A_z^2$ is extremely small compared to the gravitational force, say $< g^2/100$, than the value becomes unreal as it gets closer to free fall, or to values we would only experience near the pole. We then get rid of these vales as they can affect the classification accuracy.

Feature selection This consists in the step of selecting those features, from those already in the dataset, that are actually relevant to the classification, leaving out does that do not bring any advantage. Less features can lead to a significant reduction in the model complexity and thus the training time requirement as well. Feature selection in also the base approach to deal with what we already introduced as *curse of dimensionality*. Different methods allow generating some indeces about features relevance, including Logistic Regression itself (see Section 5.2). All the features collected turn out to be somewhat significant to the model, both from a mathematical and a conceptual point of view, for all the reasons explained in Chapters 2 and 3 sustained by the related literature. Nevertheless, as explained earlier in Section 3.3, we were able to safely remove the speed component from our records without any significant loss in accuracy (less than 0.5 % overall); this allows us to still get an accurate model while getting rid of a dimension that, while still not having a relevant weight in the computation, would have a significant impact on battery life for the reasons explained in Section 3.3.

Feature extraction Not to be confused with feature selection, feature extraction is the process of deriving *new* features from the ones already available in the dataset. Sometimes the input data is unnecessary large, and less dimensions can be derived by feature extraction by reducing the amount of variable neccessary to explain the data. We performed feature extraction by transforming the three-axial geomagnetic field values into its magnitude, as we are interested in the perturbation of its magnitude instead of its repartition along the axis. This transformation is explained in Section 3.4 and Equation (3.2).

One necessary step in most Machine Learning problem is that many models require some data transformation to work. Categorical classification methods need encoding for the classes; this means we transform the categories to be coded no longer with a name or tag, but as a binary vector as large as the number of categories; each binary value turns to 1 when that instance belongs to the correspondent class. Python module `sklearn` provides a `LabelEncoder` class for this task.

```python
from sklearn.preprocessing import LabelEncoder
labelEncoder = LabelEncoder()
labelEncoder.fit(y)
y = labelEncoder.transform(y)
```

Expecially when working with NN, one fundamental step is **feature scaling**. Having the network working with big values is not preferable, as it takes away meaning from smaller values. If different features have a significant difference in value range, features with a shorter range lose their importance. There are different ways to scale values; the most common one are standardization—or Z-score normalization, and min-max scaling. **Z-scre standardization** rescales all features so that they will all follow a **standard normal distribution**, i.e. with mean value $\mu = 0$ and standard variation $\sigma = 1$. The standardized value for each instance, called z-**score**, is computed as

$$z = \frac{x - \mu}{\sigma} . \tag{6.1}$$

Recentering all values around 0 is not only useful for the reasons just explained, but are usually also a conceptual requirement of many ML models.

6.2.1 Training

Although it might seems we are dealing with two different problems—Semantic Location detection and Activity Recognition, we discussed how they can be successfully merged together in Section 2.3, sustained by works like Raj et al. [50].

In the previous overview on different classification models, in Section 5.2, we presented results for Deep Learning and other common classifiers in Table VII. The table reports four results for each model: SL only, AR only, and SLAR, both *pre-combined* and *post-combined*. Post-combined classification means that, after performing the classification on SL only and AR only, we combine the two subclasses to obtain those of interest. Not too much surprisingly, even if the results for the two separate classifications are the highest, the performances of all classifiers **drop significantly after merging** them. The reason is that type-I and type-II errors accumulate to produce an higher uncertainty.

On the other hand, running the classification on the entire dataset to solve the nine-category problem as a whole (pre-combined) produces much better results, sometimes as accurately as the two classifications by itself, expecially for Deep Learning. One of the reasons for this has its roots in what explained in Section 5.3: Neural Networks, and Deep Learning (DL) in particular, are unprecedentedly able to decide on the importance of not only each feature by itself, but on many, complex combination of them. The final concept is that both **SL and AR benefit from each other's features**.

Finally comes the moment for the actual training. As explained in Sections 5.4 and 5.4.2, we are going to exploit the benefit of batched classification for time series data [8, 58, 17].

For this goal, we need to *reshape* the data in **segments**. We already discussed how Dernbach et al. [16] shows that classification accuracy is stable across different window size; thus the values are basically almost arbitrary, but we discussed some considerations to keep in mind in Section 5.4.2, i.e. that we want a window wide enough to capture the characteristic. repeating cycle of some activities like walking pr biking.

For this process of *reshaping* then we choose a number of **time steps** for each window, i.e. the **window length**, and a **sliding step**, the number of records to shift the window of, i.e. the number of record going out, and in, each time the window slides. The following few lines of ython code build up the final dataset after the preprocessing in Section 6.2.[4] The `stats.mode()[0][0]` in the last line assigns to each window the **majority class** (mode) of the classes of all the records belonging to such window.

```
TIME_STEPS = 16
STEPS = 4
segments = []
labels = []

for i in xrange(0, len(x[:, 0]) - TIME_STEPS, STEPS):
  segments.append(x[i : i + TIME_STEPS, :])
  labels.append(stats.mode(y[i : i + TIME_STEPS])[0][0])
```

[4]The variables `x` and `y` in the listing are two `numpy` 2-D arrays containing the records and the labels, respectively. The `x[a:b, c:d]` syntax is a *rectangular* indexing, i.e. selects the rows from `a` to `b`, and columns from `c` to `d`—right end is exclusive. A semicolon `':'` by itself acts as a wildcard.

import tensorflow as tf

Listing 6.1: Example of a Keras model definition.

```
import keras
from keras import regularizers
from keras.models import Sequential
from keras.layers import *
from keras.initializers import RandomNormal
from keras.utils import np_utils

sess = tf.Session(config)
keras.backend.clear_session()

models = Sequential()
model.add(
  LSTM(64, input_shape = (window_length, 11)\caption{},
    bias_initializer = RandomNormal(),
    kernel_regularizer = regularizers.l2(),
    kernel_initializer = 'uniform',
    activation = 'relu')
  )
model.add(Dropout(0.2))
model.add(
  LSTM(64,
    bias_initializer = RandomNormal(),
    kernel_regularizer = regularizers.l2(),
    kernel_initializer = 'uniform',
    activation = 'relu')
  )
model.add(Dropout(0.5))
model.add(Dense(units = 9, activation = 'softmax'))
```

Listing 6.1: Example of a Keras model definition.

Now that we have the dataset ready for training, we need to define the model. Keras provides an easy `add` method to add layers to the `model`, as shown in Listing 6.1. The code creates a model with as first layer an LSTM layer of 64 units and 11 inputs (as the number of features), and a second one again of 64 units. These two layers have a *relu* activation function. *Relu* stands for Rectified Linear Units. Back in Section 5.3 we introduced activation function, and referred as them as *sigmoids* in Section 5.4. The sigmoid is the most commonly used in ML, but is not the only choice. A sigmoid has a shape of:

$$f(x) = \text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{6.2}$$

$$f'(x) = f(x)(1-f(x)) \tag{6.3}$$

Given the sigmoid expression and its derivative in Equations (6.2) and (6.3), the result is that the maximum value in the sigmoid's derivative is one fourth of the maximum value of the function itself; this means that errors are reduced by a factor of four at each layer, that can result in loss of data. Rectified Linear Units have recently replaced sigmoids in DL, replacing Equation (6.2) with

$$f(x) = \max(x, 0) \tag{6.4}$$

rectifying the negative part of the output, in a way that seems to be more similar to the actual human neurons way of operation. Research has shown that training with *relu* activation functions results in much faster training time [34]. These two layers are interleaved with two *dropout layers*, explained in Section 5.4.1.

The last layers instead has a *softmax* activation function. This is a very common choice in multinomial classification. A sigmoid provides one result in the $[0, 1]$ interval and thus can work as an activation function for two classes. The *softmax* function (or normalized exponential function) instead divides the result across the different classes, so that each of their results is in $(0, 1]$ and they together sum to 1. This gives a mathematically correct notion of class probability, which is the scope of the classification problem. With this function, we can obtain the output in the best form we can expect:

- once defined a *confidence threshold*, if there is a majority class that overs it, then we can take it as the predicted class;
- if not even the majority class has a confidence higher than the threshold, then the result is considered as the set of all classes with their respective confidence.

This is because the softmax function is compliant to the definition of a categorical probability distribution:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum\limits_{k=1}^{K} e^{z_k}} \qquad j = 1, \ldots, K \tag{6.5}$$

where $\mathbf{z}$ is the input from the previous layer, and j references the single output units—in our case $K = 9$ since we have nine categories.

Now that the model has been described and the dataset is in the final form, we can proceed with the training. Is common practice in model training to use validation method for parameter tuning and performance evaluation. The most commonly used validation method is **k-fold Cross Validation**. The dataset is split in k folds, and the training is repeated k time. Each time, a different combination of $k-1$ folds is used for training, and the left-out k-th fold is used as unseen data to test the performances. Repeating the process k times results in using each fold at least once for training and exactly once for validation. This helps reduce overfitting the model on the training data, that could result in excellent training accuracy and low test accuracy. Overfitting can also be limited introducing a simple regularization techniques called **early stopping**. The training of a NN is an operation repeated several times, and each time the weights are updated and adjusted to reach a configuration that provides the desired result. Each iteration is called **epoch**, and for complex models and data the training usually involves an high numbers of epochs, in the order of hundreds. Yet, many times the model reaches a stable state after lesser epochs; after this point, more training epochs not only incur in westing a large amount of time in keeping on training, but also leads the model into learning too much from the data we use for training that it gets overfitted. Early stopping allows the training to be halted when the classification accuracy remains steady for a certain amount of epochs, assuming that no more benefit would come out of keeping on training further.

CHAPTER 7

PERFORMANCE EVALUATION

We have seen throughout Chapters 5 and 6 which are the different motivations that led us to the final model, detailed in **??**. Related work and similar research, discussed in Chapter 2, has firstly led us towards Deep Learning. We saw back in Section 2.3 how different Machine Learning approach on both the model and the features set affect performances significantly.

In Section 5.2 we saw summarized in Table VII which are the differences in terms of accuracy for different methods and approach; this data is represented in Figure 17. Since we choose to run the classification of both Semantic Location and Activity Recognition, for the reasons discussed in Chapter 5 and supported by the work in Section 2.3, we can see how Deep Learning is not only the best performing model overall—along with Decision Trees, but that it is the one that best responds to the combined classification, for the reasons explained in Section 5.3.

Figure 17 showes how Decision Trees are the ones who suffer the most when classifying over the entire dataset. When we overviewed classification models in Section 5.2, we saw that DTs suffer significantly when the dimensionality is high. Also, a linear model like Logistic Regression has an hard time trying to give sense to complicated data like time series sensor data: its accuracy in Activity Recognition is significantly lower than in Semantic Location detection. Overall, all the models, except DTs, perform better when the dataset is combined

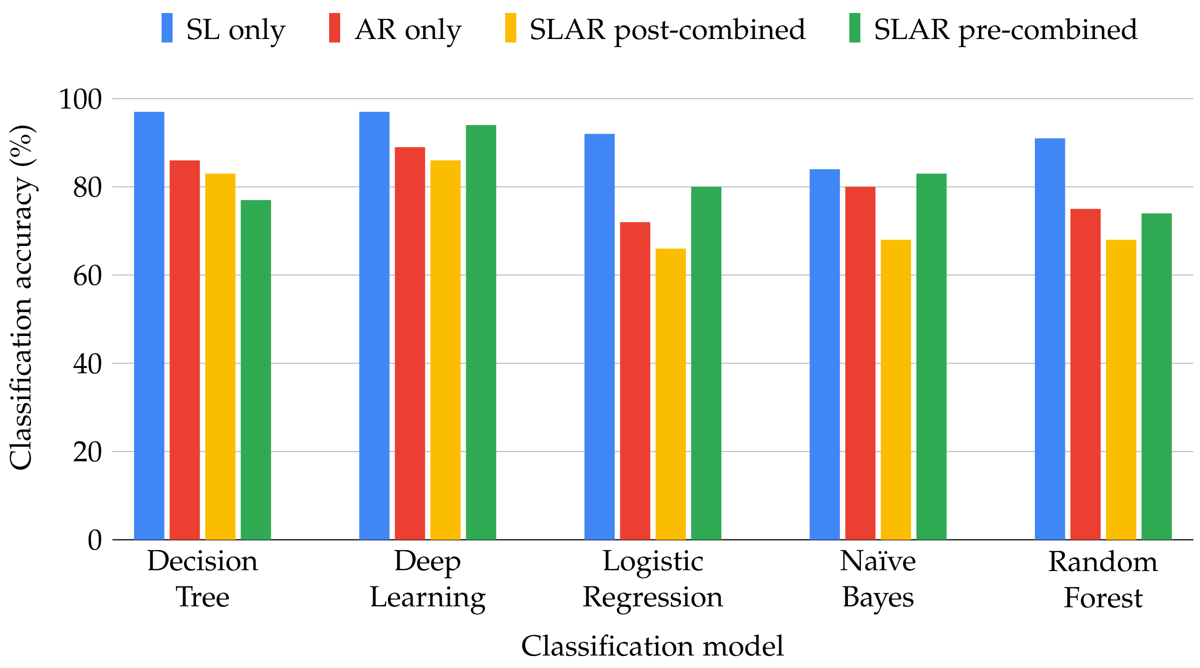


Figure 17: Overview of different classification models performances in different settings.

before the classification, while combining the results after makes the overlap of type-I and type-II errors induce in significantly lower accuracy.

7.1 Model accuracy

The main goal now is to determine how well the model performs. This is a necessary step to understand if the model we have been training really responds to the problem and how. The performance of a model are usually tested both on training data and unseen data. The reason why we firstly look at training data alone, is one important point in model training and evaluation. The purpose of having well separated training and test data is that we want

TABLE IX: CONFUSION MATRIX FOR COMBINED SEMANTIC LOCATION AND ACTIVITY RECOGNITION ON TRAINING DATA.

Class: Actual (down) v. Predicted (across)	**Indoor biking**	**Indoor running**	**Indoor stationary**	**Indoor walking**	**In vehicle**	**Outdoor biking**	**Outdoor running**	**Outdoor stationary**	**Outdoor walking**
Indoor biking	**99 %**	0	1 %	0	0	0	0	0	0
Indoor running	1 %	**97 %**	0	2 %	0	0	0	0	0
Indoor stationary	0	0	**97 %**	2 %	0	0	0	0	0
Indoor walking	0	0	4 %	**94 %**	0	0	0	1 %	1 %
In vehicle	0	0	1 %	0	**99 %**	0	0	0	0
Outdoor biking	0	0	0	0	0	**99 %**	0	1 %	0
Outdoor running	0	0	0	0	0	0	**98 %**	0	2 %
Outdoor stationary	0	0	1 %	2 %	0	2 %	0	**95 %**	1 %
Outdoor walking	0	0	0	2 %	0	0	0	1 %	**97 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class. Evaluated over 10-fold CV.

to *leave out* test data so that we can consider them as **unseen**, as will very likely be most of the future data that we will come across after deployment.

The reason is trivial, yet is a very common error in Machine Learning tasks. Training and testing the model on the entire data available leaves us without a real clue of how the model will actually behave in the future, and will significantly *bias* the classifier towards the training

TABLE X: CUMULATIVE CONFUSION MATRIX FOR INDOOR/OUTDOOR ON TRAINING DATA.

Class: Actual (down) v. Predicted (across)	**Indoor**	**Outdoor**
Indoor	**99 %**	1 %
Outdoor	2 %	**98 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class. Evaluated over 10-fold CV.

data. This is the same reason why we introduced **Cross Validation** in the first place; other validation methods, like the *validation set* approach, continuously test the model against the validation data, allowing information from it to *"leak"* in the model training each time the training process is reproduced.

As a first step, then, we evaluate the model during the training phase through 10-fold Cross Validation as explained in **??**. Most common performance measures in classification are derived from four values, defined for each category of the classification problem:

true positives is the count of instances of the class A_i correctly classified;

true negatives is the count of instances *not* of the class A_i correctly *classified not in the class* A_i;

false positives is the count of instances *not* of the class A_i misclassified in the class A_i;

false negatives is the count of instances of the class A_i misclassified as not of the class A_i.

From this measures, the accuracy is defined as

$$\begin{aligned}\text{Accuracy} &= \frac{\text{\# correct predictions}}{\text{\# total predictions}} \\ &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{TN}}\end{aligned} \tag{7.1}$$

It is particularly worth mentioning that this measure is more than often misleading. If we were to build a model to diagnose rare diseases, an high accuracy is not enough as the model could be extremely good at telling healthy conditions but just not useful in diagnosing those rare cases of diseases. This is why accuracy is not a good measures for classification when the real-world distribution of the data is highly class-imbalanced.

Another simple, yet often more indicative measure is the **precision**. The precision is defined for each class of the problem, as

$$\begin{aligned}\text{Precision} &= \frac{\text{\# actual positive}}{\text{\# classified positive}} \\ &= \frac{\text{TP}}{\text{TP + FP}}\end{aligned} \tag{7.2}$$

From the definition in Equation (7.2), follows that the values along the diagonals in the confusion matrices, marked in bold in Tables IX to XIV, are the precision scores of the relative class.

TABLE XI: CUMULATIVE CONFUSION MATRIX FOR ACTIVITY RECOGNITION ON TRAINING DATA.

Class: Actual (down) v. Predicted (across)	**Biking**	**Running**	**Stationary**	**Walking**	**In vehicle**
Biking	**99 %**	0	1 %	0	0
Running	1 %	**97 %**	0	2 %	0
Stationary	0	0	**98 %**	2 %	0
Walking	0	0	2 %	**98 %**	0
In vehicle	0	0	1 %	0	**99 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class. Evaluated over 10-fold CV.

Let us now take a look at the results tabulated in Tables IX to XIV. The first three confusion matrices report training performances; the last three are derived from testing the model on unseen data. The first obvious observation is that precision values on unseen data in Tables XII to XIV are averagely lower than their relative values on training in Tables IX to XI. We expect this: the high dimensionality of the dataset, and the high variation range of each predictor, along with the extremely random nature of the variables, thus so that unseen data can be significantly different from the training data. We will address more this problem in Chapter 8. What is important is that classification on unseen data is overall not extremely

TABLE XII: CONFUSION MATRIX FOR COMBINED SEMANTIC LOCATION AND ACTIVITY RECOGNITION ON UNSEEN DATA.

Class: Actual (down) v. Predicted (across)	**Indoor biking**	**Indoor running**	**Indoor stationary**	**Indoor walking**	**In vehicle**	**Outdoor biking**	**Outdoor running**	**Outdoor stationary**	**Outdoor walking**
Indoor biking	**97 %**	0	2 %	1 %	0	0	0	0	0
Indoor running	0	**93 %**	2 %	1 %	0	0	0	0	3 %
Indoor stationary	0	2 %	**87 %**	9 %	0	1 %	0	0	0
Indoor walking	2 %	0	11 %	**79 %**	2 %	0	0	0	2 %
In vehicle	0	2 %	3 %	1 %	**94 %**	0	0	0	0
Outdoor biking	0	0	0	0	0	**72 %**	2 %	2 %	24 %
Outdoor running	0	0	0	0	0	9 %	**90 %**	0	1 %
Outdoor stationary	0	0	4 %	14 %	0	0	0	**81 %**	1 %
Outdoor walking	0	0	0	2 %	0	2 %	0	3 %	**93 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class.

TABLE XIII: CUMULATIVE CONFUSION MATRIX FOR INDOOR/OUTDOOR ON UNSEEN DATA.

Class: Actual (down) v. Predicted (across)	**Indoor**	**Outdoor**
Indoor	**99 %**	1 %
Outdoor	5 %	**95 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class.

worse than the one on training, letting us think that the model should not have an high level of overfit.

Overall, classification accuracy reaches good values with this algorithm, but it is worth to take a look at particular cases that can help us understand potential problems, not only in the model but in the overall approach to the work. Although training accuracy in Tables IX to XI is quite stable across all classes, we can spot some faults on unseen data.

For instance, the class *outdoor biking* in Table XII has a significantly lower precision than the others, as low as 72 %, and the largest majority of misclassified instances mistakenly falls in the category *outdoor walking*. This behavior can also be seen from the cumulative confusion matrix for AR, in Table XIV: the *biking* class reports the lowest precision, with a high share of instances misclassified as *walking*.

TABLE XIV: CUMULATIVE CONFUSION MATRIX FOR ACTIVITY RECOGNITION ON UNSEEN DATA.

Class: Actual (down) v. Predicted (across)	**Biking**	**Running**	**Stationary**	**Walking**	**In vehicle**
Biking	**86 %**	1 %	2 %	11 %	0
Running	3 %	**97 %**	0	1 %	0
Stationary	0	2 %	**90 %**	8 %	0
Walking	2 %	0	8 %	**89 %**	1 %
In vehicle	0	2 %	3 %	1 %	**94 %**

Confusion matrix reports the percentage of classified samples per class and not the absolute count; values in bold along the diagonal are the precision of each class.

It is not particularly easy neither to understand where and why a ML algorithm fails, nor which might be the problem in the data. After all, ML moved towards complex models like Deep Learning to achieve tasks that we do not necessarily understand: most of the times, complex DL models are nothing more than a *black box* to us, as is the human brain. What we can hypothesize based on our human knowledge of the problem reality, is that even if *biking* as a different motion *"fingerprint"* than *walking*—see Figures 3 (a) and 5 (a)—their paces, that we can think of as "frequencies", are actually quite similar, when we bike and walk at a regular pace. This is significantly different than *running* instead. We do not have at the moment any significant research literature to support this hypothesis, as target classes vary from work to work.

It is hard to tell why this low precision for the *biking* class occurs only for *outdoor*—cfr. Table XII; the only clue coming from real word knowledge is that indoor biking does not suffer much noise in motion data as an indoor bike is steady—outdoor biking, instead, suffers from bike and road condition. Also, the two activities are more different than what it might seem: in indoor biking, unlike other activities, the user is not subject to an actual acceleration, except for the one whom the device only is subject to; in other words, the subject is moving on the bike, but the bike is not.

All these and other potential instability of the work not only can relate to the real world scenario, but also on a vast number of other factors, as data quality, quantity and variability. Any further assumption to be made would require a more precise analysis of the data with a significantly larger quantity of diverse data as to assert the extents to which this model is applicable to and any eventually derivable improvement.

Another observation worth mentioning, is that the overall *outdoor* classification precision drops from 98 % during training (Table X) to 95 % on unseen data (Table XIII). Although the problem of Semantic Location recognition seems the easier one, given its higher overall performances in Table VII and Figure 17, it gets slightly more tricky on unseen data for the *outdoor* class; this might be due to the fact that outdoor environments vary much more from one another than indoor ones; the high number of variables—ambient noise, ambient light, geomagnetic disturbance, and cellular signal strength—seem to have more variability outdoor than indoor.

7.2 Model deployment on mobile

We have introduced TensorFlow back in Sections 1.6 and 6.1.1, motivating its choice by its ability to be deployed on mobile Android application through the Android TensorFlow Mobile framework. We know from Section 1.6.1 that the TensorFlow team also developed TensorFlow Lite, a very lightweight yet efficient version of TensorFlow for mobile applications that has smaller binary size and supposedly better performances. Yet, as of September 2018, TensorFlow Mobile is still a developer preview, and only supports a smaller subset of operators.[1] This means that not all models are yet portable to the Lite framework, as our is not, and we will have to use the regular Mobile version for the time being. We will see that fortunately this does not make a great impact as the inference in the application will have extremely low power consumption.

Figure 18 shows the abstraction of the application structure. The raw sensor data collection at the lower level works exactly like we saw it for collecting the data for the dataset itself, back in Chapter 3—except now we can avoid collecting positioning data that we discovered counterproductive in Section 3.3.

The data preprocessing step follows what we saw in Section 6.2. Every time a new record is generated, it wrapped with a timestamp and is pushed in a queue:

[1]From: `tensorflow.org/mobile`, visited September 2018.

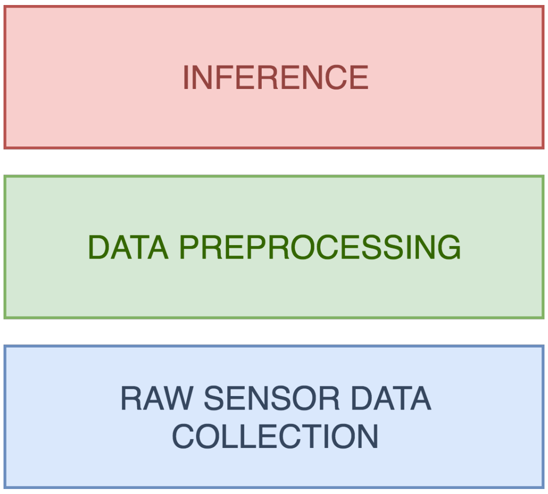


Figure 18: Application Structure.

```
private void pushRecord(
        @NonNull float[] record,
        @NonNull RobustScaler robustScaler);
```

Inside this method the record gets preprocessed to make the data compliant to the preprocessing explained in Section 6.2—note that the accelerometer values correction explained in Section 3.2 is performed in the same way as soon as the record is generated. The feature scaling described in Section 6.2 is performed by a `Scaler` class. Then, inside a critical section, whenever the queue of generated records reaches the defined window width (see Section 5.4.2), a new window instance is built and the records that are no longer needed are removed:

```
drainSemaphore.acquireIninteruptibly();
if (generatedRecords.size() >= CLASSIFICATION_WINDOW_LENGTH) {
  ArrayList<FloatArrayWrapper> topK = generatedRecords
      .stream()
      .limit(CLASSIFICATION_WINDOW_LENGTH)
      .collect(Collectors.toCollection(ArrayList::new));
}
for (int i  = 0;
    i < (SLIDE_CLASSIFICATION_WINDOW ? CLASSIFICATION_WINDOW_SLIDING :
       CLASSIFICATION_WINDOW_LENGTH);
    generatedRecords.poll(), i ++);
```

A `SlarClassifier` class performs the classification on the generated window; the result is displayed.

```
SlarClassifier.PredictionResult predictionResult = slarClassifier
  .predictWindow(topK.stream()
    .map(FloatArrayWrapper::asArray)
    .collect(Collectors.toList()));
if (predictionResult != null)
  runOnUiThread(() -> textView.setText(predictionResult.toString()));
drainSemaphore.release();
```

Of course now the relevant part is how the `SlarClassifier` is implemented to run the model generated in Section 6.2.1. Once we have trained the TensorFlow model, we need to obtain a **frozen version**. The application must be instructed to load the TensorFlow libraries in the *gradle* file, and importing them into the class:

```
import org.tensorflow.Tensor;
import org.tensorflow.TensorFlow;
import org.tensorflow.contrib.android.TensorFlowInferenceInterface;
```

Then we can use the libraries inside the `SlarClassifier` class. The model will be loaded into an `TensorFlowInferenceInterface` object:

```
private TensorFlowInferenceInterface inferenceInterface;
private String[] labels;
SlarClassifier(@NonNull final Context context) {
  inferenceInterface = new TensorFlowInferenceInterface(
    context.getAssets(), MODEL_NAME
  );
  labels = loadLabels(context);
}
```

Once the classifier is instantiated, the inference on a classification window is run as follows:

```
public PredictionResult predictWindow(
      @NonNull final float[] flatWindow) {
  long startTime = System.nanoTime();
  float[] confidences = new float[OUTPUT_SIZE];
  inferenceInterface.feed(INPUT_NODES, flatWindow, INPUT_SIZE);
  inferenceInterface.run(OUTPUT_NODES);
  inferenceInterface.fetch(OUTPUT_NODE, confidences);
  return new PredictionResult(confidences, System.nanoTime() -
     startTime);
}
```

where `PredictionResult` is a declared inner class to represent the result of a classification, along with its run time, providing methods to access it. Table XV summarizes the two classes supporting the inference task, `SlarClassifier` and `PredictionResult`.

TABLE XV: DOCUMENTATION FOR THE SLAR CLASSIFIER CLASS.

Type	Field/Method and Description
	`class SlarClassifier`
`float`	**`CONFIDENCE_THRESHOLD`**
`int`	**`FEATURE_SIZE`** The dimensionality (number of features) of the records to be classified; must match the output `Tensor` dimension.
	`inferenceInterface`
`String`	**`INPUT_NODES`** The name of the input `Tensor` of the model.
`long[]`	**`INPUT_SIZE`** An array representing the input dimension of a classification instance, nominally: 1, `WINDOW_SIZE`, `FEATURE_SIZE`.
`String`	**`LABEL_FILE_NAME`** The name of the asset containing the list of labels in the order they are output from the model.
`String[]`	**`labels`** The names of the different classes.
`String`	**`MODEL_NAME`** The name of the frozen graph `.pb` file representing a TensorFlow model.
`String`	**`OUTPUT_NODE`** The suffix of the nodes in the output `Tensor`.
`String[]`	**`OUTPUT_NODES`** A list of the suffices of the nodes in the output `Tensor`.
`int`	**`OUTPUT_SIZE`**

Continues on next page

Continues from previous page

Type	Field/Method and Description
	The dimension of the output `Tensor`, corresponding to the number of classes.
`class`	**`PredictionResult`**
`int`	**`WINDOW_SIZE`** The width, or length, of the classification window, if wider, or longer, than a single record, 1 otherwise.
`String[]`	**`loadLabels`**`(Context context)`
`PredictionResult`	**`predictWindow`**`(Collection<float[]> collection)` Feeds the given record to the inference model and produces a classification result.
`PredictionResult`	**`predictWindow`**`(float[] flatWindow)` Runs the inference on the provided flat window with a `TensorFlowInferenceInterface` and produces a classification result.
`float[]`	**`toFlatArray`**`(Collection<float[]> collection, int... dimensions)` Returns a flattened array of the given `Collection`, reshaping according to the given `dimensions`.
	`class SlarClassifier.PredictionResult`
`float[]`	**`confidences`** Vector of the confidences for each class from this classification result.
`Optional<String>`	**`inferenceTime`**
`float[]`	**`getConfidences`**`()`
`String`	**`getInferenceTime`**`()`

Continues on next page

Continues from previous page

Type	Field/Method and Description
`int`	**`getMostConfidentActivity`**`()`
`boolean`	**`hasMajorityClass`**`()`
`boolean`	**`hasMajorityClass`**`(float threshold)`
`boolean`	**`hasClassificationResult`**`()`
`String`	**`toString`**`()`
`String`	**`toStringInspect`**`()`

Ends from previous page

7.3 Time, memory and power efficiency

Since the application runs on a mobile device and possibly all day long, for the purposes explained in Section 1.7, it is important to analyze its computational performances and energy consumption requirements.[2] The inference time of a TensorFlow Mobile module is extremely optimized by the TensorFlow library. Execution time on mobile is important not only because of its repercussions of the non-idle time of the application resulting in more power consumption, but also because most average category smartphones have lower execution power than computers' CPUs; a longer inference time may slow down the application itself and the other device's services and applications. The application performs the following main tasks:

(i) reads sensor values;

(ii) runs inference;

(iii) shows/saves inference results.

The first and last tasks are very fast and low power operations executed all the times; moreover, all the sensors whose values are used for the classification, are continuously produced independently from the application requests—except for the ambient sound amplitude discussed in Section 3.7. With the TensorFlow Mobile optimizations, the inference of the model produced in Chapter 6 is on average around 11.2 ms, with a minimum of 4.8 ms and a max-

[2]All hardware-dependent values in this Section refer to measurements performed on a OnePlus 6 device running Android 8.1 Oreo with API level 27. All measurements might vary on other devices or different OS versions.

imum of 17.9 ms, over a sample of 400 records. Of course the computation time depends on how often the inference is run; in our case, the test application run the inference each second, that with a sampling frequency of 25 Hz totals 4 samples per second. The chosen classification window is 12 samples wide (3 s).

In terms of memory, the application size is extremely contained. The TensorFlow Mobile optimized model is a file of less than 300 kB; the overall applications with all its *assets* (model file, a file containing the list of labels, and a file containing values for the data normalization—the last two pieces of information could be hard-coded into the application source) is 50 MB and uses a little overhead of less than 200 kB to save the current state (activity and location labels) to reload when starting, which is only necessary when labeling data is necessary, so not in deployment mode. Of course the application produces data records. At a production rate of one record per second, the record storage file grows by 8.8 kB/h.

As for power consumption, the application is extremely power-efficient due to the low power consumptions of the sensors explained throughout Chapter 3. It is difficult to define a long-term average battery consumption of the application, as the Android OS battery management module reports a power consumption of less than 5% during a complete battery cycle (from fully charged to power-off).

CHAPTER 8

FUTURE WORK

Ubiquitous and Pervasive Computing are not newborn concepts nor are they anywhere close to the full of their possibilities. Internet connection is making its way into every physical component of our everyday life, as we said back in Section 1.1. All branches of Ubiquitous Computing still have a long way to go and no work can be said complete nor comprehensive as new technologies, techniques, and concepts arise on a daily basis.

What we have achieved so far is a *skeleton* that carries out a non trivial task in a quite trivial way—even if the model itself might be complicated, the workflow is quite straightforward and focused on what we could define the most obvious aspects.

On the other end, there are several techniques that can be incorporated to achieve even better results, both changing or extending the knowledge base of the Machine Learning module. Also, not only this module can be further improved and adapted, it can also gain performance and reach a larger number of applications when scaled for multiple platforms.

Least, but not last, the module can be adapted to the user needs in terms of quality and performances. Let us take a closer look to some of the most straightforward continuations of this work.

(i) First and foremost, it is crucial that the model gets extended to all devices. This might turn out to be less trivial then what we might think lacking access to a large variety of devices. As we explained in Chapter 3 and mentioned other times throughout this work, different makes and models of devices, and sometime even different units of the same model, mount different make and models of sensors. This is a huge deal, just remember from Section 4.1 that Android is actively used by over two billion devices, as of May 2017, leading to a potentially very large variety in sensor data.

For a model like this to be robust and reliable platform-wise, one has to analyze massive quantities of data coming from a variety of devices and engineer a solution based on how much this data actually varies among different devices. At that point, one simple solution might be to just toughen the model training on as much data as possible coming from different sources; another solution would be to engineer a mapping that brings "outcast" sensor readings similar to the others. Less feasible would be to have a different model for different platforms, but this could really be a necessity in the case some devices fail in being able to produce a dataset close enough to the chosen one.

(ii) Even if the model performs quite well on this specific test case, its production performances can be unpredictably different. To enhance the classification accuracy there are different techniques that can be adopted, either separately or in conjunction.

  (a) Common place detection: starting with the basics, the algorithm can be supported by a module that recognizes frequently visited places, either user-annotated (e.g. home, work, gym) or statistically inferred from either the user or a crowd of users (e.g. grocery store, park, beach). The knowledge about the environment is part of the Semantic Location and will very likely support the indoor/outdoor detection, but we saw back in Chapter 7 that the two tasks very likely also support each other. It is important to notice that this common place detection would require the use of positioning data that we excluded from our data, with possible repercussions on power consumption—see Section 3.3.

(b) The environment is not necessarily a place, and does not necessarily need traditional positioning systems to be recognized. One example is the easy detection of the "in vehicle" mode when the device is connected to a car whose on board *infotainment* is equipped with the Android Auto firmware.[1] The OS provides easy means to discover if the device is connected to a supporting car environment. Another example is detecting when the device is *casting* its display content to the home TV. This last example needs further investigation on wether any application would be allow to access this information.

(c) One other step to improve accuracy is to progressively train the model further with user-annotated data. This is a technique that already exists in some commercial application. For example, Google Maps uses it to improve Semantic Location and Activity Recognition allowing the user to confirm or correcting places and activity in their personal timeline—a Google Maps tool that tracks user activities and positioning throughout the day. Also, this tool that the user can enable in their Google Maps application, can be directly exploited to support this model; again, a

[1] *Infotainment* is a *portmanteau* word indicating a device, usually equipped with a display, that provides both information and entertainment, commonly used to indicate car decks.

deeper investigation is needed to understand to what extent this information can be accessed outside Google's map application.

(iii) Another option for further development is multi-platform integration. As said multiple times, Ubiquitous Computing has made its way into a lot of devices and many of them now run an Android-based distribution. These device provide further insightful data; beside Android Auto mentioned before, we have available an Android Wear distribution for smartwatches that not only produce more motion data, but also new type of information like heartbeat rate or body temperature, depending on the device. This data can be extremely insightful for Activity Recognition tasks.

Besides wearable devices, there is a variety of other devices with potentially useful information like Google Home, a home personal assistant that, among the plethora of information that collects from the user interaction with it, can easily tell if the user is inside is house or workplace wherever one of these devices is installed.

Further research headed in this direction should consider not only data variability as mentioned before, but also to engineer a solution that can work independently on wether this data is available or not at every different moment. The actual type of data produced by wearable and home devices has to be further investigated.

(iv) As most commercial products, it would be a possible improvement to give the user the possibility to customize the behavior of the application choosing a custom trade-off between accuracy, privacy, and power efficiency, which as discussed is a major issue for mobile devices. The user could be allowed to choose wether to enable location services, or if to allow the microphone to capture noise level for privacy concerns. On most Android devices the user already has some degree of control, for example on the accuracy to power efficiency trade-off for location services.

CHAPTER 9

CONCLUSION

The most important achievement of this work is to have proven that there is a solid way of implementing a complex task on mobile devices, exploiting their possibilities in data collection and more-than-sufficient computation power. We follow a less commonly chosen path in research in both activity recognition and microenvironment detection that gets rid of *ad-hoc*, invasive, sometimes expensive sensor in favor of low-power, easily accessible on-board sensors.

We exploite most of the sensors available on common Android devices and some other non-properly sensors that can provide insightful information. We collect labeled data for various activities and environment and set up a machine learning model that can handle it. We chose a less common variation of Artificial Neural Networks called Long Short-Term Memories that were specifically thought to better handle timeseries data.

We use motion sensors (accelerometer and gyroscope) to address activity recognition, and several other data to infer indoor and outdoor: light and proximity, magnetic field sensor, cellular radio signal strength, and microphone. We then perform some feature extraction and selection to address dimensionality. We perform data cleaning and preprocessing and proceed to learn and test the model with both *k*-fold cross validation.

Deep Learning technologies like the TensorFlow framework help us build a model and modify it easily to respond well enough to the task. It provides the possibility to easily train and modify the model on the fly on a workstation or cloud, with both CPU and GPU support. Once the definitive model is ready, it allows to export it into a representation that can be deployed on mobile. On the Android device, a TensorFlow Mobile library handles the inference run task on the data fed to it.

We achieved significantly high result on test data, easiliy comparable to the most high result obtained in more pervasive and expensive ways in the related research literature. On the mobile side, we achive an extremely lightweight application with extremely low power and memory consumption. The module can be easily incorporated as a model in a larger application.

Privacy issues have been limited as much as possible; the ability of running the inference directly on mobile allows the data to never leave the device, without even the need of storing, not even momentarily, the sensors reading on the device, limiting both data leaks and memory usage.

CITED LITERATURE

[1] *Computer Games I*. Springer New York, 2011. ISBN 1461387183. URL `https://www.ebook.de/de/product/19298247/computer_games_i.html`.

[2] United states code. United States Government Publishing Office, 2011. URL `www.gpo.gov/fdsys/granule/USCODE-2011-title47/USCODE-2011-title47-chap5-subchapII-partI-sec222`. Supplement 5, Title 47, Chapter 5, Subchapter II, Part I, Section 222, Privacy of customer information.

[3] World Health Organization (WHO). 7 million premature deaths annually linked to air pollution. World Health Organization Media centre, March 2014. URL `http://www.who.int/mediacentre/news/releases/2014/air-pollution/en/`. Accessed June 2018.

[4] World Health Organization. Ambient air pollution: A global assessment of exposure and burden of disease. *WHO Library Cataloguing-in-Publication Data*, 2016. ISSN 978 92 4 151 1135 3. URL `http://apps.who.int/iris/bitstream/handle/10665/250141/9789241511353-eng.pdf;jsessionid=A3A9866B2F7B5122055231159EE86ADE?sequence=1`.

[5] Ong Chin Ann and Lau Bee Theng. Human activity recognition: A review. In *2014 IEEE International Conference on Control System, Computing and Engineering (ICCSCE 2014)*. IEEE, nov 2014. doi: 10.1109/iccsce.2014.7072750.

[6] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Action classification in soccer videos with long short-term memory recurrent neural networks. In *Artificial Neural Networks – ICANN 2010*, pages 154–159. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-15822-3_20.

[7] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Sequential deep learning for human action recognition. In *Lecture Notes in Computer Science*, pages 29–39. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-25446-8_4.

[8] Ling Bao and Stephen S Intille. Activity recognition from user-annotated acceleration data. In *International Conference on Pervasive Computing*, pages 1–17. Springer, 2004.

[9] Christian Becker and Frank Durr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9(1):20–31, aug 2004. doi: 10.1007/s00779-004-0270-2.

[10] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[11] Ryan Block. Google is working on a mobile os, and it's due out shortly, August 2007. URL `https://www.engadget.com/2007/08/28/google-is-working-on-a-mobile-os-and-its-due-out-shortly/`. Accessed July 9, 2018.

[12] Carl R Boyd, Mary Ann Tolson, and Wayne S Copes. Evaluating trauma care: the triss method. trauma score and the injury severity score. *The Journal of trauma*, 27(4):370–378, 1987.

[13] Zhenyu Chen, Yiqiang Chen, Shuangquan Wang, and Zhongtang Zhao. A supervised learning based semantic location extraction method using mobile phone data. In *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*. IEEE, may 2012. doi: 10.1109/csae.2012.6273012.

[14] Saisakul Chernbumroong, Shuang Cang, Anthony Atkins, and Hongnian Yu. Elderly activities recognition and classification for applications in assisted living. *Expert Systems with Applications*, 40(5):1662–1674, apr 2013. doi: 10.1016/j.eswa.2012.09.004.

[15] Jaewoo Chung, Matt Donahoe, Chris Schmandt, Ig-Jae Kim, Pedram Razavai, and Micaela Wiseman. Indoor location sensing using geo-magnetism. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 141–154. ACM, 2011.

[16] Stefan Dernbach, Barnan Das, Narayanan C Krishnan, Brian L Thomas, and Diane J Cook. Simple and complex activity recognition through smart phones. In *Intelligent Environments (IE), 2012 8th International Conference on*, pages 214–221. IEEE, 2012.

[17] Richard W DeVaul and Steve Dunn. Real-time motion classification for wearable computing applications. *2001 Project Paper*, 2001.

[18] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5 (1):4–7, February 2001. doi: 10.1007/s007790170019.

[19] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[20] Ronald A Fisher. The statistical utilization of multiple measurements. *Annals of eugenics*, 8(4):376–386, 1938.

[21] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[22] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[23] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2017. ISBN 0262035618. URL `https://www.ebook.de/de/product/26337726/ian_goodfellow_yoshua_bengio_aaron_courville_deep_learning.html`.

[25] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, may 2013. doi: 10.1109/icassp.2013.6638947.

[26] Tao Gu, Zhanqing Wu, Xianping Tao, Hung Keng Pung, and Jian Lu. epSICAR: An emerging patterns based approach to sequential, interleaved and concurrent activity recognition. In *2009 IEEE International Conference on Pervasive Computing and Communications*. IEEE, mar 2009. doi: 10.1109/percom.2009.4912776.

[27] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] Google Inc. Build and train machine learning models on our new google cloud tpus. Google Blog, May 2017. URL `www.blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning`. Accessed June 2018.

[30] Google Inc. Cloud tpu machine learning accelerators now available in beta. Google Cloud Platform Blog, February 2018. URL `cloudplatform.googleblog.com/2018/02/Cloud-TPU-machine-learning-accelerators-now-available-in-beta.html`. Accessed June 2018.

[31] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of Machine Learning and Data Mining*, pages 314–315. Springer US, 2017. doi: 10.1007/978-1-4899-7687-1_192.

[32] Kourosh Khoshelham. Accuracy analysis of kinect depth data. In *ISPRS workshop laser scanning*, volume 38, page W12, 2011.

[33] Eunju Kim, Sumi Helal, and Diane Cook. Human activity recognition and pattern discovery. *IEEE Pervasive Computing*, 9(1):48–53, jan 2010. doi: 10.1109/mprv.2010.7.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[35] Nicholas D Lane, Mashfiqui Mohammod, Mu Lin, Xiaochao Yang, Hong Lu, Shahid Ali, Afsaneh Doryab, Ethan Berke, Tanzeem Choudhury, and Andrew Campbell. Bewell: A smartphone application to monitor, model and promote wellbeing. In *5th international ICST conference on pervasive computing technologies for healthcare*, pages 23–26, 2011.

[36] D. Li and D. L. Lee. A lattice-based semantic location model for indoor navigation. In *Proc. Ninth Int. Conf. Mobile Data Management (mdm 2008)*, pages 17–24, April 2008. doi: 10.1109/MDM.2008.11.

[37] Dandan Li and Dik Lun Lee. A topology-based semantic location model for indoor applications. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems - GIS '08*. ACM Press, 2008. doi: 10.1145/1463434.1463443.

[38] Mo Li, Pengfei Zhou, Yuanqing Zheng, Zhenjiang Li, and Guobin Shen. IODetector. *ACM Transactions on Sensor Networks*, 11(2):1–29, dec 2014. doi: 10.1145/2659466.

[39] Juhong Liu, O. Wolfson, and Huabei Yin. Extracting semantic location from outdoor positioning systems. In *Proc. 7th Int. Conf. Mobile Data Management (MDM'06)*, page 73, May 2006. doi: 10.1109/MDM.2006.87.

[40] Jun Liu, Amir Shahroudy, Dong Xu, and Gang Wang. Spatio-temporal LSTM with trust gates for 3d human action recognition. In *Computer Vision – ECCV 2016*, pages 816–833. Springer International Publishing, 2016. doi: 10.1007/978-3-319-46487-9_50.

[41] Marcus Liwicki, Alex Graves, Santiago Fernàndez, Horst Bunke, and Jürgen Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proceedings of the 9th International Conference on Document Analysis and Recognition, ICDAR 2007*, 2007.

[42] Robert Love. Why does gps use so much more battery than any other antenna or sensor in a smartphone? Quora.com, July 2013. URL `www.quora.com/Why-does-GPS-use-so-much-more-battery-than-any-other-antenna-or-sensor-in-a-smartphone`. Accessed June 2018.

[43] G. Milette and A. Stroud. *Professional Android Sensor Programming*. ITPro collection. Wiley, 2012. ISBN 9781118240458. URL `https://books.google.com/books?id=dZjo-254FucC`.

[44] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997. ISBN 0-07-042807-7. URL `https://www.amazon.com/Machine-Learning-Tom-M-Mitchell/dp/0070428077?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0070428077`.

[45] B. Najafi, K. Aminian, A. Paraschiv-Ionescu, F. Loew, C.J. Bula, and P. Robert. Ambulatory system for human motion analysis using a kinematic sensor: monitoring of daily physical activity in the elderly. *IEEE Transactions on Biomedical Engineering*, 50(6):711–723, jun 2003. doi: 10.1109/tbme.2003.812189.

[46] Alfred Ng. Google's android now powers more than 2 billion devices, May 2017. URL `https://www.cnet.com/au/news/google-boasts-2-billion-active-android-devices/`. Accessed July 2018.

[47] Anh Tuan Nghiem, Edouard Auvinet, and Jean Meunier. Head detection using kinect camera and its application to fall detection. In *2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA)*. IEEE, jul 2012. doi: 10.1109/isspa.2012.6310538.

[48] Donald J. Patterson, Lin Liao, Dieter Fox, and Henry Kautz. Inferring high-level behavior from low-level sensors. In *UbiComp 2003: Ubiquitous Computing*, pages 73–89. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-540-39653-6_6.

[49] Valentin Radu, Panagiota Katsikouli, Rik Sarkar, and Mahesh K. Marina. A semi-supervised learning approach for robust indoor-outdoor detection with smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems - SenSys '14*. ACM Press, 2014. doi: 10.1145/2668332.2668347.

[50] Alvin Raj, Amarnag Subramanya, Dieter Fox, and Jeff Bilmes. Rao-blackwellized particle filters for recognizing activities and spatial context from wearable sensors. In *Experimental Robotics*, pages 211–221. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-77457-0_20.

[51] Lenin Ravindranath, Calvin Newport, Hari Balakrishnan, and Samuel Madden. Improving wireless network performance using sensor hints. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 281–294, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1972457.1972486`.

[52] Daniele Riboni and Claudio Bettini. Context-aware activity recognition through a combination of ontological and statistical reasoning. In *Ubiquitous Intelligence and Computing*, pages 39–53. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-02830-4_5.

[53] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[54] Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779, mar 2007. doi: 10.1162/neco.2007.19.3.757.

[55] Shai Ben-David Shai Shalev-Shwartz. *Understanding Machine Learning*. Cambridge University Pr., 2014. ISBN 1107057132. URL `https://www.ebook.de/de/product/22370006/shai_shalev_shwartz_shai_ben_david_understanding_machine_learning.html`.

[56] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[57] Leon O. Stenneth. *Human Activity Detection Using Smartphones and Maps*. phdthesis, University of Illinois at Chicago, College of Engineering, Department of Computer Science, December 2013. URL `http://hdl.handle.net/10027/11247`.

[58] Kristof Van Laerhoven and Ozan Cakmakci. What shall we teach our pants? In *Wearable Computers, The Fourth International Symposium on*, pages 77–83. IEEE, 2000.

[59] Vivek Veeriah, Naifan Zhuang, and Guo-Jun Qi. Differential recurrent neural networks for action recognition. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 4041–4049. IEEE, 2015.

[60] James Vincent. Google's new machine learning framework is going to put more ai on your phone. The Verge, May 2017. URL `www.theverge.com/2017/5/17/15645908/google-ai-tensorflowlite-machine-learning-announcement-io-2017`. Accessed June 2018.

[61] He Wang, Souvik Sen, Ahmed Elgohary, Moustafa Farid, Moustafa Youssef, and Romit Roy Choudhury. No need to war-drive. In *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*. ACM Press, 2012. doi: 10.1145/2307636.2307655.

[62] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *Computer Vision – ECCV 2016*, pages 20–36. Springer International Publishing, 2016. doi: 10.1007/978-3-319-46484-8_2.

[63] Lu Xia, Chia-Chih Chen, and J. K. Aggarwal. View invariant human action recognition using histograms of 3d joints. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, jun 2012. doi: 10.1109/cvprw.2012.6239233.

[64] Jaeyoung Yang, Joonwhan Lee, and Joongmin Choi. Activity recognition based on RFID object usage for smart mobile devices. *Journal of Computer Science and Technology*, 26(2): 239–246, mar 2011. doi: 10.1007/s11390-011-9430-9.

[65] Mingyang Zhong, Jiahui Wen, Peizhao Hu, and Jadwiga Indulska. Advancing android activity recognition service with markov smoother. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*, pages 38–43. IEEE, 2015.

[66] Chun Zhu and Weihua Sheng. Motion- and location-based online human daily activity recognition. *Pervasive and Mobile Computing*, 7(2):256–269, apr 2011. doi: 10.1016/j.pmcj.2010.11.004.

[67] Wentao Zhu, Cuiling Lan, Junliang Xing, Wenjun Zeng, Yanghao Li, Li Shen, Xiaohui Xie, et al. Co-occurrence feature learning for skeleton based action recognition using regularized deep lstm networks. In *AAAI*, volume 2, page 8, 2016.

[68] Maryam Ziaeefard and Robert Bergevin. Semantic human activity recognition: A literature review. *Pattern Recognition*, 48(8):2329–2345, aug 2015. doi: 10.1016/j.patcog.2015.03.006.

VITA

NAME	Marco Mele
EDUCATION	
	Master of Science in Computer Science, University of Illinois at Chicago, July 2018, USA
	Master's Degree in Data Science for Computer Engineering, September 2018, Polytechnic University of Turin, Italy
	Bachelor's Degree in Computer Engineering, October 2016, Polytechnic University of Turin, Italy
	Technical High School Diploma in Information and Communication Technologies, June 2013, "G.B. Pentasuglia" Technical High School, Matera, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2015 - IELTS Examination level B2 (6.5/9)
	AY 2017/18 One Year of study abroad in Chicago, Illinois
	AYs 2014/17 Lessons and exams attended in English
SCHOLARSHIPS	
Fall 2017	Italian scholarship for TOP-UIC students from the Polytechnic University of Turin, Italy
Spring 2017	Teacher's Assistantship (TA) position for the Undergraduate course of Computer Architecture at the Polytechnic University of Turin, Italy
Fall 2016	Part–time position as Technical Assistant at the LABInf, Laboratory for Advanced Computer Science, Department of Control and Computer Engineering, Polytechnic University of Turin, Italy
Spring 2016	Teacher's Assistantship (TA) position for the Undergraduate course of Computer Architecture at the Polytechnic University of Turin, Italy

VITA (continued)

TECHNICAL SKILLS

Operating Systems	Linux and UNIX–like, with basics in system administration, networking, and proficiency in Bash–like scripting, acquired in years of Unix user experience, work and OS academic courses
Programming	7–year school experience in C programming, academic level knowledge of Python, Java8, Android SKD, x86 Assembly, awk/sed, MySQL, POSIX
Data Analysis	Python tools (Pandas, Numpy and SciKitLearn, Keras), Twnsorflow, RapidMiner, Hadoop MapReduce, Apache Spark and Storm

PROJECTS

2016–2017	**Data Spaces, Human Resource Analytics:** data mining, supervised and unsupervised learning techniques on a dataset of employment data. The work involved th euse of RapidMiner, providing statistical learning and data mining tools.
	Distributed Programming: laboratory practice in socket programming for client–server interaction and a project of web programming with DBMS integration.
	Big Data: laboratory practice with the **BigData@PoliTO** academic laboratory with Hadoop HDFS and MapReduce, Apache Spark and Spark Streaming for Big Data processing techniques on various data sources.
2017–2018	**Data Science for Networks:** "Detecting Shifts in Propensity Score Stratification when using Relational Classifiers for Network Data": research work in wich we address the problem of reviewing the *Stratified Propensity Score Analysis* used for evaluating the *Average Treatment Effect* in treatment effect observation when we deal with relational data. We proposed a new approach to Propensity Score estimation that makes use of a Relational Classifier and then compare the resulting stratification of the population with one learned by a common classifier, as many times the observed covariates may not be enough to explain the stratification in propensity of the population. The work involved researching, usage of the Twitter APIs, and programming in Python and Spark. Most part of the work and the related paper is publicly available on GitHub.