Title
A Framework for Pervasive Context Awareness

Author
Shen, Chenguang

Publication Date
2016

Peer reviewed|Thesis/dissertation

University of California
Los Angeles

A Framework for Pervasive Context Awareness

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Chenguang Shen

2016

Abstract of the Dissertation

A Framework for Pervasive Context Awareness

by

Chenguang Shen

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Mani B. Srivastava, Chair

The proliferation of pervasive computing devices with unprecedented sensing, communication, and computation capabilities has enabled continuous and ubiquitous monitoring of users and their surrounding environments. While smartphones have evolved from only communication devices into powerful personal computing platforms, connected devices such as smartwatches, cameras, motions sensors, thermostats, and energy meters, collectively dubbed as the *Internet of Things*, are also rapidly permeating our living spaces. The continuous stream of richly annotated, real-time data made available by tapping into the spectrum of sensors available on these devices has led to the emergence of a sprawling ecosystem of context-aware apps. These apps use sensors such as GPS, microphone, accelerometer, and gyroscope to make diverse inferences about user activities and contexts.

Typical context-aware apps employ a suite of machine learning algorithms to extract semantically meaningful inferences from sensor data. However, today's connected devices and mobile operating systems are not designed to support sensing and inference workloads. Compared with the rich network stack on pervasive devices, where one can leverage protocols and abstractions in different layers, context inference apps are composed in a *monolithic* fashion. That is, developers have to implement all data collection and inference logic using the raw data. As a result, the development and execution of context inference apps today are accompanied by myriad difficulties. First, the limited sensor coverage from running inferences on a single device and the need of hand-picking high-level features have made it challenging

to achieve high inference accuracy in building the classification model of apps. Second, the continuous execution of complex algorithms on the main app processor of smartphones and the use of energy-hungry sensors have resulted in high energy consumption of inference apps, especially when they are executed on battery-powered pervasive devices. Finally, the monolithic development practice limits the adoption of inference apps onto heterogeneous devices, and requires enormous amount of efforts from app developers in composing the apps.

In this dissertation we make three research contributions towards building a framework for pervasive context awareness:

- We first showcase that context-aware inferences can benefit from heterogeneous connected devices such as smartwatches. We develop an example app to autonomously infer workout exercises of users from only sensors on a commercial smartwatch, achieving 90% classification accuracy for both cardio and weightlifting exercises while extending the watch battery life by up to 19 hours compared with prior approaches.

- Having identified the problems from our example app, we then perform three optimizations of context inference apps. We have achieved (1) comparable inference accuracy as traditional models and acceptable latency using deep learning without hand-picking features; (2) up to 30× speed-up of deep learning tasks using mobile GPUs and up to 60% energy saving of off-loading inference tasks from the CPU to the DSP; and (3) up to 37% accuracy improvement and up to 67% less energy consumption for context-aware apps from watch-phone coordinations.

- Finally, we close the loop by proposing the design and implementation of a programming framework for context inference apps, with a set of programming abstractions and an associated runtime. The framework helps reduce development tasks by up to 4.5× and source lines of code by up to 12×. It also tackles runtime challenges and achieves 3× better inference accuracy by handling environmental dynamics.

With this work we have also created an open-source toolkit for developing and executing context inference apps using heterogeneous pervasive devices.

The dissertation of Chenguang Shen is approved.

Tyson Condie

William J. Kaiser

Jens Palsberg

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2016

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor Mani Srivastava for all the guidance and support throughout my graduate study. His insightful ideas, sharp understanding of technologies, and great attention to details are all extremely precious and valuable in shaping my research work and my future career. To me, Mani is not only a teacher and an advisor, but also a true role model.

I would like to thank the members of my thesis committee, Tyson Condie, William Kaiser, and Jens Palsberg for the time and effort in helping me improve my research work and this dissertation. I would also like to thank all my colleagues and friends in the UCLA Networked and Embedded Systems Lab, especially Bo-Jhang Ho, Paul Martin, Lucas Wanner, Salma Elmalaki, Yasser Shoukry, Supriyo Chakraborty, Moustafa Alzantot, Kasturi Raghavan, Haksoo Choi, Henry Herman, Jonathan Friedman, Zainul Charbiwala, and Chenni Qian. This work is not possible without the help and feedback from each of you.

Next, I would like to thank my mentors and collaborators Aman Kansal, Amar Phanishayee, Jie Liu, Michel Goraczko, Ratul Mahajan, and all members of the Sensing and Energy Research Group at Microsoft Research. My two internships there offered me a great understanding of industry research and gave me opportunities to design and implement real-world systems.

Finally, I would like to thank my parents for everything they have provided for me. They are the real MVP.

This material is based upon work supported in part by the NSF under awards #0905580, #1029030 and #1213140, and by the NIH under award #U54EB020404. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF and the NIH.

Vita

2007	First Prize in National Olympiad in Informatics Provincial (NOIP), a national high-school programming contest of China.
2009-2012	Undergraduate Research Assistant, Software School, Fudan University, Shanghai, China
2011-2012	Undergraduate Teaching Assistant, Software School, Fudan University
2011	Research Intern, UCLA-CSST program, UCLA, Los Angeles, CA, USA
2012	B.Eng. in Software Engineering, Fudan University
2012-2016	Graduate Student Researcher, Electrical Engineering Department, UCLA
2014	Teaching Assistant, Computer Science Department, UCLA
2014	Research Intern, Microsoft Research, Redmond, WA, USA
2014	M.S. in Computer Science, UCLA
2015	Research Intern, Microsoft Research
2016	Software Engineer Intern, Facebook, Menlo Park, CA, USA

Publications

- Exploring Hardware Heterogeneity to Improve Pervasive Context Inferences. Chenguang Shen, Mani Srivastava. IEEE Computers (under review), 2016.

- MiLift: Efficient Smartwatch-based Workout Tracking Using Automatic Segmentation. Chenguang Shen, Bo-Jhang Ho, Mani Srivastava. IEEE Transactions on Mobile Computing (under review), 2016.

- Beam: Ending Monolithic Applications for Connected Devices. Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, Ratul Mahajan. ATC 2016, USENIX Annual Technical Conference.

- A Case for Ending Monolithic Apps for Connected Devices. Rayman Preet Singh, Chenguang Shen, Amar Phanishayee, Aman Kansal, Ratul Mahajan. HotOS 2015, 15th USENIX Workshop on Hot Topics in Operating Systems.

- It's Time to End Monolithic Apps for Connected Devices. Rayman Preet Singh, Chenguang Shen, Amar Phanishayee, Aman Kansal, Ratul Mahajan. USENIX ;login:, October 2015, Vol. 40, No. 5.

- Towards a Rich Sensing Stack for IoT Devices. Chenguang Shen, Haksoo Choi, Supriyo Chakraborty, Mani Srivastava. ICCAD 2014, IEEE/ACM International Conference on Computer-Aided Design.

- ipShield: A Framework For Enforcing Context-Aware Privacy. Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, Mani Srivastava. NSDI 2014, USENIX Symposium on Networked Systems Design and Implementation.

- Exploiting Processor Heterogeneity for Energy Efficient Context Inference on Mobile Phones. Chenguang Shen, Supriyo Chakraborty, Kasturi Rangan Raghavan, Haksoo Choi, Mani Srivastava. HotPower 2013, ACM Workshop on Power-Aware Computing and Systems.

- MiDebug: Microcontroller Integrated Development and Debugging Environment. Chenguang Shen, Henry Herman, Zainul Charbiwala, Mani Srivastava. IPSN 2012, ACM/IEEE International Conference on Information Processing in Sensor Networks.

CHAPTER 1

Introduction

The proliferation of pervasive computing devices with unprecedented sensing, communication, and computation capabilities has enabled continuous and ubiquitous monitoring of users and their surrounding environments. While smartphones have evolved from only communication devices into powerful personal computing platforms, connected devices such as smartwatches, cameras, motions sensors, thermostats, and energy meters, collectively dubbed as the *Internet of Things*, are also rapidly permeating our living spaces. The continuous stream of richly annotated, real-time data made available by tapping into the spectrum of sensors available on these devices has led to the emergence of a sprawling ecosystem of context-aware apps. These apps use sensors such as GPS, microphone, accelerometer, and gyroscope to make diverse inferences about user activities and contexts, including transportation mode [RMB10, RSD10], location [KKE10], workout exercise [CCC07], conversation episode [RAP11], mood and stress [ESK11, LLL13], physiological state [NDA13b], and nutrition intake [DYN10].

Context-aware apps employ a suite of machine learning algorithms to extract semantically meaningful inferences from sensor data. Underlying the various types of context inferences, we abstract out a canonical inference pipeline, shown in Figure 1.1. Apps today typically subscribe to raw sensor data streams, where hardware fabric, device drivers, and system services work in concert to pass the sensor samples to apps. There are mainly two phases in the lifetime of an inference app:

- **Learning**: The app first collects data and ground truth labels to perform the model training. It extracts features to reduce the dimensionality of sensor data. The features, together with ground truth labels collected from the user, are used to learn a model

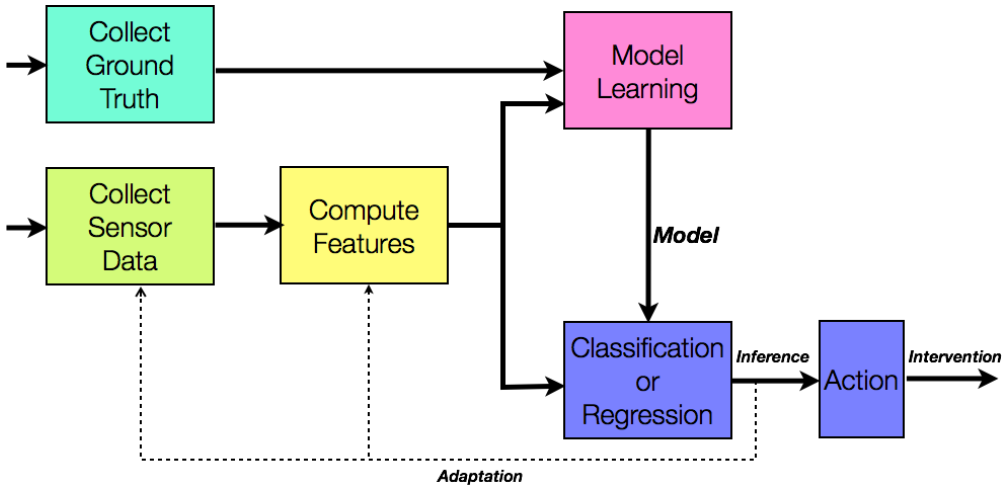


Figure 1.1: A canonical pipeline for context inferences.

for this pipeline. This learning phase is typically done in the cloud because of the complexity of learning algorithms, and the learned model is shipped to the mobile device for classification execution. In reality, many context-aware inference apps are shipped with a model previously learned, so the app is ready for classification after installation.

- **Classification**: Also called **Prediction**. The app collects new data and, with the model, it performs classification over the stream of extracted features to infer context labels (i.e. which class the current context falls under). The context label is later used as the evidence for other decisions or actions. The inference pipeline can also be updated adaptively at runtime.

However, today's connected devices and mobile operating systems are not designed to support sensing and inference workloads. Compared with the rich network stack on pervasive devices, where one can leverage protocols and abstractions in different layers, sensing and inference tasks are treated as second-class citizens on these devices. Apps are composed in a *monolithic* fashion, that is, developers have to implement all data collection and inference logic using the raw data. As a result, development and execution of context inference apps today are accompanied by myriad difficulties, summarized as follows:

Limited sensor coverage and inference accuracy: For inference executions, the first

challenge is achieving high inference accuracy. Due to the monolithic development practice, most context-aware inference apps run on a single device, e.g. a smartphone, and therefore require a user to carry the phone for accurate sensor readings. Whenever the user leaves the phone elsewhere, e.g. on a desk, to focus on other tasks or to charge the phone, the sensors cannot capture any meaningful data. Taking the example of an activity tracking app, the user would still perform various kinds of activities with the phone away from him/her, but the inference app on the phone would not be able to report correct activity labels, and would normally classify he/she as not moving. The above situation will result in limited sensor coverage in terms of the app's ability to capture human movement changes and therefore poor inference accuracy.

Moreover, some user activities such as workout exercises typically involve movements of different body segments such as hands, arms, waists, and legs, and they cannot be accurately monitored by a single smartphone. The inability of context-aware apps to leverage sensing devices other than smartphones has greatly limited their sensing and inference coverages. Recent work has shown that wearable devices, e.g. smartwatches, can be used to assist smartphones for context inferences such as activity recognition [VSM15, SBS15] but did not quantify the benefit of doing so.

The inference accuracy of apps is also limited by other factors. For example, traditional machine learning models rely on the successful choice of features, or the so-called *feature engineering* process, to yield satisfying accuracy.

High energy consumption: The limited battery capacity of pervasive connected devices calls for a detailed study and optimization of energy consumption of context-aware apps. The algorithms used in these apps, including feature computations and classifications, are computationally intensive and power hungry. Additionally, they often need to run in the background continuously to monitor user contexts and behaviors for just-in-time feedback, notifications, and interventions. Currently most always-on context-aware apps run on the main app processor of smartphones representing a significant portion of the phone's overall workload and energy consumption. This situation is usually exacerbated by the use of power-

hungry sensors like GPS and cellular radios. For example, Ju et al. [JLY12] reported that apps consume 4% - 22% of CPU cycles for inference executions, and Kansal et al. [KSB13] observed a reduction of phone standby time from 430 to 12 hours when an app is continuously using the Android proximity API. The high power consumption of context-aware apps and the reduction in battery life as a result will only get worse as apps with even more sophisticated context inference capabilities emerge.

Monolithic development paradigm: Although pervasive sensing devices can in theory enable a wide variety of inferences, developing context-aware apps in practice is arduous because the tight coupling of apps to specific hardware and the lack of inference abstractions require each app to implement the sensing and inference logic on specific hardware devices. This monolithic approach is problematic for both app developers and end users. App developers must employ expertise in data science and machine learning in order to create effective feature computation and classification algorithms. They have to write device-specific logic to collect data, implement various feature computation blocks, train a machine learning model, and wire all feature blocks and classifications together to complete an inference pipeline. This complicates the development process and hinders broad distribution of their apps.

Moreover, because apps are tightly coupled with hardware, it remains challenging to leverage heterogeneous devices for context inferences. In the monolithic approach, an app developed for a particular device cannot leverage other heterogeneous devices. This includes selecting appropriate devices from the current deployment, handling dynamics such as user mobility and intermittent network connections, and partitioning the computation across different devices.

For end users, each sensing device they install is limited to a small set of inference apps even though the hardware capabilities may be useful for a broader set of apps. Existing solutions from both mobile operating systems [gooa, appb] and the mobile sensing community [CLL11, KSB13] either focus on smartphones only or lack flexibility to configure inference pipelines.

While recent work has attempted to tackle these challenges individually, each attempt

is isolated and has limitations, as discussed in Section 1.2. This dissertation first creates an example app to showcase the benefit of running context inferences on a non-smartphone device, then describes a set of optimizations on the inference accuracy and energy efficiency of inferences, and finally proposes the design and implementation of a programming framework for developing context inference apps on pervasive connected devices.

1.1 Contribution

In this dissertation we make three key research contributions, summarized as follows.

1.1.1 Motivating Example: Leveraging Smartwatches for Context Inferences

In Chapter 2, we first showcase that context-aware inferences are not only limited to smartphones apps, but can also benefit from other sensing devices such as smartwatches. We propose the design and implementation of *MiLift*, a workout tracking system on commercial off-the-shelf (COTS) smartwatches. MiLift leverages a new generation of Android smartwatches such as the Moto 360 [mota], which benefit from powerful hardware resources, Bluetooth Low Energy radios, and a rich set of sensors. As users normally wear smartwatches for longer periods of time than carrying smartphones, watches can extend sensor coverage of workout tracking and therefore improve its inference accuracy. MiLift applies *automatic segmentation* to eliminate the burden on users. Using a single smartwatch, MiLift can accurately and efficiently track both cardio and weightlifting exercises without requiring inputs from users. Additionally, MiLift applies optimization techniques such as context-aware duty-cycling and lightweight repetition detection to continuously inferring contexts on smartwatches. MiLift is not only a proof-of-concept example of leveraging the benefits of wearable devices, but also motivates our following work on accuracy and energy optimizations, as well as the multi-device inference framework.

1.1.2 Accuracy and Energy Optimization of Context Inferences

To address the challenges associated with context inference apps today, in Chapter 3 we describe a set of efforts to maximize the inference accuracy and to minimize the energy consumption of apps. We first apply deep learning, e.g., Recurrent Neural Networks (RNN), for training the classification models from mobile sensor data. We have shown that RNN can achieve comparable inference accuracy as traditional models while eliminating the need for feature engineering in development and feature calculation at runtime. Second, we explore the off-loading of computation from main app processors (CPU) to co-processors available in mobile System-on-Chip (SoC) architectures, such as Graphic Processing Units (GPU) and Digital Signal Processors (DSP). The off-loading helps context inferences achieve better performance and energy efficiency. Last, we propose the use of both smartwatches and smartphones to improve always-on context inferences. The coordination can help increase the sensor coverage and inference accuracy by alternating the inference execution across devices at any given time. We also discuss two energy optimization techniques enabled by the coordination, including energy-efficient inference partitioning and eliminating energy-hungry sensors. Observations from these optimizations can be adopted by developers today to improve the context inference executions in their apps.

1.1.3 Closing the Loop: A Programming Framework for Context Inferences

Finally, we tackle the monolithic development paradigm used by today's context-aware app developers. We posit that inference logic for context awareness, traditionally left up to apps, ought to be abstracted out as a system service, thus decoupling "what is sensed and inferred" from "how it is sensed and inferred". Such decoupling enables apps to work in heterogeneous environments with different sensing devices while at the same time benefiting from shared and well trained inferences. In Chapter 4, we propose *Beam*, a programming framework and associated runtime which provides apps with inference-based programming abstractions. Beam introduces the key abstraction of an *inference graph* to not only decouple applications from the mechanics of sensing and drawing inferences, but also directly

Category	C1	C2	C3	C4
Mobile workout tracking apps	●			○
Mobile health frameworks	●	○		●
Weightlifting tracking systems		○	●	
Emerging apps (VimoFit [vim])	●		●	
MiLift (this work)	●	●	●	●

Table 1.1: Summary of prior workout tracking approaches and whether they meet our design challenges, C1: single-device sensing; C2: automatic segmentation; C3: weightlifting exercise tracking; and C4: efficient resource usage. ○ denotes partial fulfillment.

aid in addressing three important challenges: (1) device selection in heterogeneous environments, (2) efficient resource usage, and (3) handling device disconnections. Context-aware apps simply specify their inference requirements, while the Beam runtime bears the onus of identifying the required sensors in the given deployment and constructing an appropriate inference graph.

Based on the above findings on composing and running context inferences across multiple devices, we create and release a suite of open-source toolkit apps to assist developers in data collection, inference composition, and inference execution on Android. The design and implementation of the toolkit are described in Chapter 5.

1.2 Related Work

Apps on smartphones explore a variety of human contexts including transportation modality [RMB10] [HNT], social interactions [XLL13] [NDA13a], and physical and mental healthiness [RMM10] [LFR12] [LLL13] [HXZ13]. Recently the emergence of the Apple Watch [appa], Android smartwatches (e.g., the Moto 360 [mota]), and the Microsoft Band [msb] also prompts the development of context inferences on smartwatches and wearable devices [MPA14] [SBS15] [NGG15] [LGM15].

We group prior work in to the following categories and highlight our contributions.

1.2.1 Workout Tracking

Existing solutions of workout tracking and management using mobile devices can be summarized into the following categories (shown in Table 1.1):

Mobile workout tracking apps including RunKeeper [run], Strava [str], and MapMyRun [mapb] focus on cardio exercise tracking and management using a single smartphone. Due to the limited sensor coverage when using phones, these applications only support specific types of cardio exercises and require users to manually start and stop workout sessions.

Mobile health frameworks such as Google Fit [gooc], Apple HealthKit [hea], and Microsoft Health [msh] provide APIs for both app developers and data scientists. Each framework also provides an app for users to manage workout tracking. Typically built-in as part of the mobile operating systems, the resource usage of these frameworks are well optimized. However, they mostly focus on cardio exercises and do not support tracking of weightlifting exercises. The front-end apps require manual selection of workout types as well.

Weightlifting tracking systems: The weightlifting classification system in MiLift is motivated by several prior work in this space. Chang et al. [CCC07] performs free weight exercise tracking using two wearable accelerometers, one in a user's glove and one in a waist pocket. Their system can automatically classify types of free weight exercises and count reps. MyoVibe [MLN15] and Burnout [MLN16] embeds wearable sensors in fitness clothing and leverages muscle vibrations to identify muscle activations and to estimate fatigues during exercises. RecoFit [MSG14] provides a model to segment exercises from non-exercise activities and count weightlifting exercises, but does not study the feasibility of running continuous tracking on user devices. NuActiv [CSG13] uses a smartwatch and a smartphone to track exercises and everyday activities by decomposing activities into semantic attributes. It applies zero-shot attribute-based learning for recognizing newly unseen types of exercises. Pernek et al. [PHK13] achieves repetition detection of weightlifting exercises using Dynamic Time Wrapping. FEMO [DSY15] tracks repeating patterns of free weight exercises by instrumenting dumbbells with RFID sensors and measuring frequency shifts caused by Doppler Effect. myHealthAssistant [SBV11] employs multiple customized sensors on a user's body and uses

a smartphone as a hub to track weightlifting exercises. Mortazavi et al. [MPA14] can count reps of weightlifting exercises but call for manual type selection.

Emerging apps and web services including the VimoFit app [vim], the Atlas wristband [atl], and the Microsoft Band API [ban] aim at autonomous workout tracking. Although some of them support rep counting for guided workouts, they still require certain manual inputs from users. Section 2.5.3 shows the high energy consumption of VimoFit and compares it with MiLift. Other services such as JEFIT [jef], WorkoutLabs [wor], and Gymwolf [gym] provide workout management and guidance from self-report workout data.

Different from prior work, MiLift is the first end-to-end system on commercial smartwatches that uses automatic segmentation to track both cardio and weightlifting exercises without requiring users to start/stop tracking and/or select workout types. MiLift also tackles challenges such as single device sensing and efficient resource usage described in Section 2.2.

1.2.2 Deep Learning on Mobile and Wearable Sensor Data

There have been several attempts to perform deep learning on mobile sensor data. DeepSpying [BR15] leverages RNN for keystroke detection from smartwatch accelerometer data, but achieves very limited overall detection accuracy (around 70%). Hammerla et al. [HHP16] uses RNN for activity recognition on a datset with wearable sensor data, but provides no implementation or profiling on mobile devices. Ross et al. [RRP14] benchmarks OpenCL back-end functions on a mobile GPU, but focuses on low-level matrix operations instead of neural network operations. Tschopp et al. [Tsc15] profiles the performance of running Convolution Neural Networks on CPUs and on GPUs with OpenCL, but does not study the consequences of performing mobile inference workloads on mobile processors. DeepEar [LGQ15] and Bhattacharya et al. [BL16] apply deep learning on mobile and wearable sensor data for inferences and profile the classification on DSPs, but do not consider mobile GPUs which are more suitable for parallel tasks. DeepX [LBG16] and LEO [GLR16] manage to execute deep neural networks on mobile GPUs using CUDA or OpenCL, but

do not consider Recurrent Neural Networks (RNN) to explore the temporal correlation in time-series data. In contrast, our work is the first to provide an end-to-end example of performing context inferences on mobile and wearable sensor data using RNN. We have shown that by leveraging mobile GPUs, RNN can achieve comparable accuracy and latency as the traditional feature calculation plus classification models.

1.2.3 Leveraging Heterogeneous Processors in Mobile SoC

To address the power-hungry nature of context-aware apps on mobile phones, several researchers have recently explored leveraging low-power processors. Priyantha et al. [PLL11] uses an external MSP430 microcontroller for off-loading frequent sampling of sensor data. Ra et al. [RPK12] examines the partitioning of different modules of a sensing app between an app processor and low-power processors to reduce the overall energy consumption. However, the low-power processors used in these experiments are low-end microcontroller class processors optimized for sampling and buffering, and not capable of meaningfully doing more complicated context inferences. Moreover, such low-end processor cores are deeply embedded in the hardware fabric and unlikely to be exposed to app developers. Philosophically, these work target off-loading of simple frequent tasks from the main app processor, whereas our work explores off-loading of computationally intensive tasks.

Among other work in this space is Lin et al. [LWZ12] which compares heterogeneous loosely coupled processor cores, e.g. Cortex-A9 + Cortex M3, with tightly coupled processor cores with identical instruction set architectures but different power-performance operating points, e.g. ARM big.LITTLE [arm]. They implement Distributed Shared Memory (DSM) between the Cortex-A9 and Cortex-M3 [LWL12], which enables efficient data exchange between the two asymmetric processors. Based on this model, they propose K2, a prototype OS distributing both app and OS workloads on the TI OMAP4 SoC [TIa], by reusing most of the Linux 3.4 source code [LWZ14].

Although using mobile GPUs for general-purposed computing tasks is a relatively new field of research, there has been a number of attempts from different vendors and the open

source community. The NVIDIA CUDA [cud] is the most widely used framework for deep learning and has a large developer community. It is supported by most popular deep learning frameworks as the processing back-end, such as Tensorflow [ten], Theano [the], Torch [tor], and Caffe [JSD14]. However, the CUDA driver is not open-source, and therefore it can run on only a few mobile devices with NVIDIA mobile GPUs, such as the NVIDIA Tegra [NVI] tablets and the Google Project Tango [tan] tablets. The open-source solution OpenCL can be used by Torch or Caffe using custom implementations, and supports GPUs from different vendors including the popular Ardeno GPU on Qualcomm Snapdragon SoCs [sna]. But OpenCL as of today only supports low-level functions such as matrix operations instead of neural networks operations. It is also no longer officially supported by Android. Google's Renderscript [ren] is the official solution of GPU computing on Android since the version 4. It can run on most stock Android phones but support only low-level BLAS functions as well. Besides, developers have no control on where the code runs. There are also emerging solutions such as the NVIDIA Vulkan [vul] and Memkite on iOS [mem], but they either lack details now or do not support mobile GPUs.

1.2.4 Context Inferences on Smartwatches

Recent work has started leveraging smartwatches to assist context inferences running on smartphones and other devices. Vigneshwaran et al. [VSM15] uses smartwatches for activity recognition. Mortazavi et al. [MPA14] executes exercise counting on smartwatches. Shoaib et al. [SBS15] makes preliminary investigation into the fusion of data from watch and phone sensors. Liu et al. [LKL15] uses watches to detect unsafe driving behaviors. Recently Huang et al. proposes WearDrive [HBC15] as an energy-efficient storage system for wearables which can be adopted by apps in this work for further optimization of the energy cost of storage and communication. Unlike prior work, this work conducts a systematic study on the accuracy and energy implications of performing context inferences across both the phone and the watch.

1.2.5 Programming Framework for Context Inferences

Beam's inference graph draws inspiration from data-flow graphs used in a wide range of scenarios such as routers [KMC00], operating systems [Rit84, MP96], data-parallel computation frameworks [IBY07, goob], and Internet services [WCB01]. Beam is the first framework that provides inference abstractions to decouple apps, inference algorithms, and devices, using the inference graph for device selection, efficiency, and disconnection tolerance.

Beam is also motivated by the following categories of work:

Device abstraction frameworks: HomeOS [DMA12] and other platforms [hom, rev, VKP06, ABY14] provide homogeneous programming abstractions to communicate with devices. For instance, HomeOS apps can use a generic motion sensor role, regardless of the sensor's vendor and protocol. These approaches only decouple device-specific logic from apps, but are unable to decouple inference algorithms from apps. Moreover, they cannot provide device selection or inference partitioning capabilities.

Cross-device frameworks: Rover [JdT95], an early distributed object programming framework for mobile apps, allows programmers to partition client-server apps; it provides abstractions such as relocatable objects and queued remote procedure calls to ease app development. Sapphire [ZSV14], a more recent framework, requires programmers to specify per-object deployment managers which aid in runtime object placement decisions, while abstracting away complexities of inter-object communication. MagnetOS [LRW05] dynamically partitions a set of communicating Java objects in a sensor network with a focus on energy efficiency. Like these frameworks, channels in Beam abstract away local and remote inter-module communication.

Macro-programming frameworks [BHS07, GGG05, LAH06] provide abstractions to allow apps to dynamically compose dataflows [MWM06, NMW07]. Semantic Streams [WZL06] and Task Cruncher [TKN10] address sharing sensor data and processing across devices. However these approaches focus on data streaming and simple processing methods, e.g., aggregations, rather than generic inferences, and do not target general purpose devices e.g., phones, PCs. In addition, they do not address device selection or inference partitioning at runtime.

Mobile sensing frameworks: Existing work has focused only on apps requiring continuous sensing on a *single* mobile device. Kobe [CLL11], Auditeur [NDA13a], and Senergy [KSB13] propose libraries of inference algorithms to promote code re-use and explore single device energy-latency-accuracy trade-offs. Other work [LHL14, JLY12, KLJ08, KSB13] has focused on improving resource utilization by sharing sensing and processing across multiple apps on a mobile device. None of these approaches address problems such as modular inference composition, device selection with user mobility, inference partitioning across multiple devices, or handling disconnections.

Beam fundamentally differs from the above prior work by using the inference graph to decouple apps from sensing and inferences, aid in device selection to operate in heterogeneous environments, and support global resource optimizations.

1.3 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 discusses the design and implementation of MiLift, which automatically segments exercises from non-workout activities so that users do not need to manually start/stop tracking or select exercise types. MiLift can achieve above 90% precision and recall for tracking both cardio workouts and weightlifting exercises. It can also extend the battery life of a Moto 360 watch by up to 8.25× (19.13h) compared with previous approaches.

- Chapter 3 discusses three techniques we have proposed to improve the inference accuracy and energy efficiency of context-aware apps, including (1) applying deep learning on mobile sensor data, achieving comparable accuracy and acceptable latency without hand-picking features; (2) exploiting co-processors for context-aware apps, with up to 30× speed-up when running deep learning tasks on mobile GPUs and up to 60% energy saving when off-loading inference tasks to DSPs; and (3) exploring watch-phone collaborations which result in up to 37% accuracy improvement and up to 67% less

energy consumption for context-aware apps.

- Chapter 4 proposes and evaluates Beam, a programming framework and associated runtime for multi-device context inference apps. Using Beam, we develop two representative apps (Quantified Self and IFTTT Rules), where we show up to 4.5× lower number of tasks and 12× lower source line of code in development effort. Moreover, Beam results in up to 3× higher inference accuracy due to its ability to select devices in heterogeneous environments, and Beam's dynamic optimizations match hand-optimized apps for network resource usage.

- Chapter 5 describes the design and implementation of our open-source toolkit for composing and executing context inference apps.

- Chapter 6 presents our concluding remarks.

CHAPTER 2

Motivating Example: Context Inferences on Smartwatches

In this chapter, we use workout tracking as an example app to showcase that context-aware inferences are not only limited to smartphones apps, but can also benefit from other sensing devices such as wearables. This not only a proof-of-concept example of leveraging the benefit of wearable devices, but also motivates our following work on accuracy and energy optimizations, as well as the multi-device inference framework.

2.1 Motivation and Contribution

The emergence of mobile sensing devices such as smartphones and wearables has enabled ubiquitous and continuous context inferences, including various types of health monitoring and workout tracking apps. The unique location of mobile devices on human bodies and their abilities to capture human movements have made them ideal targets to track physical workout exercises of users. With rising obesity and cardiovascular diseases linked to physical inactivity, such workout tracking can provide quantitative data on users' everyday activities and assist both users and physicians in achieving better health care, rehabilitation, and self-motivation [PGR08, CBV15, KMH13]. Compared with self-reporting, the use of mobile devices for workout tracking can provide more accurate summaries for both *cardio* and *weightlifting* exercises and avoid over- or under-estimations [RMB10, GTV14]. Although there are a variety of approaches for mobile workout tracking, they lack the ability to *automatically* segment workout activities and therefore impose substantial burdens on users, such as requiring users to manually start/stop tracking or to select exercise types. Failure to

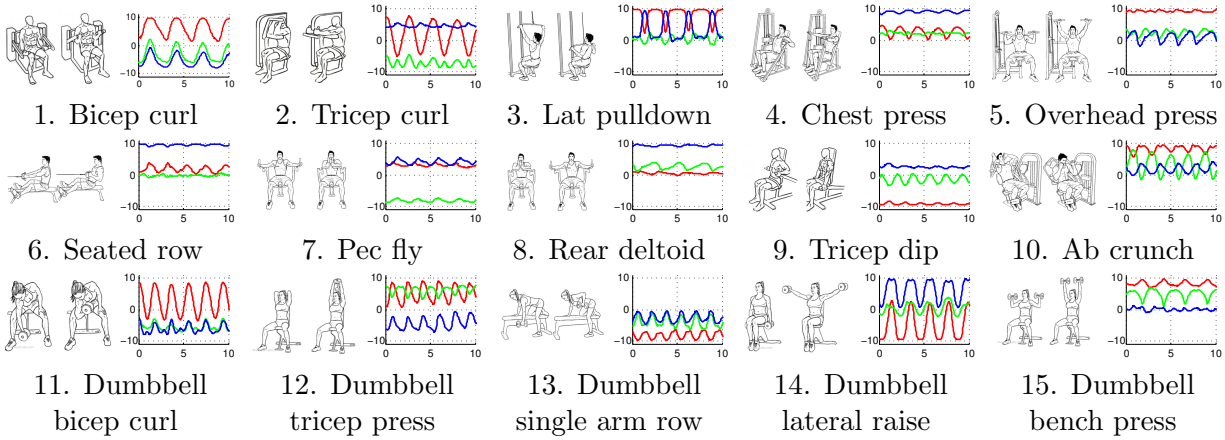


Figure 2.1: Illustration of 15 weightlifting exercises considered in this chapter (image sources [wor, gym]) and repeating patterns in gravity sensor data traces collected from our user study. x-axis shows time in s and y-axis shows 3-axis gravity readings in m/s^2. Type 1-10 are machine exercises, and 11-15 are free weight exercises.

provide timely inputs may lead to inaccurate tracking results and/or excessive energy consumption. These burdens have made prior approaches less attractive compared with simple self-reporting.

For example, smartphone apps such as RunKeeper [run] and Strava [str] can only track specific types of exercises (e.g. running and biking) and require users to manually start and stop tracking. Mobile health monitoring frameworks such as Apple HealthKit [hea] and Google Fit [gooc] leverage both smartphones and wearable devices but require users to specify the type of workouts. While most previous work focused on monitoring cardio workouts, a few considered tracking weightlifting exercises [CCC07, MPA14, DSY15]. However, these approaches introduce extra user burdens such as the use of multiple sensors and energy-hungry algorithms while still requiring certain degrees of user inputs for accurate tracking. Recently there have been attempts from commercial apps to perform automatic exercise segmentation, as seen in the Pocket Track feature of RunKeeper [poc] and the VimoFit app [vim]. Nevertheless, they focus on limited types of exercises or consume excessive battery energy.

In this chapter, we describe the design and implementation of MiLift, a workout tracking system that uses *automatic segmentation* to eliminate the burden on users. MiLift leverages

a new generation of Android smartwatches such as the Moto 360 [mota], which benefit from powerful hardware resources, Bluetooth Low Energy radios, and a rich set of sensors. Using a single smartwatch, MiLift can accurately and efficiently track both cardio and weightlifting exercises without requiring inputs from users. Additionally, MiLift applies optimization techniques such as context-aware duty-cycling and lightweight repetition detection to continuously inferring contexts on smartwatches.

We highlight the research contributions of MiLift as follows:

First, MiLift can automatically segment both cardio workouts and weightlifting exercises from non-workout activities using a two-stage classification model. It is the first system to apply and fully evaluate such an automated algorithm in workout tracking. Unlike previous tracking approaches, users do not need to provide any manual inputs to MiLift, such as selecting types of exercises or starting/stopping exercise sessions. MiLift runs in the background on a smartwatch and also provides a UI to visualize the workout summary of users for management. From our user study, MiLift's automatic segmentation feature is proven valuable to individuals who regularly perform gym exercises.

Second, MiLift can track both weightlifting machines and free weight (dumbbell) exercises, as shown in Figure 2.1. MiLift meets real-world user requirements during weightlifting exercises, including (1) automatically detect the start and stop of weightlifting sessions (sets); (2) count repetitions (reps) of exercises; (3) classify the type of exercises. Our evaluation on a dataset of 2528 sets of weightlifting exercises (24408 reps) collected by 22 users shows that MiLift can achieve above 90% average precision and recall for both weightlifting session detection and exercise type classification. The average error of rep counting in MiLift is 1.12 reps (out of an average of 9.65). Weightlifting detection in MiLift requires no model training and is user-independent.

Finally, to achieve efficient resource usage, MiLift employs two techniques that have not been applied to wearable context inferences by prior work: a context-aware duty-cycle optimization and a lightweight revisit-based weightlifting detection algorithm. Our experiments on a Moto 360 smartwatch indicate that watch battery lives can be extended by up to 8.25×

(19.13h) by running MiLift instead of unoptimized tracking apps. Even with a continuous execution of MiLift, the watch battery can last for more than a day and therefore will not require extra charging by users.

The MiLift app is open-source and available on Github [mil].

2.2 Challenges and Design Choices

We discuss several key challenges towards an autonomous and efficient workout tracking system and highlight the design choices made in MiLift.

2.2.1 C1: Single-device Sensing

Workout tracking apps running on smartphones cannot accurately sense user activities when the phone is placed away from the user. Moreover, workout exercises typically involve movements of different body segments such as hands, arms, waists, and legs and cannot be accurately monitored by a single smartphone. Prior tracking algorithms either placed more than one sensing device (e.g. a phone and a watch) on a user [CCC07, CSG13] or required instrumentation of weightlifting equipment [DSY15]. In contrast, smartwatches are less intrusive since most users wear them for the majority of the day. Watches can sense wrist orientations and partial torso movements whereas smartphones, typically carried in pockets, can only capture body postures. Therefore MiLift uses a single smartwatch to replace smartphones and other sensors previously used for workout tracking.

2.2.2 C2: Automatic Segmentation

Most workout tracking apps require users to manually choose workout types and start/stop the tracking of each session. If a user fails to mark the session end in time or even forgets to do so, the tracking algorithm could overestimate the current session and/or consume excessive energy. To eliminate user burdens, MiLift can detect a user's activity transitions and automatically segment different workout exercises using a two-stage classification model: it

first applies a lightweight classifier on low-power inertial sensor data to determine high-level user activities such as non-workout, walking, running, and weightlifting, and only starts detailed weightlifting analysis upon detection of weightlifting exercises. This multi-stage model is motivated by prior work on hierarchal activity classification [KLL10, ZMN10, XSW11].

2.2.3 C3: Weightlifting Exercise Tracking

Medical researchers have shown that weightlifting exercises (or weight training, strength training) can help improve metabolic function and muscle strength [PFB00, CHS05]. Although *free weight exercises*, such as those using dumbbells and barbells, can sometimes lead to greater muscle activities than *machine-based weightlifting exercises* [MF94], both should be combined to maximize training outcome [CDS05, PFB00]. Most workout tracking apps monitor cardio activities such as walking and running, but few can efficiently track weightlifting exercises, including the following metrics:

- Number of *sets*: each set is a workout session that includes several repetitions of the same weightlifting exercise.
- Number of repetitions (*reps*) in a set: each rep is an instance and the basic unit of a particular weightlifting exercise.
- *Type* of the exercise: for example, dumbbell bicep curl.

MiLift exploits repeating patterns of human arms during weightlifting exercises as demonstrated in Figure 2.1, and performs weightlifting classification include set detection, rep counting, and exercise type classification. MiLift considers 10 types of weightlifting machine exercises (#1-#10) and 5 types of dumbbell-based free weight exercises (#11-#15).

2.2.4 C4: Efficient Resource Usage

The limited battery capacity of mobile devices calls for a detailed study and optimization of energy consumption of workout tracking services. We propose that the battery life of a smartwatch should last for at least 16 hours (a full day except sleeping) even with continuous workout tracking, so that users do not need to charge the watch during the day.

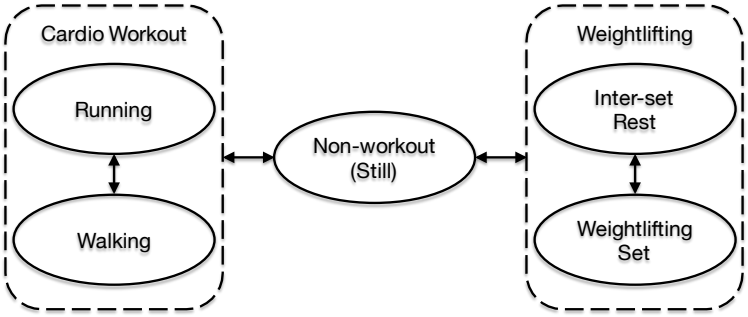


Figure 2.2: State transitions of a user's workout activities.

Previous approaches can rapidly drain out device batteries because of the continuous nature of inference executions and the use of complex algorithms for weightlifting tracking such as Dynamic Time Wrapping (DTW) [PHK13]. MiLift applies two techniques to achieve efficient resource usage on watches: 1) a context-aware duty-cycling optimization, and 2) a lightweight algorithm for weightlifting detection.

2.3 System Architecture

In this section, we propose the system architecture of MiLift, describe the implementation of a two-stage classification model, and discuss optimization techniques.

2.3.1 Overview

MiLift aims to track workout activities of a user using only a smartwatch (C1). Exercises of a user can be categorized into three groups: non-workout (still), cardio workouts such as walking and running, and weightlifting. Figure 2.2 describes state transitions of these exercises. A user can start weightlifting or cardio workouts from the non-workout state. However, we assume that a user cannot perform weightlifting right after a cardio workout or vice versa because there has to be a short transition period to non-workout first, for example, taking a rest or preparing for the next exercise. State transitions also take place within each group: the user can switch between walking and running during a cardio session or take rests between weightlifting sets.

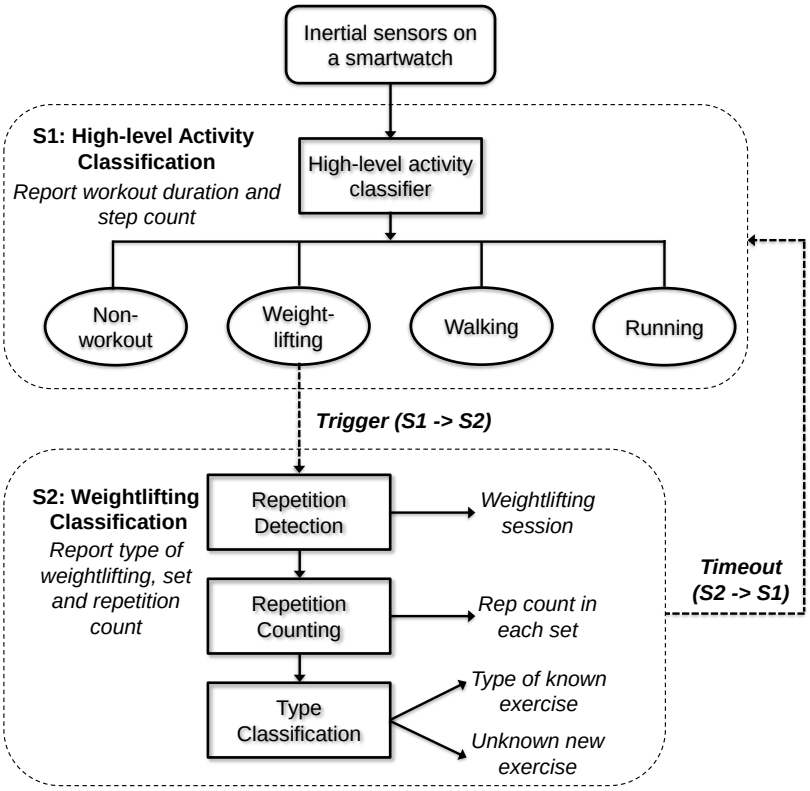


Figure 2.3: MiLift architecture and state transitions.

Motivated by the state transitions of user activities, MiLift uses a two-stage classification model to accurately and efficiently track workout activities, shown in Figure 2.3. The model contains two stages:

S1: High-level activity classification: S1 aims at detecting high-level activity state transitions of a user shown in Figure 2.2 and tracking cardio workouts. It implements a lightweight algorithm to label a data window with activities including non-workout, walking, running, and weightlifting. If walking or running is detected, session duration and step counts are logged. Once weightlifting is detected, MiLift wakes up the weightlifting classification module described below which involves more complicated computations.

S2: Weightlifting classification: This module analyzes inertial data and performs detailed weightlifting classification (C3) including set detection, rep counting, and type classification. We have implemented an autocorrelation-based algorithm and a lightweight revisit-

based algorithm to achieve efficient resource usage (C4).

To perform automatic segmentation on user activities and eliminate the burden on users to manually start/stop tracking of sessions (C2), MiLift transits between S1 and S2 based on current user contexts. The state transition also helps preserve battery energy of smartwatches (C4).

2.3.2 High-level Activity Classification

In MiLift, inertial data samples from smartwatches are first labeled by a high-level activity classifier. Motivated by prior work on mobile sensing [MSS06, RMB10, HNT], the classifier takes a window of 3-axis accelerometer data and generates an activity label such as *non-workout, walking, running*, or *weightlifting*. The classification pipeline includes:

Sampling and preprocessing: MiLift uses a $1s$ classification window on accelerometer data sampled at $50Hz$. The choice of $1s$ window size is also seen in previous activity recognition work [RMB10, HNT] so that the data window contains enough samples for feature functions but also keeps only one type of activity in a single window. Data is buffered for each second and then sent to the next stage for feature extraction.

Feature extraction: This submodule takes an $1s$ window of accelerometer data and reduces its dimension by applying a set of feature functions in both time and frequency domains. Features considered in our system are mean, variance, range, root mean square (RMS), mean absolute deviation (MAD), magnitude, skewness, kurtosis, quartiles, median, and energy coefficients between 1-$5Hz$ from Discrete Fourier Transform (DFT). For each feature, MiLift fuses three accelerometer axes by computing on each axis separately and then taking an average. To improve performance and reduce feature calculation workload, we apply two feature selection algorithms implemented in scikit-learn [PVG11] including the univariate statistical test and the tree-based feature ranking. These two selection algorithms rank features based on their significances and select 7 of them for use in MiLift, including mean, standard deviation, MAD, range, the 1st quartile, the 2nd quartile, and DFT at $5Hz$. The feature vector is then sent to the next stage for classification.

Classification: The classification module takes a feature vector as input and generates an activity label for each window (i.e., every second). We have implemented two categories of classification models. The first approach uses Conditional Random Fields (CRF) to continuously label each data window represented by the feature vector. CRF is commonly used for sequence labeling tasks such as part-of-speech tagging and image segmentation [SM06]. Prior studies also use CRF for tasks on sensor data such as room occupancy inference from motion sensors and human activity recognition from wearable accelerometers [YTS14, LLN11, VVL07, NDH10].

In MiLift, CRF is used to exploit the temporal correlation among workout activities. Each state y_t in CRF corresponds to a ground truth activity label at time t, and each accelerometer feature vector is used as an observation x_t. The joint probability of a state sequence $\mathbf{y}$ and an observation sequence $\mathbf{x}$ is modeled as:

$$p_\lambda(\mathbf{y}|\mathbf{x}) = \frac{1}{Z_\lambda(\mathbf{x})} \cdot \exp(\sum_{j=1}^{n} \sum_{i=1}^{m} \lambda_i \mathbf{f}_i(\mathbf{y}, \mathbf{x}))$$

where n is the total time considered, $\mathbf{f}$ is a set of m feature functions internally used by the CRF (not to be confused with our accelerometer features), λ is a weight vector for all feature functions, and Z serves as a normalization term. Because most CRF implementations only accept nominal (string) observations, they cannot process floating-point accelerometer features. Therefore we use a k-means algorithm to group each feature into a cluster and use the corresponding integer cluster ID as an input observation to CRF. We have considered different numbers of k-mean clusters including 4-20 for the best CRF training performance.

In addition to the continuous CRF, our second approach for high-level activity classification applies an instance classifier to label each $1s$ window and then uses a Hidden Markov Model (HMM) to smooth the label series. We consider three popular instance classifiers including Random Forest (RF), Decision Tree (DT), and Support Vector Machine (SVM). Parameters of all models are tuned in the training stage, such as maximum depths for DT, number of trees and size of feature subsets used for each split for RF, and kernel types for SVM. These models label each individual $1s$ window independently. However, adjacent

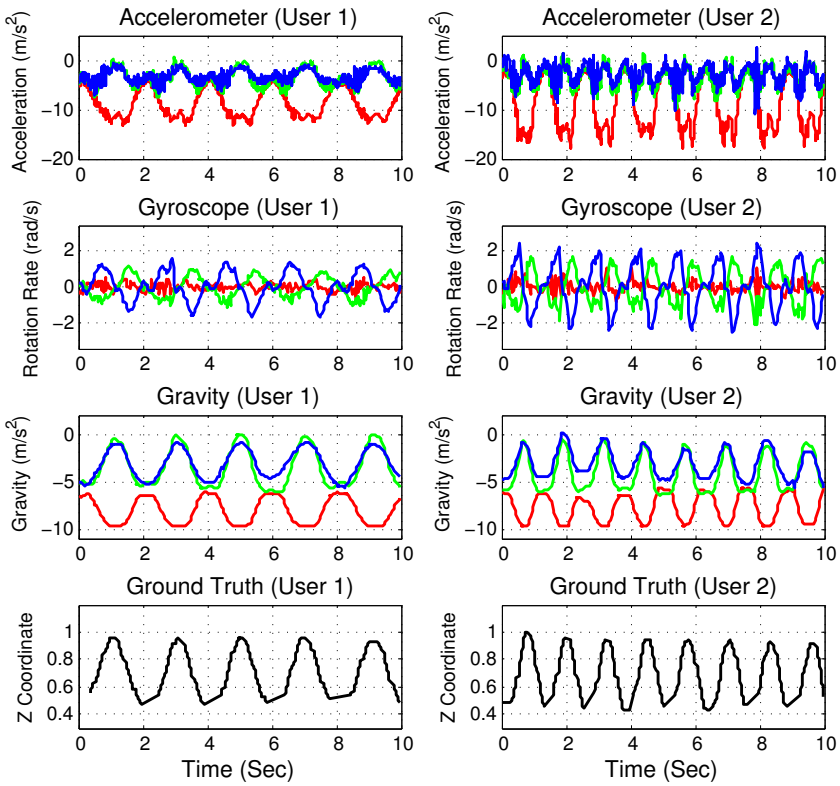


Figure 2.4: Sensor traces comparison of dumbbell single arm row performed by two users, showing that gravity sensor can best demonstrate the repeating patterns.

windows are temporal-correlated as workout activities are continuous and would not transit frequently within a short period. For example, a user may briefly raise his or her arms while sitting in the office but instance models may classify this action as weightlifting. We apply an HMM classifier to smooth output activity labels of instance classifiers and filter out unlikely activity transitions for better accuracy. To generate the HMM model, we use ground truth activity labels as hidden states and output labels from instance classifiers as observations.

2.3.3 Weightlifting Classification

The second state in the two-stage classification model of MiLift is a weightlifting classification module. It is waken up by the high-level activity classifier when users are performing weightlifting exercises. This module achieves three tasks: (1) detecting weightlifting sessions and label boundaries of *sets*, (2) counting number of *reps* (repetitions) in each set, and (3) classifying type of current weightlifting exercise. The weightlifting classification module

must conserve battery energy of smartwatches (C4).

2.3.3.1 Sensor Choice

There are several available inertial sensors on a smartwatch that can capture human motions, including but not limit to accelerometers and gyroscopes. A gravity sensor is a low-power software sensor fusing both accelerometer and gyroscope [andb]. Gravity sensor data can reflect wrist orientations of users and can be a good indicator of repeating weightlifting exercises with wrist movements. Using the dumbbell single arm row exercise as an example, Figure 2.4 demonstrates the ability of different sensors to capture weightlifting exercises. Sensor data traces collected from two users each performing one set of exercises are compared with ground truth motion traces captured by an OptiTrack Prime 13 tracking system [optb]. For both users, the gravity data is less noisy than the accelerometer and gyroscope data and can better highlight the repeating pattern in signal traces generated by weightlifting exercises. Moreover, amplitudes of the gravity data across two users are more consistent compared with the other two sensors, indicating that gravity sensor can better identify types of weightlifting exercises across different users. Therefore MiLift uses the Android gravity sensor for weightlifting detection. However, we acknowledge that single-point sensing has limited coverage on the human body and discuss this issue in Section 2.6.

Figure 2.5 (a) demonstrates a trace of gravity sensor data during a user's weightlifting exercise session (bicep curl machine). This session can be clearly identified by repeating patterns in the gravity sensor trace. Moreover, since the corresponding gravity data are cyclical, each rep in this session can be identified and the number of reps can be counted. In contrast, we see arbitrary patterns recorded during non-workout periods shown in Figure 2.5 (b). Therefore, MiLift quantifies repeating patterns in gravity data to detect weightlifting sessions and count reps, using two approaches including an autocorrelation-based algorithm and a revisit-based algorithm. MiLift then classifies the type of the detected weightlifting activity.

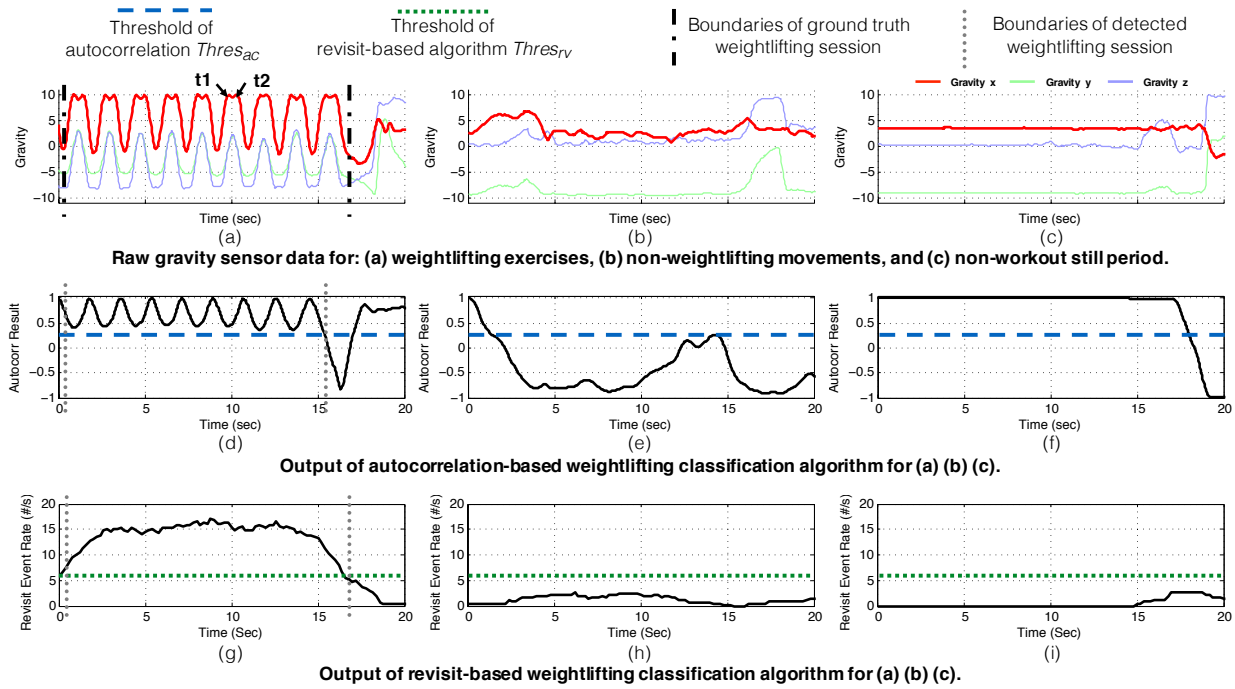


Figure 2.5: Results of applying autocorrelation-based algorithm and revisit-based algorithm on gravity sensor data for three cases: weightlifting (bicep curl), non-weightlifting movements, and non-workout still period.

2.3.3.2 Autocorrelation-based Weightlifting Detection

Weightlifting session detection: Autocorrelation is a technique for examining the periodicity of a signal series. It calculates correlation between a sample window at time t and another same-size window with an offset in time known as the *lag* ℓ. The output of an autocorrelation series can be represented as a function of ℓ:

$$\mathbf{R}_\ell = \frac{S_t^{\mathbf{T}} S_{t-\ell}}{|S_t| \cdot |S_{t-\ell}|}$$

where S_t is a $w-$element vector representing a signal window with size w starting from time t. If a sample window and another window with lag ℓ have similar signal patterns, the autocorrelation value $\mathbf{R}_\ell$ will be close to 1 indicating significant similarity between these two windows. We apply a threshold value $Thres_{ac}$ on $\mathbf{R}_\ell$ to identify weightlifting sessions. In order to capture periodicity of weightlifting exercises, we choose a small window size but longer than the typical duration of a rep ($6s$ from our analysis) to cover all possible rep

lengths.

For weightlifting session detection, we apply autocorrelation using $1s$ sliding windows on gravity data to capture all possible sessions. Figure 2.5 (d)(e)(f) plot autocorrelation results when applied on three different cases of sensor readings shown in Figure 2.5 (a)(b)(c). The figures also show the threshold $Thres_{ac}$ used for repetition detection as well as both ground truth and detected weightlifting sessions. When autocorrelation is performed on a weightlifting session, the output values are close to 1 and demonstrate spike patterns (shown in (d)). A peak in autocorrelation results is aligned with a valley in raw data because this is the beginning of a new rep and autocorrelation discovers the maximum similarity between current window and the previous one. MiLift applies $Thres_{ac}$ on peaks from autocorrelation results to identify a weightlifting session. In contrast, when the input signal trace indicates no repeating wrist motion, for example when a user is adjusting the weights, the autocorrelation output values show no peaks and are relatively low (shown in (e)). However, one exception exists that can lead to constant high autocorrelation values. When a user's wrist is stationary during breaks and gravity readings are relatively constant, the output values can remain close to 1 (shown in (f)). Our algorithm filters out such cases by removing sessions with consistently high autocorrelation values but no spike patterns.

Another issue in this algorithm is which axis should we consider for autocorrelation. From the gravity signals shown in Figure 2.1, there is no universally dominant axis with the most obvious peak-valley patterns. In MiLift, we explored three techniques to select the best gravity axis: (1) summing up absolute values of each axis; (2) taking absolute values of sums of each axis; (3) performing autocorrelation on each axis separately and choosing the best result. Our experiment indicates the last strategy is the most robust against false positives and demonstrates the best accuracy among the three.

Rep counting: After weightlifting sessions are detected and labeled, the rep counting part is achieved by simply performing a naive peak detection on autocorrelation results. Because non-repeating signals are already filtered out, the number of reps can be derived by simply counting peaks.

2.3.3.3 Revisit-based Weightlifting Detection

Although the autocorrelation-based algorithm can effectively detect repetitions in time-series data, it can incur $O(wL)$ overhead upon arrival of each new gravity sample where L is the maximum lag of interest and w is the window size, leading to heavy computations (challenge C4). We propose a lightweight algorithm which quantifies revisit events in gravity sensor data to detect weightlifting sessions and count reps with less computation and better efficiency.

Weightlifting session detection: Figure 2.5 (a) shows that gravity signal demonstrates repeating patterns during weightlifting sessions. This implies that gravity sensor readings (i.e., $(grav_x, grav_y, grav_z)$) with similar values can be found within a short time frame, typically not longer than the period of a rep. For example, the gravity reading marked as $t1$ has a similar value to sample $t2$ about one second later. A necessary condition of a weightlifting session is that repeating patterns of sensor readings are seen within a short time window whose length is similar to the typical length of one rep ($6s$ from our analysis), indicating that the majority of gravity samples in this window have a similar sample within the same window. However, the strategy of comparing sensor data windows whenever a new sample arrives can cause high overhead, as seen in autocorrelation. Instead, we present a heuristic algorithm to discover repeating patterns in the signal stream, called the *revisit-based approach*:

First, each sample of sensor readings ($grav_x$, $grav_y$, $grav_z$) is discretized at an interval of I giving it a discretized value vector D where:

$$D = \left(\left\lfloor \frac{grav_x}{I} \right\rfloor \cdot I, \left\lfloor \frac{grav_y}{I} \right\rfloor \cdot I, \left\lfloor \frac{grav_z}{I} \right\rfloor \cdot I \right)$$

Therefore two similar samples will share the same D value. Our experiments indicate the best choice of I is 0.6.

Second, we maintain a hash table $\mathcal{H}$ to store incoming samples where D is used as the

key and timestamp t of current sample is used as the value:

$$\mathcal{H}[D] = t$$

Next, we define that a *revisit event* occurs when a hash collision happens, for instance, when a sample with a key D and a timestamp t arrives, the key D already exists in $\mathcal{H}$ with a value t' ($\mathcal{H}[D] = t'$). The *revisit time frame* $T_{revisit}$ of this event is defined as the time difference between two samples with the same discretized value D:

$$T_{revisit} = t - t'$$

Since we only use revisit events to identify reps, we set a threshold value $Thres_{revisit}$ as $6s$ to cover the longest possible rep. We only consider revisit events with $T_{revisit} < Thres_{revisit}$. $\mathcal{H}[D]$ is then updated with value t.

Finally, we define *revisit event generation rate* to represent the number of revisit events generated in the past $1s$ at a given time. A high revisit event generation rate suggests that revisit events with small revisit time frames are generated frequently within a short period of time and thus indicating the occurrence of repeating patterns, which can then identify an ongoing weightlifting session. We apply a threshold value $Thres_{rv}$ on the revisit event generation rate of each second to select significant (high) values.

The above revisit-based algorithm only takes $O(1)$ time to update the revisit event generation rate when a new sensor reading arrives and can significantly reduce computational cost compared with the $O(wL)$ overhead incurred for each new sample by the autocorrelation-based algorithm.

Figure 2.5 (g)(h)(i) plot results of our revisit-based algorithm on three different input traces shown in Figure 2.5 (a)(b)(c). The figures also show the threshold $Thres_{rv}$ used for repetition detection as well as both ground truth and detected weightlifting sessions. If input signals have cyclical patterns due to repeating motions, the revisit event generation rate raises and remains high until the end of current weightlifting session (shown in (g)). In

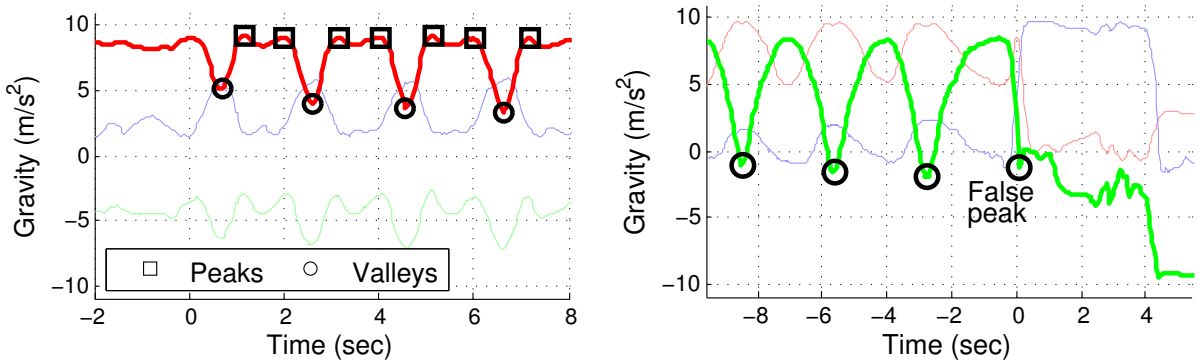


Figure 2.6: Two cases that can lead to failures of naive peak detection. Left: for bicep curl if we only consider upper peaks, each rep will be counted twice. Right: for ab crunch the weightlifting session boundary looks similar to a rep leading to a false count.

contrast, revisit event generation rates are much lower if there is no repeating pattern in the input trace (shown in (h)). However, whenever a user is stationary and the gravity readings are relatively constant, revisit events will be generated at a high rate equal to the sensor sampling rate. Our algorithm excludes this situation by discarding any sensor reading that has the same discretized value D as its immediate predecessor. As shown in Figure 2.5 (i), this effectively prevents stationary motions from being detected as weightlifting.

Rep counting: After the beginning and end of weightlifting sessions are detected, our revisit-based algorithm performs peak detection on raw gravity sensor data to count reps. However, this is more difficult than the peak detection step involved in the autocorrelation-based algorithm because noises in raw data are not filtered out and therefore we cannot simply count peaks for the number of reps.

There are three issues preventing an accurate rep counting using naive peak detection on raw gravity data. First, a dominant axis that presents the most distinct repeating patterns out of three gravity axes has to be selected. Second, neither upper peaks nor bottom peaks (valleys) always better indicate repeating patterns. As shown in Figure 2.6 left, simply counting peaks yields double-counting errors because upper parts of bicep curl reps sink in the middle resulting in vertically reversed w-shapes in the signal, i.e. two peaks. Third, the weightlifting session boundaries detected above may not be precise. Figure 2.6 right shows

that if we consider bottom peak (valley) values, the valley at $t = 0$ satisfies all constraints even if it is not a rep but the end of this session.

To address the first issue, we select the axis with the largest range between the maximum and minimum value within the session to reduce ambiguities during rep counting. Next, we have to decide whether the number of reps should be derived by counting peaks or valleys. We use vertical displacements within a small delta time (i.e. second derivatives) to capture spikiness of peaks/valleys where a larger displacement indicates a spikier peak/valley. For both peaks and valleys, we compute the average vertical displacement V and count the one with larger V as number of reps to improve the counting accuracy. In the example of Figure 2.6 left, valleys should be considered since they have larger vertical displacements than peaks. Finally, to reduce false rep counts from weightlifting session boundaries, we consider the three axes as a whole when ambiguity occurs at the beginning or end. Taking the example of Figure 2.6 right, the average reading of three gravity axes at $t = 0$ is different from any other detected valleys, allowing us to remove it as a false rep count.

Both the weightlifting detection algorithms require no model training. Because neither algorithm considers any user-specific feature, our proposed weightlifting algorithms are user-independent as well.

2.3.3.4 Weightlifting Type Classification

In routine weightlifting exercises, a user is not only interested in statistics such as the number of sets and reps, but also the type of weightlifting exercises performed in each set. Once weightlifting sessions are detected and numbers of reps are calculated, MiLift starts weightlifting type classification and labels each detected session with an exercise type.

Our intuition for weightlifting type classification is that wrist positions of a user during different weightlifting exercises will lead to unique orientations of a smartwatch. To quantify watch positions, MiLift aggregates 3-axis gravity sensor readings from the current weightlifting session and computes a set of features for each axis, including mean, standard deviation, minimum, maximum, and range. MiLift then applies an SVM classifier to label the type of

# of participants	22	Duration of non-workout	$14.88h$
# of weightlifting types	15	Duration of walking	$9.22h$
# of weightlifting sets	2528	Duration of running	$7.95h$
# of weightlifting reps	24408	Duration of weightlifting	$36.15h$
Total duration: $68.20h$			

Table 2.1: Summary of data collection in our user study.

the current session. We choose SVM here because of the relative simplicity of this classification problem and the ability of SVM to generate confidence scores for each class of types, which can be used to determine if an exercise type is unseen. To fine tune performance of the SVM classifier we have considered different kernels such as linear, Gaussian (with different radius r), and polynomial (with different degree d).

Another significant aspect of type classification is the ability to determine whether a new sample instance belongs to known exercise types during training or a new type of exercise. To identify new types, MiLift takes advantage of confidence scores reported by the SVM classifier. For each incoming instance, a vector of confidence scores is calculated by the SVM, representing the probability that this instance belongs to each known type class. If the maximum probability in this vector falls below a certain threshold $Thres_{conf}$, MiLift determines this instance belongs to a new type of weightlifting exercise and asks the user for a correct label. This also enables MiLift to perform active learning by re-training the model once a new exercise type is seen. We plan to address this topic in the future (Section 2.6).

2.3.4 Context-aware Optimization

To achieve both automatic segmentation (C2) and efficient resource usage (C4), MiLift takes advantage of available user contexts and applies a context-aware state transition mechanism. Figure 2.3 shows the transition among two classification states, *S1* and *S2*.

S1: MiLift executes the high-level activity classification in this state. It involves sampling of low-power accelerometer and incurs only lightweight computation and low energy consumption. Because a user typically keeps the same activity for a period of time and will not

switch activities frequently within seconds, MiLift duty-cycles the execution of S1 by a 20% schedule. MiLift classifies a $1s$ window every $5s$ so that it is able to capture most activity transitions while conserving battery energy. Note we design the 20% duty-cycle schedule as a system parameter and it can be changed based on user preferences and current battery states.

Moreover, to prevent everyday activities from being mis-classified as gym exercises (i.e. reducing false positives), S1 opportunistically leverages coarse location contexts such as those inferred from network accesses or WiFi signatures. For a given user, workout exercises mostly take place near similar locations (e.g. a gym) or WiFi networks (e.g. gym WiFi hotspots). MiLift can automatically learn the typical exercise location of a user after the first few app uses and from WiFi connection histories. If a coarse location is known from recent queries or WiFi scans initiated by the OS or other apps, it can be compared with the user's typical exercise locations. Once weightlifting is detected by S1, only when the two locations match will MiLift transit to S2. The opportunistic use of coarse locations as opposed to querying exact GPS coordinates ensures no additional energy is consumed.

S2: MiLift performs the more sophisticated weightlifting classification in this state. Although S2 is designed to be lightweight and efficient, the complexity of this state is relatively higher than S1 because of the computation incurred by each new sensor reading. Therefore it is triggered by S1 and only starts executing upon detection of weightlifting exercises. When no new weightlifting exercise is detected for a certain timeout (e.g. $5min$), MiLift transits back to S1 and performs the simpler high-level activity classification.

Although performing duty-cycled classification will not prevent the watch from being waken up frequently, i.e. the watch will still have to perform sensing and classification in S1 every 5 seconds, Section 2.5.2 and Section 2.5.3 have proven such an optimization to be effective since the watch can last for more than a full day with this two-stage classification model running continuously. This is because most of watch battery energy is spent on data sampling and computation rather than simply remaining power-on. Nevertheless, we acknowledge that the use of low-power co-processors (e.g. DSPs and dedicated sensor hubs) for always-on sensing and computation tasks will further reduce the energy consumption of

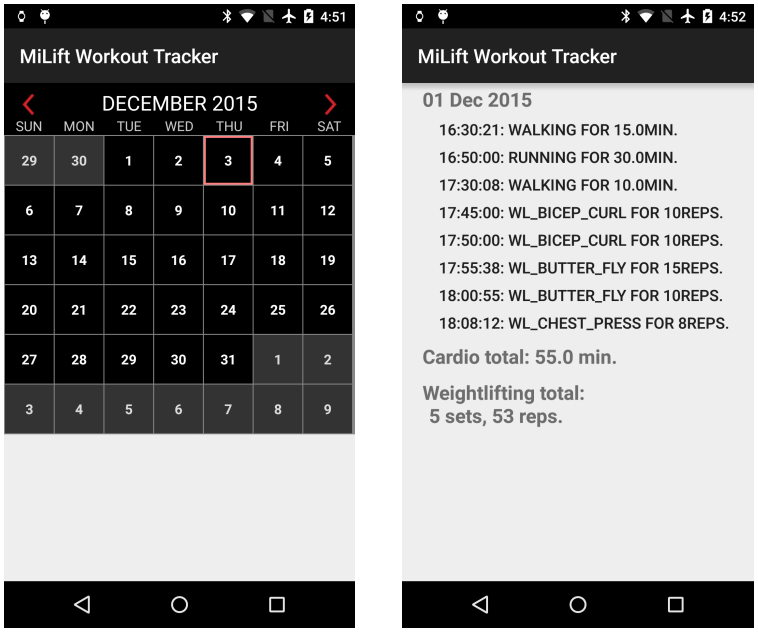


Figure 2.7: Screenshots of the MiLift Android app. Left: a calender for workout management; Right: a workout activity summary of a given date.

context inferences, as discussed in Section 2.6.

2.4 Implementation

Hardware device: The commercial off-the-shelf watches we used for data collection include Moto 360 [mota], Moto 360 Sport [motb], LG G Watch R [lgw], and ASUS ZenWatch 2 [asu]. For experiments on energy profiling we used the Moto 360 which has a 1GHz OMAP3 CPU, 512MB RAM, 4G flash storage, and a 3.8V/320mAh Lithium-ion battery. We have implemented a smartwatch data recording app that logs accelerometer and gravity sensors from the watch. Users were also assisted by Google Nexus 5 phones during data collection. Although we used four different watch models for data collection, the recording app implements the same sampling frequency even when running on different models.

User study: Our data collection campaign was performed as part of a user study that involves 22 participants[1]. Participants are aged from 19 to 27 and consist of both males and females. All participants regularly engage in gym exercises. We have collected 68.2 hours

[1]Our user study is approved by UCLA under IRB#16-000303.

of workout data in total, including 2528 weightlifting sets (24408 reps), as summarized in Table 2.1. For ground truth recording, we used the Moves app [mov] to log cardio activities and let participants manually record set/rep counts in weightlifting exercises. In addition, we asked each participant to finish a post-completion survey on workout tracking (Section 2.5.3).

Offline analysis: With the collected data, we first conducted an offline analysis to train classification models. The high-level activity classifier and the weightlifting type classifier were implemented using the scikit-learn library [PVG11] in Python. The HMM smoothing of high-level activities and weightlifting classification algorithms including session detection and rep counting were implemented in Matlab.

Android implementation: We implemented MiLift as an Android app using Android 5.1.1 (API 22). MiLift contains two components: a watch app that performs workout tracking and a phone app that provides visualization and assistance for workout management. The watch app executes real-time classifications as a background service and does not require user inputs. Multiple open-source projects were used to port trained classification model to Android, including CRF++ [crf] and jahmm [jah]. Figure 2.7 left shows the MiLift workout calendar where users can choose a particular day to view their progresses. Figure 2.7 right displays workout activities performed in a given day showing durations for cardio workouts and rep counts for weightlifting exercises.

2.5 Evaluation

We first evaluate MiLift using a set of micro-benchmarks including accuracy (Section 3.4.4.3) and power (Section 2.5.2) profiling of classification models. We then use a macro-benchmark to analyze the effect of MiLift on required user tasks and smartwatch battery lives compared with previous approaches (Section 2.5.3).

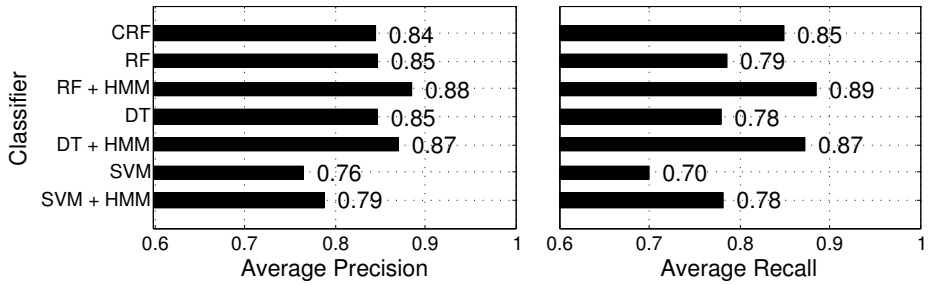


Figure 2.8: Weighted average precision and recall of high-level activity classification (10-fold cross validation).

2.5.1 MiLift Tracking Accuracy

2.5.1.1 Accuracy of High-level Activity Classification

The high-level activity classifier is the first state of the two-stage classification model and aims at detecting high-level activities of users before triggering the weightlifting classifier. To select the best model for this classification task, we first used a 10-fold cross validation to examine the performance of the four classifiers, including the continuous graph model CRF and three instance classifiers RF+HMM, DT+HMM, and SVM+HMM. The best parameters selected for each classifier include maximum depth of 8 for DT, 64 classifiers and 4 features used at each split for RF, and linear kernel for SVM. For CRF, input feature values are first transformed to integers using cluster centers trained from k-means clustering algorithm, and our parameter tuning has chosen the use of 16 clusters for best CRF performance. Figure 2.8 plots the weighted average precision and recall for all four models from the cross validation, including results before and after HMM smoothing for the three instance classifiers. Among the four models DT+HMM and RF+HMM has the best overall performance (nearly 90%) but CRF only performs slightly worse (around 85%). As discussed in Section 2.3.2, the results suggest that HMM smoothing is crucial in reducing false positives and increasing the overall performance of all three instance classifiers.

We then used the best model parameters selected from cross validation to showcase real-world performances of the high-level activity classifiers. We had one user collected an all-day data trace consisting of 15 hours of continuous accelerometer data. This trace

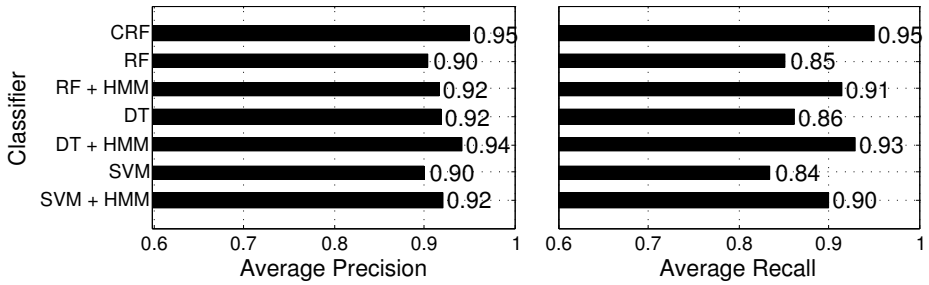


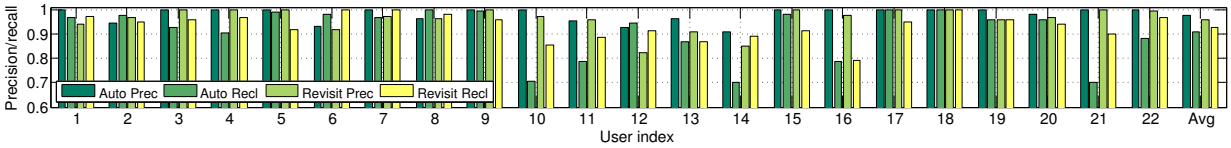
Figure 2.9: Weighted average precision and recall of high-level activity classification (15-hour all-day trace).

CRF	**RF+HMM**	**DT+HMM**	**SVM+HMM**
1.6 MB	105.1 MB	45 KB	5.6 MB

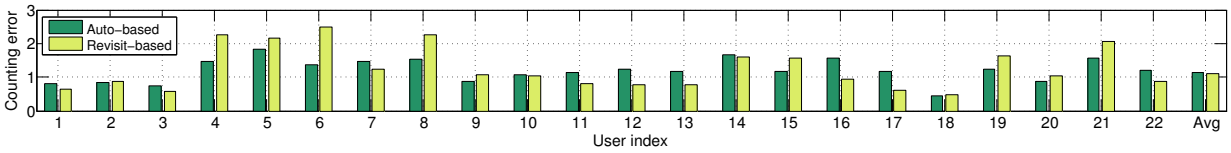
Table 2.2: Memory footprints of high-level classifiers.

includes the user walking around school, sitting during classes and study, running in a gym, and performing weightlifting exercises. Our entire dataset described in Section 2.4 is divided into two parts: all but this 15-hour data trace is used to train the high-level classifiers and this trace itself is used as the testing set. Figure 2.9 plots the resulting weighted average precision and recall for classifications on the all-day trace. All four models achieve above 90% average precision and recall. Among them, DT+HMM and CRF have the best overall accuracy but the other two models perform only slightly worse. The results above have proven that the high-level activity classification models in MiLift can successfully separate non-workout activities from gym exercises in real-world user scenarios. Moreover, continuous graphical models (e.g., CRF, HMM) can effectively increase the accuracy of time-series classification tasks by exploiting temporal correlations in data.

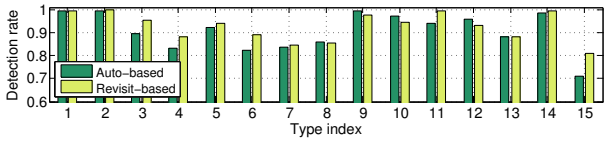
Finally, with RAM becoming another power-hungry component in mobile SoC as RAM power can exceed CPU power in certain workloads [CH10], another factor in choosing classification models is the memory footprint. Table 2.2 compares the size of the four trained models. DT+HMM and CRF have much smaller model sizes compared with the other two because of the relative simplicity of their model structures. Considering both their classification precision/recall performances and the memory footprints, we have chosen to use



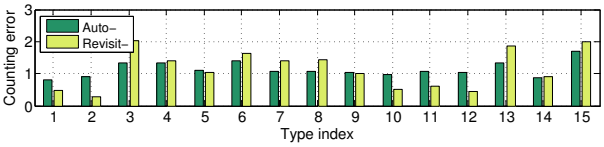
(a) Session (set) detection accuracy based on users.



(b) Rep counting error based on users.



(c) Session (set) detection accuracy based on types of exercises.



(d) Rep counting error based on types of exercises.

Figure 2.10: Weightlifting detection performance, reported by users and by types of exercises (as shown in Figure 2.1).

DT+HMM and CRF for implementation in MiLift.

2.5.1.2 Accuracy of Weightlifting Classification

Session (set) detection and rep counting: We first evaluate the accuracy of weightlifting session detection and rep counting using our two proposed algorithms. Figure 2.10 (a) illustrates the weighted average precision and recall of session (set) detection based on users. In general, both algorithms demonstrate high overall accuracy, as autocorrelation-based approach achieves 97.5% precision and 90.7% recall while revisit-based approach yields 95.7% precision and 92.6% recall. Figure 2.11 shows three common cases seen in our user study that lead to errors in weightlifting set detection: (a) some users tend to take short pauses within a set, possibly because of tiredness; (b) users can sometimes finish the last rep in an incorrect posture; (c) some users may adjust their body postures (including wrist orientations) during a set. All three cases can cause irregular patterns in sampled gravity data and set detection errors from both approaches.

For rep counting, both the autocorrelation-based and the revisit-based approach have

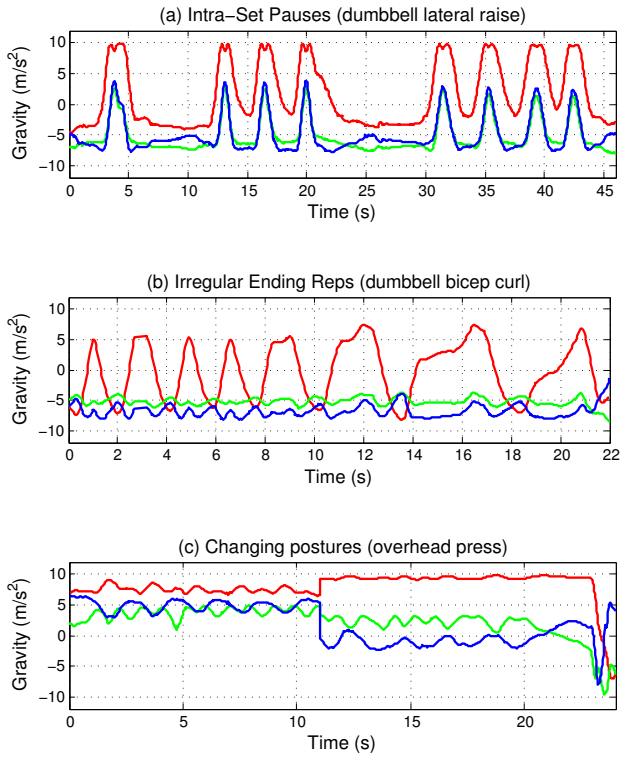


Figure 2.11: Three common error sources of weightlifting session (set) detection seen in our user study.

similar average errors, shown in Figure 2.10 (b). Here the rep counting error is defined as the difference between the true number of reps and the number of reps counted by our algorithms in each set. On average the autocorrelation-based algorithm and the revisit-based algorithm report errors of 1.125 and 1.122 reps per set, respectively. Some rep counting errors are caused by cases shown in Figure 2.11, because inaccurately detected session (set) boundaries can lead to rep miscounts as well. Other errors are results of unconventional postures and actions performed by some users during weightlifting exercises. Participants can sometimes incorrectly log ground truth leading to errors as well. Nevertheless, the rep counting errors are insignificant considering a user performs 9.65 reps on average within each set. In addition, the errors are only slightly greater than user expectations from our survey (Table 2.4).

To study the root cause of errors, we report session (set) detection and rep counting results based on the type of weightlifting exercises. Figure 2.10 (c) plots the recall or detection rate

of each exercise type shown in Figure 2.1. Note that the precision metric is not applicable here since a workout trace may contain multiple types of exercises and therefore we cannot identify false positive sessions for each exercise type. Both algorithms can nearly perfectly identify all sessions of bicep curl (#1), tricep extension (#2), tricep dip (#9), and dumbbell lateral raise (#14). However, seated row (#6), pec fly (#7), and rear deltoid (#8) are not well captured by either approach. This is because the vertical displacements in gravity data, i.e. the range between peaks and valleys, are small in all three axes for these exercises, making both algorithms susceptible to noises. Seated row does not involve substantial wrist motions and will not affect gravity readings much. Though both pec fly and rear deltoid require large forearm movements, the wrist trajectories of users fall into a horizontal plane and the watch rotates around z-axis in the global frame (perpendicular to the ground). Such rotation will not cause significant changes in gravity readings.

Although free weight exercises can in general cause more complicated body movements than machine-based exercises, MiLift's performances on session detection and rep counting for free weight exercises are similar to those for machine-based exercises. For free weight exercises, MiLift has high errors detecting dumbbell bench press (#15) due to similar reasons stated above. It is also worth noting that keeping body motions stable while performing dumbbell bench press exercises is known to be difficult even for experienced gym users, which adds more noises to the sampled gravity data.

Figure 2.10 (d) reports rep counting errors by exercise types and shows similar error trends as seen in Figure 2.10 (c): type #6-#8 and #15 have higher rep counting errors than others. Lat pulldown (#3) and dumbbell single arm row (#13) also have relatively high errors due to unclear repetition patterns shown in gravity data traces from some users.

Weightlifting Type Classification: We have chosen a Gaussian SVM with $r = 0.05$ for MiLift to determine the type of weightlifting exercises in a session and identify unseen exercise types. We used a 10-fold cross validation on the entire weightlifting dataset to test the performance of this classifier, which shows a precision of 89.71%, a recall of 89.53%, and an F1-score of 89.48% (all weighted average) for all 15 exercise type classes. Figure 2.12 left plots the confusion matrix of all exercise types from cross validation, suggesting that

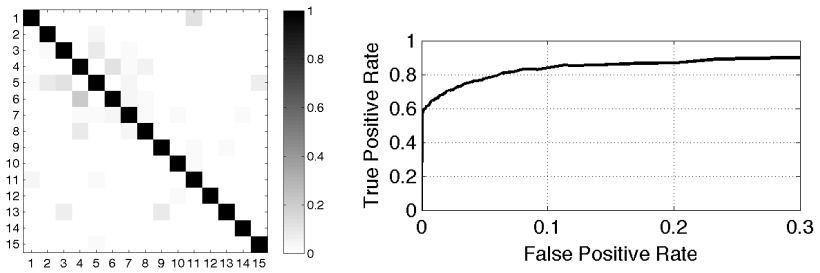


Figure 2.12: Left: confusion matrix of weightlifting type classification. Right: ROC curve of leave-one-type-out experiments.

our SVM model has good performance on all classes and does not bias towards certain weightlifting types.

Another important benchmark of weightlifting type classification is the accuracy of identifying newly unseen types. Section 2.3.3.4 proposes our algorithm in MiLift to identify new types by applying a threshold value $Thres_{conf}$ on confidence scores generated by the SVM. We evaluated the performance of this approach by conducting a leave-one-type-out cross validation. For each weightlifting type, we trained an SVM model using data from all other 14 types, and used this model to classify the entire dataset including instances of the 14 known types and the 1 unknown type. Because an instance of an unknown type can be similar to instances of one or more of the known types, the accuracy of identifying unknown types depends on the value of $Thres_{conf}$. A smaller $Thres_{conf}$ will allow more instances of unknown types to be correctly identified but will also lead to more false positives. Figure 2.12 right plots the ROC curve (true positive rate over false positive rate) of identifying unknown types obtained by adjusting $Thres_{conf}$ from 0.1 to 0.8. The results illustrate our SVM model can achieve a true positive rate of 85% with about 10% false positive rate indicating an effective classifier in determining new types.

The micro-benchmark results show that MiLift can accurately track both cardio and weightlifting workouts (C3).

2.5.2 Energy Efficiency of MiLift Models

Table 2.3 compares the average power consumptions of MiLift at different classification states and the battery life estimations if each state is executed continuously on a Moto 360 smartwatch. We also measured the sleeping power of the watch (S0) for comparison. There are two conclusions we can draw from the results:

First, the two high-level classification models DT+HMM and CRF are energy efficient after the 20% duty-cycle optimization discussed in Section 2.3.4. If only executing these two models, the watch battery can last for more than 16 hours meeting the efficiency challenge (C4). This is achieved even with the duty-cycle executions frequently waking up the watch, indicating that sampling and classification are more energy hungry for the watch than remaining awake. Out of the two models, DT+HMM is about 25.96% more energy efficient than CRF in terms of watch battery life because of DT's lower classification complexity than CRF.

Second, in terms of watch battery life, our revisit-based weightlifting classification algorithm is 41.34% more energy efficient compared with the autocorrelation-based approach, due to less computation incurred by each new sensor sample ($O(1)$ compared with $O(wL)$ as discussed in Section 2.3.3). Note that our micro-benchmarks estimate watch battery lives assuming each state is executed continuously. During real-world executions, weightlifting classification S2-A or S2-R does not need to be executed continuously but will only be triggered by the high-level activity classifier S1-D or S1-C. Section 2.5.3 presents an empirical study showing how our two-stage model improves the battery life of smartwatches.

2.5.3 User Task and Battery Life Analysis

To evaluate MiLift in real-world scenarios, we conducted a macro-benchmark by empirically comparing MiLift with previous approaches using two metrics: (1) number of tasks a user needs to perform for accurate workout tracking, and (2) the battery life of a smartwatch running continuous tracking. We assumed the watch is used only during the day ($16h$), and the user performs weightlifting exercises for 12.5% of the time every day ($2h$). Each approach

State	Description	Power (mW)	Battery Life (h)
S0	Sleep	13.10	97.47
S1-D	High-level activity classification (DT+HMM)	47.08	25.83
S1-C	High-level activity classification (CRF)	58.83	20.67
S2-A	Weightlifting classification (autocorrelation)	159.58	7.62
S2-R	Weightlifting classification (revisit)	112.91	10.77

Table 2.3: Average power consumption and battery life benchmarks of Moto 360. Showing battery life estimations if executing each state continuously.

is described as follows:

Baseline app 1: To continuously track cardio workout activities, app 1 runs an always-on cardio activity classifier during the entire day. However, users have to manually start and stop weightlifting classification before and after each weightlifting session (12.5% of the day). Users need to manually select workout types and manually count sets/reps for weightlifting exercises as well. This app is similar to previous mobile workout tracking apps.

Baseline app 2: To reduce manual input from users, app 2 continuously executes both the cardio activity classifier and the weightlifting classifier all day. Users still have to manually select workout types and manually count sets/reps for weightlifting exercises. This app is similar to existing weightlifting tracking systems.

Baseline app 3: The VimoFit workout tracking app [vim] executes continuously for the entire day. Users need to manually start and stop tracking but the type of exercises and set/rep counts can be automatically determined.

MiLift: MiLift automatically segments exercises from non-workout activities and does not require any manual input from users. It uses a two-stage classification model to duty-cycle the executions for efficient resource usage.

We used power profiles from Section 2.5.2 to estimate watch battery lives for baseline app 1, app 2, and MiLift. For app 3 (VimoFit) we performed a separate experiment to log its battery usage. For app 1, app 2 and MiLift, we used the more efficient DT+HMM approach as the high-level (cardio) activity classifier. App 1 and 2 implemented the more tra-

Linear Scale Question (1 to 5)	**Score**
Do you find it useful that MiLift can automatically detect ongoing exercises (cardio and weightlifting)?	4.13 ± 0.83
Do you find it useful that MiLift can detect the type of exercises you are performing?	4.47 ± 0.74
Do you find it useful that MiLift can automatically detect and count weightlifting sessions (sets)?	4.67 ± 0.62
Do you find it useful that MiLift can count number of reps (repetitions) in weightlifting exercises?	4.73 ± 0.59
Short Answer Question	**# of Rep**
What is the max rep counting error you can tolerate?	0.8 ± 0.49

Table 2.4: Survey questions and answers. For scale questions, 1 (5) stands for strongly disagree (strongly agree).

ditional autocorrelation-based algorithm for weightlifting detection, while MiLift considered the revisit-based algorithm.

User tasks: We first present the results of our user survey (Section 2.4) in Table 2.4. Users find the features of MiLift valuable, including automatic exercise segmentation, exercise type detection, weightlifting set detection, and weightlifting rep counting. This proves that tracking approaches without automatic exercise segmentation can be less attractive to active exercisers. Next, Table 2.5 compares the user tasks required in different approaches, suggesting that all three baseline apps rely on certain user tasks to accurately track exercises. Although baseline app 3 can automatically detect types of weightlifting exercises, the detection accuracy was poor. In our experiments, VimoFit was only able to correctly classify 3 of the 15 exercise types. In contrast, MiLift removes the burden on users and can provide fully autonomous workout tracking for both cardio and weightlifting exercises.

Energy efficiency: MiLift can extend watch battery life by 18.25% (3.31h), 241.56% (15.17h), and 824.57% (19.13h) compared with these three baseline apps, respectively. For baseline app 2 and app 3 the watch battery will run out before the end of the day forcing users to charge the watch. The VimoFit app used for baseline app 3 currently also keeps the watch screen on during executions therefore greatly exacerbating its power consumption. The energy saving of MiLift is achieved by the low-power weightlifting detection algorithm

Approach	User Tasks	Battery Life
Baseline 1	(1) Manually start/stop weightlifting sessions; (2) Manually select workout types; (3) Count weightlifting sets and reps.	$18.14h$
Baseline 2	(1) Manually select workout types; (2) Count weightlifting sets and reps.	$6.28h$
Baseline 3	(1) Manually start/stop weightlifting sessions.	$2.32h$
MiLift	None	$21.45h$

Table 2.5: User task and watch battery life comparisons of MiLift and baseline approaches.

and the context-aware optimization.

The macro-benchmark suggests that MiLift can significantly better preserve battery power of smartwatches than previous approaches (C4) and remove user burdens (C2).

2.6 Discussion

We discuss potential improvements for future work:

Sensing scope and system generalization: Although MiLift can detect a variety of weightlifting exercises including both machine workouts and free weights, exercises which do not involve wrist motions cannot be tracked. This includes both single-arm exercises not performed on the watch-wearing hand and other types of exercises such as leg-based ones. We argue that incorporating other non-intrusive sensors such as shoe motion sensors can cover more exercise types. Beyond cardio and weightlifting exercises, algorithms proposed in MiLift can be generalized to detect any exercise or human activity that involves repeating motions, including but not limited to rock climbing, hiking, and racket sports. However, tracking exercises with mild body movements such as yoga may require coordination of multiple sensors.

Energy optimization: We plan to leverage strategies proposed by prior work to further optimize the energy consumption of MiLift, such as offloading sensing and prepossessing from the main processor in mobile SoCs to low-power co-processors [PLL11, GLR14, LGQ15], exploiting the coordination of multiple mobile devices and the cloud [MVS15, SSP16, CZW15],

and exploring the correlation among possible user contexts [Nat12] other than the coarse location used in this work. These optimization techniques will help save energy budgets of smartwatches for other possible workloads.

Active learning: The weightlifting type classification in MiLift can determine if an instance belongs to an unknown type class. However, currently MiLift does not use such new instances to reinforce its classification model. We plan to implement an active learning system similar to [CSG13] where MiLift would query users for a ground truth label whenever it detects a new type of exercise and then improve the model.

Weight tracking and quality assessment: Prior work has shown that incorrect usages of weightlifting equipment can lead to ineffective training or even injuries. Researchers have proposed several metrics to indicate exercise quality [SBV11, DSY15]. Currently MiLift cannot track the amount of weights used in each exercise. However, we observed that users demonstrate different inertial patterns with different weights. This includes changes of peak-to-peak intervals and noise patterns in the accelerometer/gravity data traces. Therefore a similar quality assessment module can be added to MiLift to track weights and to provide exercise feedbacks.

2.7 Summary

In MiLift the combination of a two-stage classification model and a lightweight weightlifting detection algorithm has enabled autonomous and efficient tracking of both cardio and weightlifting workouts. MiLift automatically segments exercises from non-workout activities so that users do not need to manually start/stop tracking or select exercise types. Our evaluations indicate that MiLift can achieve above 90% precision and recall for tracking both cardio workouts and weightlifting exercises. MiLift can also extend the battery life of a Moto 360 watch by up to 8.25× (19.13h) compared with previous approaches. Finally, our user study suggests MiLift's automatic segmentation ability is valuable to individuals actively engaging in gym exercises.

CHAPTER 3

Accuracy and Energy Optimization of Context Inferences

After proposing the design and implementation of an example context inference app that leverages smartwatches, we then describe a set of efforts to optimize the inference accuracy and energy consumption of context-aware apps. Observations from these optimizations can be adopted by developers today to improve the context inference executions in their apps.

3.1 Overview and Challenges

As proposed in Chapter 1 the development and execution of context-aware apps today are accompanied by myriad challenges, including poor inference accuracy as a result of limited sensor coverage, high energy consumption, and monolithic development practice. In this chapter, we discuss the first two challenges in detail, while Chapter 4 presents our effort in creating a programming framework for context-aware inference apps.

3.1.1 Inference Accuracy

One commonly used metric to quantify the performance of a context inference model is the inference accuracy. Accuracy is used as a statistical measure of how well a classification model correctly identifies or excludes a condition. It is the proportion of true results (both true positives and true negatives) among the total number of cases examined [Met78]. Other popular metrics include precision, recall, and f-score [Faw06, OD08].

We summarize challenges that limit the inference accuracy of context-aware apps as

follows:

- **Model performance largely depends on hand-picked features (I-1).** Traditional machine learning models used in context inferences take calculated *features* rather than raw sensor data as inputs due to their inability to process high-dimensional data. This requires developers to select a set of aggregation functions, i.e. features or attributes, as the correct representation of the data. For example, human activity classification models typically aggregate 3-axis accelerometer data using features such as mean, variance, Discrete Fourier Transforms (DFTs) [MSS06, RMB10]. This process, also known as the feature engineering, is often critical for achieving high model accuracy but requires domain expertise and experience. Features today often contain near human-level understanding of the data, and therefore the process of selecting the best features often requires complicated algorithms such as these presented in prior work [KJ97, GE03]. In summary, the performance of a machine learning model (e.g. inference accuracy) will largely depend on the success of feature engineering and feature selection. Failures to pick appropriate features or to select top ones can significantly reduce the inference accuracy of a model.

- **Single device sensing provides limited coverage on users (I-2).** The performance of a model, such as accuracy, precision, and recall, also depends on the relative position of the phone to the user [PPC12]. For instance, most activity recognition apps rely on inertial sensor readings and require the user to carry the smartphone. Whenever the user leaves the phone elsewhere (e.g. on a desk) to focus on other tasks or to charge the phone, the sensors cannot capture any meaningful data, resulting in *limited sensor coverage* in terms of their ability to capture human movement and therefore poor inference accuracy. During those periods of time the user would still perform various kinds of activities, but the inference app on the phone would not be able to report correct activity label, and would normally classify the user as not moving.

3.1.2 Energy Consumption

A large amount of mobile and embedded devices used for pervasive context inferences are battery-powered and have limited energy budget due to their restricted dimensions. This calls for a detailed study and optimization of energy consumption of context-aware apps.

The problem of high energy consumption seen in some context inference apps today can be attributed to the following causes:

- **Continuous executions and use of energy-hungry sensors (E-1).** Context inference apps often need to run in the background continuously to monitor user contexts and behaviors for just-in-time feedback, notifications, and interventions. This situation is usually exacerbated by the use of energy-hungry sensors like GPS and cellular radio. For example, Ju et al. [JLY12] reported that apps consume 4% - 22% of CPU cycles for inference executions, and Kansal et al. [KSB13] observed a reduction of phone standby time from 430 to 12 hours when an app is continuously using the Android proximity API.

- **Inability to use specialized low-power co-processors (E-2).** Currently most always-on context-aware apps run on the main app processor of smartphones representing a significant portion of the phone's overall workload and energy consumption. Modern mobile platforms are sophisticated system on chips (SoCs) where the main app processors are complemented by multiple co-processors. Recently, chip vendors have undertaken efforts to make these previously hidden co-processors such as the Graphics Processing Unit (GPU) and the Digital Signal Processor (DSP) programmable. However, today's alway-on context inference apps do not leverage these specialized low-power co-processors and therefore must keep the main app processor awake for sensing and inference tasks.

- **Complex feature computation and classification algorithms (E-3).** The algorithms used in context-aware inference apps, including feature computations and classifications, are computationally intensive and energy hungry. In particular, feature

computation works on high-dimensional raw sensor data and can be more time- and energy-consuming than classifications. For instance, one of our proof-of-concept implementations performing activity recognition using Support Vector Machine (SVM) shows that feature extraction takes about 10× more time compared to the SVM classification. The high power consumption of context-aware apps and the reduction in battery life as a result will only get worse as apps with even more sophisticated context inference capabilities emerge.

3.1.3 Design Choices

We describe a set of efforts to maximize the inference accuracy and to minimize the energy consumption of context-aware apps.

- **Applying deep learning algorithms** We propose the use of deep learning, such as Recurrent Neural Networks (RNN), for context inferences. Using workout tracking as an example, we have shown that RNN can work on high-dimensional raw data for classification and therefore eliminates the need for feature engineering. RNN helps remove the dependency of model accuracy on successful feature selection (Challenge I-1) and avoid feature computation at runtime (Challenge E-3) while achieving similar inference accuracy compared with previous classification models. This part is discussed in Section 3.2.

- **Exploiting low-power co-processors** We explore the energy and performance implications of off-loading computation from a main app processor (CPU) to a low-power co-processor available in mobile System-on-Chip (SoC) architectures (Challenge E-2). We study the off-loading of the computation associated with machine learning algorithms in context-aware apps, such as classification used in traditional machine learning models and deep learning models, to a Digital Signal Processors (DSP) or a Graphics Processing Unit (GPU), respectively. We have implemented several frequently used algorithms and measured their performance and energy profiles on GPUs and DSPs. Our results indicate that off-loading to the DSP reduces energy consumption by up to

60% with negligible effect on application latency, while off-loading to GPU can result in up to 30× speed-up in executing deep learning workloads. This part is discussed in Section 3.3.

- **Exploring watch-phone coordinations** We investigate whether the coordination between watches and phones can be leveraged to tackle the challenges of context-aware inferences. We quantify the benefit of using both the watch and the phone for context inferences. Based on an example user daily routine, we achieve up to 37.4% improvement in inference accuracy due to the increased device sensor coverage made possible by the watch (Challenge I-2). Moreover, we report the energy saving of running inferences across both the watch and the phone. We demonstrate up to 67.3% less energy consumption by using an optimal partitioning of inferences between devices, and up to 61.0% less by replacing the high-power phone GPS with the low-power watch accelerometer (Challenge E-1).

The rest of this chapter describes our design details and resulting improvements from the above efforts.

3.2 Applying Deep Learning for Mobile Context Inferences

3.2.1 Overview

The performance of traditional machine learning algorithms depends heavily on the *representation* of the data they are given. Each piece of information included in the representation is known as a *feature* [GBC16]. As discussed in Section 3.1, the process of selecting features is often critical for achieving high model accuracy but requires domain expertise or even complicated feature selection algorithms. *Deep learning* solves this problem by introducing representations that are expressed in terms of other, simpler representations. Deep learning enables building complex models directly from raw data or very simple representations while learning the right representations of data in the process.

There are two important research questions in terms of applying deep learning for mobile sensing and inferences tasks:

- *Can we achieve comparable inference accuracy as traditional models?* We want to use deep learning to avoid hand-picking features, but still would like comparable accuracy as previous approaches.
- *Can we get desired performance running deep learning on mobile devices?* Deep neural networks, especially Recurrent Neural Networks (RNN) can be much more complex than traditional models. We need to perform a detailed benchmark on whether running these models on mobile devices can meet the real-time requirement of inferences.

3.2.2 Task: High-level Activity Classification

We use MiLift's high-level activity classification (Section 2.3.2) as the example task for benchmarking deep learning on mobile sensor data. The task is making classifications about a user's high-level activity, including walking, running, weightlifting, and still, from 50Hz smartwatch accelerometer data. This can be viewed as a sequential classification task on time-series data. Vanilla neural networks typically take fixed-sized input vectors and generates fixed-sized output vectors with a fixed amount of computation. They are not suitable for classifying time-series data because each input only looks at individual data vectors without considering the temporal correlation among the data.

To better classify sequential data, we turn to Recurrent Neural Networks (RNN) [Pin87, Pea89, WZ89, FN93]. RNNs have traditionally been used for mining temporal pattern from time-series data, such as building market models [Moz89] [Wer88] and natural language processing tasks [BDV03] [MKB10]. Recently RNNs have also been used for mobile sensing tasks such as keystroke detection [BR15] and activity recognition [HHP16]. RNN is a natural fit for our high-level activity classification task because exercise activities are continuous in time rather than isolated samples. RNN can operate on sequences of vectors [Kar15] such as the accelerometer data stream with timestamps in this task and leverage temporal correlations.

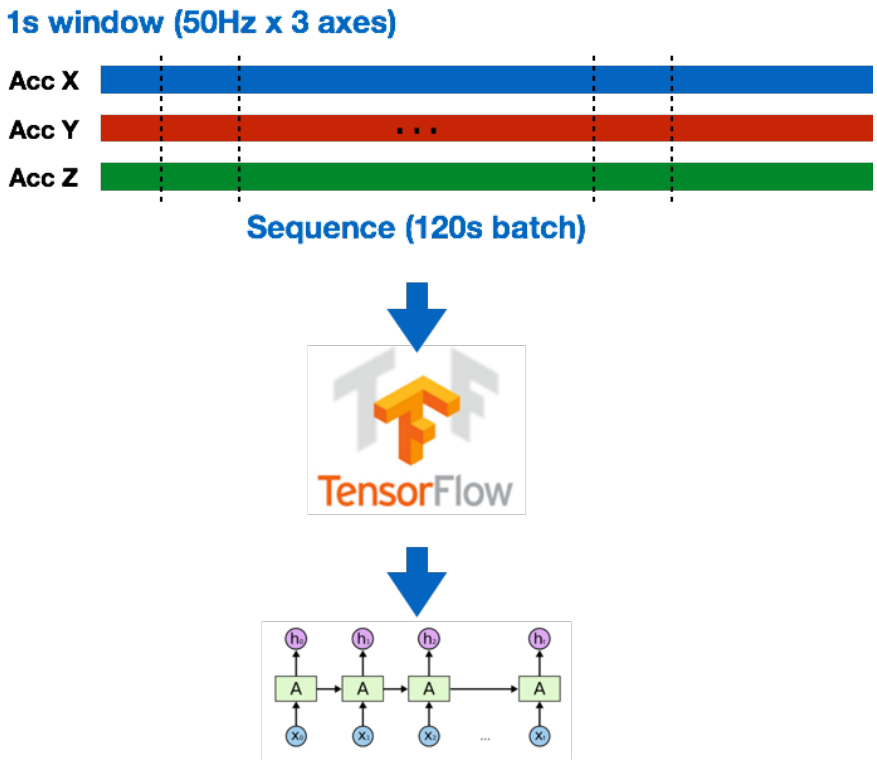


Figure 3.1: Training workflow of using RNNs for high-level activity classification.

3.2.3 Training Workflow

Figure 3.1 shows our RNN training workflow, with the main steps described as follows:

1. We apply 1 second windows on 3-axis accelerometer data at 50Hz (150 samples per second).

2. To leverage the sequential information, we put 120 windows as a batch to train and test the RNN.

3. We have tuned the RNN to use the following parameters: 4 x 128 hidden layers, cross-entropy objective function, SGD optimizer, softmax activation, and L2 regularization.

4. Finally the RNN is trained and saved by Tensorflow [ten].

Using our entire dataset collected for MiLift (68 hours), training of the above RNN model takes about 40 minutes on an NVIDIA TITAN X graphics card.

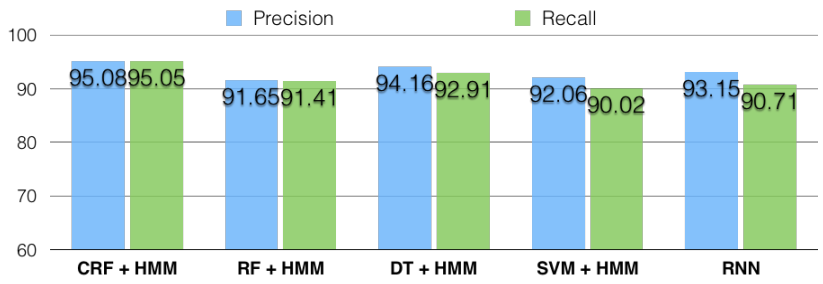


Figure 3.2: Classification precision and recall comparison of RNN and traditional models.

Inference Pipeline	Latency (1s window)
Features + Decision Tree + HMM	1 ms
Features + CRF + HMM	10 ms
RNN	600 ms

Table 3.1: Latency comparison of RNN and traditional models.

3.2.4 Evaluation

Our first research question is on the inference accuracy of deep learning models. With the trained RNN, we now compare it with previous models using the metric of weighted average precision and recall on high-level activity classifications, with the results shown in Figure 3.2. RNN has achieved comparable accuracy even without requiring hand-picking features. Note that the focus of this work is not on fine tuning the performance of RNNs. Although its accuracy results are lower than the best performing CRF, we can definitely apply other techniques from ML community to further improve the RNN.

The second question is whether RNN can meet the real-time latency requirement. We implement an end-to-end example of running RNN on real-time sensor data using a Nexus 5X phone. In terms of model size, because we have a relatively small number of hidden layers, the RNN model is only around 500KB and is even smaller than CRF and RF. However, as we can see from Table 3.3, the much more complex RNN has led to a significant slow-down in classification latency (i.e. time to classify a 1 second window). This is mostly because mobile CPUs today are not optimized to execute highly parallel tasks such as RNN classifications. Although the latency still fall in the one-second window size and will not cause any notable

Mobile SoC	App Processor (CPU)	Co-processor	Used By
Qualcomm Snapdragon 821	Dual Kyro 2.15GHz + Dual Kyro 1.59GHz	Hexagon 680 DSP, Adreno 530 GPU	Google Pixel, Pixel XL
Apple A10	Dual Hurricane 2.34GHz + Dual Zephyr (ARMv8-A)	M10 Motion Processor, 6-core GPU	Apple iPhone 7, iPhone 7 Plus
Samsung Exynos 8890	Quad Exynos M1 2.3Ghz + Quad Cortex-A53 1.6GHz	ARM Mali-T880 GPU	Samsung Galaxy S7, Galaxy Note 7
HiSilicon Kirin 950/955	Quad Cortex-A72 2.5GHz + Quad Cortex-A53 1.8Ghz	ARM Mali-T880 GPU	Huawei Mate 8, Huawei P9
NVIDIA Tegra X1	Quad Cortex-A72 1.9GHz + Quad Cortex-A53 1.3Ghz	Maxwell GPU	NVIDIA Shield TV, Google Pixel C

Table 3.2: Specifications of latest mobile SoCs.

delays to the user. The slow-down motivates us to seek leveraging mobile GPUs for better performance. On a desktop, using the Titan X for RNN tasks can yield a 30× speed-up compared with a quad-core i7 CPU. Similarly, we expect certain degree of speed-ups if the classification can be executed on mobile GPUs.

3.3 Exploiting Co-Processors in Mobile SoCs

In this section, we describe two optimizations of leveraging heterogeneous cores in mobile SoCs, including off-loading deep learning tasks to mobile GPUs, and executing the classification phases of traditional machine models on mobile DSPs [SCR, SCC14].

3.3.1 Background: Processor Heterogeneity

To motivate the benefits of leveraging processor heterogeneity, we start by showcasing the specifications of the latest mobile SoCs. Mobile processors are no longer simply app processors, but sophisticated system-on-chips (SoCs) where the main app processors are complemented by a set of heterogeneous processors. Table 3.2 summarizes the specifications of mainstream mobile SoCs today. The types of heterogeneity in mobile SoCs can be summarized as loosely-coupled or tightly-coupled.

Loosely-coupled: ARM developed big.LITTLE [arm], which couples relatively slower, lower-power processor cores with more powerful and power-hungry ones. The intention being to create a multi-core processor that can adjust better to dynamic computing needs and use

less power than clock scaling alone. The big.LITTLE is a tightly-coupled heterogeneous architecture because the big cores and the little cores typically have cache coherence and share the same instruction set architectures. Nevertheless they have different power-performance operating points. As shown in Table 3.2, major mobile SoC manufacturers today all adopt the big.LITTLE architecture.

Tightly-coupled: Mobile SoCs also include a set of embedded processors such as Digital Signal Processors (DSPs) and Graphic Processing Units (GPUs) that handle specialized work such as media processing and low-level sensor I/O. These loosely-coupled heterogeneous processors typically have no cache coherence with the app processor, and have different instruction set architecture with app processors. The co-processors are usually hidden from the app developers and are instead limited to running prebuilt firmware provided by the platform manufacturer. Recently, conscious efforts are being undertaken by various mobile processor vendors to expose the co-processor heterogeneity to the app developers and provide them with the ability to program the DSPs. The Qualcomm Hexagon DSP [hex] is a representative example of such an embedded processor on mobile SoC. It comes with custom programmability and operates in the ultra low power range.

We use the Hexagon DSP in Qualcomm Snapdragon and the GPU in NVIDIA Tegra in this work as examples to showcase the benefits of leveraging processor heterogeneity.

3.3.2 Using Mobile GPUs for Deep Learning

3.3.2.1 Overview

As described in Section 1.2.3, today there are several different architectures of leveraging mobile GPUs for deep learning. We choose the CUDA approach [cud] by running it on an NVIDIA Jetson development board with the NVIDIA Tegra processor [NVI] because the GPU in Tegra fully supports and is optimized for CUDA. Looking back at our RNN training workflow, after we export the model weights and topology, the mobile GPU can load the files and instantiate the RNN model. With this model, the GPU can then perform classification on sensor data.

Case	GPU vs CPU Speed-up	Projected Latency
RNN classification with CUDA	15×	40ms
BLAS with Renderscript	30×	20ms

Table 3.3: Speed-ups of running RNNs on a mobile GPU vs CPU.

3.3.2.2 Evaluation: Performance Speed-ups

Since the Jetson board does not support accessing real-time sensor data on its mobile GPU, we have created two example cases to demonstrate the speed-up of running RNN classification on the GPU compared with on the CPU.

1. We have implemented a complete RNN classification pipeline by running Torch [tor] and CUDA on the Jetson's Maxwell GPU. This case profiles the execution of the entire RNN classification pipeline on a mobile GPU.

2. We profile the GPU speed-ups using Google's Renderscript [ren] on commercial Android phones. In this case, we only compare the matrix operations in the Renderscript BLAS library by running a set of benchmarks on the GPU and the CPU of a Nexus 5 phone.

The results of the latency speed-ups and projected real latency on GPUs are shown in Table 3.3. As we can see, the two cases yield $15 - 30\times$ speed-ups when running on GPUs compared to on CPUs. Plugging this speed-up into the 600 ms baseline latency, where RNNs are used for high-level activity classification on real-time sensor data, we can expect a final latency of 20-40 ms, which is comparable or even better than the latency of CRF and DT + HMM. Once the mainstream DL framework such Tensorflow, Torch, and Caffe fully incorporates CUDA or OpenCL drivers, we expect to see similar numbers even with the model running on real-time data

3.3.3 Leveraging DSPs for Classification

We start by describing the off-loading of the classification phase of inference onto the DSP. Compared to feature extraction, classification computation is more unstructured, possibly leading to a larger saving by the DSP off-load. While feature extraction itself can be computationally intensive in many apps, our initial focus is on the classification. Later we also study and explore the benefits of executing learning algorithms on mobile devices in Section 3.3.4.

3.3.3.1 Classification Algorithm

We have chosen Support Vector Machine (SVM), Gaussian Mixture Model (GMM), and Random Forest (RF) as representative models to explore initially, since they are used by a variety of apps. Our implementation of SVM classification is based on libsvm [CL11]. The GMM classification code in C is based on the Voicebox [VOI] MATLAB toolbox. The RF classification implementation is based on the open source librf [Lee]. We choose SVM and GMM classifications to test the DSP's capability to performance arithmetic computation, while the RF classification is mostly used to test the DSP's ability to execute conditional statements.To exploit the energy and performance implications of the off-load, we execute and profile the classification phase on both the CPU and the DSP, and the evaluation results will be shown in Section 3.3.3.3.

Example Application. We use publicly available datasets to train the classification models for latency and energy profiling. Traditionally SVM and RF are used for general classification problems, and we use the daily activity dataset collected by TU Darmstadt [HFS08], which includes accelerometer features over a 7-day time period, collected from two wearable accelerometer sensors in pocket and wrist. We have trained an SVM model to classify the action scenario of each sample as one of {*dinner, commuting, lunch, work, undefined*}. Each sample is obtained over a 400ms window, and consists of 13 features. We randomly select 5000 samples from the dataset from training and 200 samples for classification.

GMM is mostly used for speech related classification tasks, and we use the speaker

recognition application where a GMM model is trained for each speaker. For each sound clip a likelihood score is calculated which represents the probability of the sample being generated by the GMM. The speaker corresponding to the maximum likelihood is set as the output. We use the TIDIGITS dataset [tid] as the input, and a set of MFCC features on $3s$ sound clip at $8KHz$ is used as one sample. To simplify the problem, we select 5 speakers for the training and testing (there are more than 200 speakers in the original dataset), and each speaker model has 9 mixture models.

Memory Optimization. There are two reasons that drive us to optimize the memory footprint of classification algorithms: (1) Although smartphones today are equipped with powerful processors and large amount of RAM, the DSP itself may only have limited access to RAM, therefore the data and code for classification must be able to fit into the RAM accessible to DSP; (2) In reality, even the DSP can access the shared the memory with the CPU and save all data and code in the shared memory, such access can be expensive. On the one hand the access to shared memory with CPU will add penalty to performance, consuming extra processor cycles. On the other hand, more memory usage will also lead to higher energy consumption [CH10]. Therefore we have optimized the run-time memory footprint of our classification implementations.

We first model the run-time memory footprint of the SVM, GMM, and RF classifiers, only considering the data memory used to store the models:

$$Memory_{SVM} = N_{SVM} \cdot N_{SV_i} \cdot L_{feature} \cdot unitSize \tag{3.1}$$

$$Memory_{GMM} = N_{speaker} \cdot N_{mixture} \cdot L_{feature} \cdot 2 \cdot unitSize \tag{3.2}$$

$$Memory_{RF} = N_{node} \cdot nodeSize = \sum_i N_{tree} \cdot unitsPerNode \cdot unitSize \tag{3.3}$$

The SVM memory model is shown in Equation 2.1. Since an SVM is only able to perform

binary classification, we use the one-to-one vote to solve multi-class problem, one SVM needs to be trained for a two-class classification problem. Therefore a total of $N_{SVM} = n(n-1)/2$ SVMs will be required for a multi-class problem with n classes. For each SVM, a set of *Support Vectors (SV)* have to be stored for the classification, and the size of each SV depends on the number of features $L_{feature}$ and the unit size to store each number $unitSize$. For example, the unit size will be 4 bytes if integer is used to store a number in C.

Similarly, the GMM memory model is shown in Equation 2.2. A GMM model is trained for each speaker, and contains a number of mixture models ($N_{mixture}$). Each mixture model will store the mean and covariance matrices, whose sizes will again depend on the dimension of feature space $L_{feature}$, and the $unitSize$ used.

Different from SVM and GMM memory model, the size of an RF does not directly depend on the number of features and the number of classes. Since RF is a set of trees trained on a randomly selected subset of the training dataset, using a number of randomly selected features, the size of an RF will depend on the total number of trees in the forest, as well as the size of each tree. The memory model of RF is shown in Equation 2.3.

Because all models must be loaded in to the main memory for the execution of the classification, the run-time memory footprint of classification algorithms mostly depend on the size of the classifiers. We first identify that compiler flags such as `-O2, -Os, -static, -fpack-struct` will help reduce the size of the binary code of the classification implementations, without changing the programming logic and classification accuracy.

Furthermore, more substantial memory optimization can be achieved by trading the classification accuracy. One common property of the three memory models is that they all depend on the unit size of each number stored in the model. Therefore we start to reduce the unit size by using single-precision floating numbers or even integers instead of double-precision floating point numbers. Moreover, different storing units will affect the mathematical arithmetics in the algorithms. All of SVM, GMM, and RF classification algorithms use a set of mathematical operations such as multiplication, addition, and exponentiation. As DSPs typically do not have specialized floating-point processing units, transforming most of

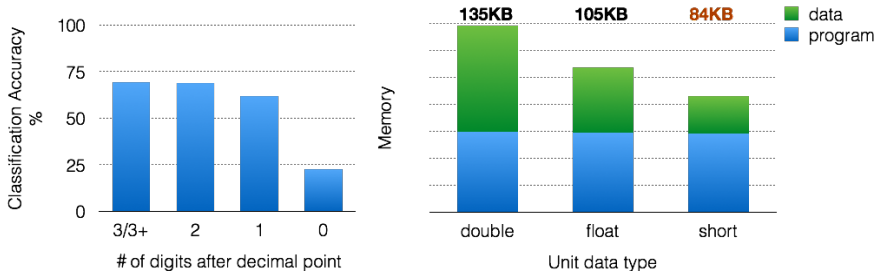


Figure 3.3: Memory optimization for SVM.

the floating-point arithmetic in the original implementations to be fixed-point will not only help reduce the memory footprint, but also improve the performance of the classification on DSP.

Since the accuracy of the classification algorithm should not be sacrificed by this optimization, we conduct an experiment to show the trade-off between memory footprint and the classification accuracy. Figure 3.3 (a) shows the change in classification accuracy[1] as we change the precisions of all floating-point numbers used in the classification algorithm. The accuracy can be largely preserved even if only two digits after decimal point are kept for each floating-point number. Figure 3.3 (b) shows the run-time memory footprint of the same algorithm implemented in different basic unit data types, i.e. *double, float, short int*. By scaling the rounded floating-point numbers to short integers in the SVM classification algorithm, we have reduced 66% of the run-time data memory used by the algorithm.

Similar, Figure 3.4 shows that by replacing double-precision floating point with short integers in GMM classification, the accuracy remains almost the same, but run-time memory footprint is reduced by 44%. As discussed above, eliminating the number of floating-point operations also leads to better performance of the classification algorithm on the DSPs. Even in DSPs that do support floating point, such as those found in the latest generation of Snapdragon processors, the memory constraints remain and moving away from floating-point to the more compact fixed-point representation is beneficial.

[1]Here we are running the same classification model on the same dataset, so the change in accuracy is only caused by the change of data type used in the algorithm.

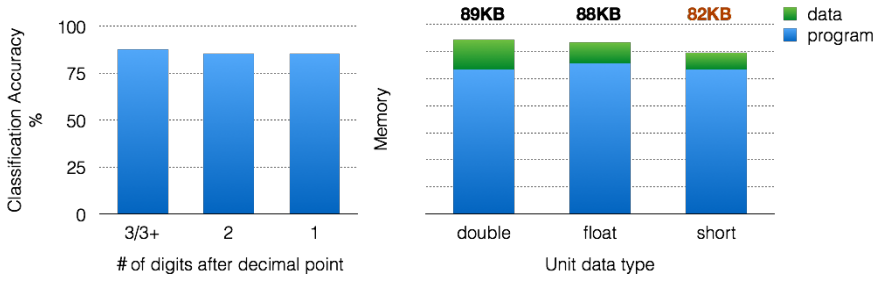


Figure 3.4: Memory optimization for GMM.

We also try to eliminate the dynamic memory allocations in the open-source implementations such as libsvm and librf. For example, our SVM implementation mSVM [msv] uses only $84KB$ of static memory, while the original libsvm uses $238KB$ static memory plus $132KB$ dynamic memory. Similarly, our mRF implementation [mrf] uses $266KB$ of static memory, compared to the librf's $310KB$ static memory and $300KB$ dynamic memory.

Generalization of Memory-Accuracy Trade-off. The idea of using smaller units to save model and the use of fixed-point operations instead of floating point as a result, can generally lead to smaller memory footprint and better performance for all problems. However, it is unclear how to balance the trade-off between memory footprint and classification accuracy. To address this issue, we plan to develop a profiling tool. Given the trained model and training data, the tool will give the possible unit types used to store each number, and the corresponding classification accuracy. The user can specify a maximum tolerable reduction in accuracy, and the tool will select the possible unit type with best memory footprint.

3.3.3.2 Experimental Platform

In order to verify the feasibility of off-loading classification algorithms to DSP and show potential gain in energy efficiency, we implemented the above mentioned classification algorithms on mobile development platforms with open programmable DSPs, including a TI Pandaboard ES [TIb] and a Qualcomm Snapdragon development tablet [QUAb], both running Android JellyBean. The Pandaboard has an OMAP 4460 SoC on board, which includes

a dual-core Cortex-A9 ARM CPU at 1.2GHz and a TI C64x low-power DSP at 500MHz. The Qualcomm tablet is shipped with a Snapdragon 800 SoC, which has a quad-core Krait CPU at 2.3GHz and a Hexagon DSP. Both the TI and the Qualcomm DSPs are running their own real-time OS (RTOS). In order to execute our implementations on the DSP, we use two RPC infrastructures proposed by TI and Qualcomm respectively. These libraries enable RPC calls from the kernel running on the CPU to the RTOS running on the DSP. Since the SVM, GMM, and RF implementations are in C, they can be compiled for native execution both on the CPU and the DSP with the latter being accessed using RPC.

For power measurement, we use two external Agilent 34410A digital multi-meters connected to a Agilent E3631A DC power supply to measure the total energy consumption of the experimental platforms. We also use two on-board resistors to measure the energy consumption of the CPU and DSP subsystems.

3.3.3.3 Evaluation

We profile the SVM, GMM, and RF classification code on CPU or DSP individually, showing the difference in latency and device-level energy consumption. Since we are only off-loading the classification phase of the inference pipeline, the app processor (CPU) still collects sensor data and extracts features, then passes the feature vector to classification implementations on either CPU or DSP.

3.3.3.4 Performance Profile

Figure 3.5 shows the latency profiling result for SVM and GMM classifications respectively, executed on both the TI Pandaboard and the Qualcomm tablet. We define *latency* as the time to classify one sample. Note that the SVM classification implementation profiled here is using fixed-point arithmetics, but the GMM implementation is not yet fully optimized for fixed-point and is still using single precision floating-point arithmetics.

Although running the classification on DSP instead of CPU incurs certain overhead in terms of latency, it is still well within the range that is acceptable from the app perspective.

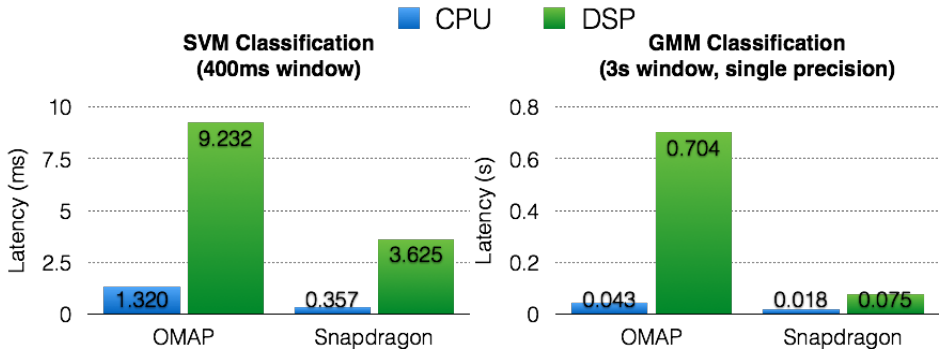


Figure 3.5: Latency profiling result for SVM and GMM.

Since the feature vector of a sample are computed over a sliding window of sensor data, the actual classification is duty-cycled by the window size. The current classification result will be valid until the next complete window. Therefore the latency is acceptable as long as it is smaller than the window size. In the activity recognition app, the user can hardly discern the increase of latency from $1ms$ to $9ms/3ms$ because each sample is calculated from a $400ms$ sliding window. In the GMM app, the window length is $3s$, therefore the execution on DSP is still sufficient because it takes less than $3s$ to recognize one clip. Generally, off-loading classification computation to DSP will lead to only negligible latency overhead.

Energy Profile. We then explore the energy implication brought by off-loading computation to DSP with the intuition that despite the increased latency, the specialized ISA of the DSP will result in overall improvement in energy efficiency. We use SVM classification on the TU daily activity dataset as an example application, since the SVM code is fully optimized for fixed-point execution. Our experiments profile the energy saving of the entire platform due to the off-load in two cases.

In the first case, a fixed number of samples are classified over a fixed amount of time, on the CPU or the DSP of the Qualcomm tablet, with result shown in Figure 3.6 (a). Overall the tablet consumes about 6.4% less energy when the classification execution is on the DSP, and the saving percentage is 70% when the baseline energy consumption of the tablet is excluded.

However, in real-world scenarios the context-aware apps often do not have the luxury of executing only a fixed number of classifications. Instead, the inference has to be exe-

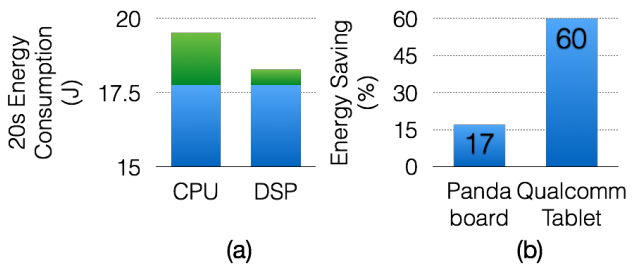


Figure 3.6: Energy profiling of SVM classification.

cuted continually, providing timely context to the user. Therefore we conduct the second experiment, where the classification phase is executed continually on the CPU or the DSP, respectively. We are classifying 300 samples as a group to amortize the overhead. This is common in reality because the inference often operates on a sliding window, and the classification algorithms run on blocks of samples and not on individual samples.

As is shown Figure 3.6 (b), the Pandaboard consumes about 17% less energy when the continuous execution of classification is on the DSP instead of the CPU, and the energy saving for the Qualcomm tablet is about 60%. Both results have proved that DSP off-load for classification computation is a promising solution for energy efficient context inference on mobile phones. Note that the Qualcomm Snapdragon SoC has a more powerful and power-hungry CPU as well as a lower power DSP compared to the TI OMAP4, therefore off-loading computation to the DSP results in a much larger energy saving in the Qualcomm tablet than in the TI Pandaboard.

Note that the 17% and 60% empirical energy savings described above are achieved by off-loading *unoptimized* classification code to the DSP. The code is compiled by standard gcc-based cross-compiler for the CPU and the DSP. Our ongoing work focuses on the optimization of these algorithms using libraries provided by the Qualcomm DSP, which would potentially offer better performance and energy efficiency. Furthermore, we are working on the off-load of feature extraction computations in addition to classifications as well.

Although the performance and energy profiling are conducted on two mobile development boards, the idea of off-loading computation to DSP can be applied to mainstream mobile phones in the commercial market as well, as shown by Tabel 3.2.

3.3.4 Learning on DSPs

We have so far only considered off-loading the classification phase to DSP. Learning, on the contrary, has traditionally been off-loaded to the cloud side for execution due to its high complexity. Nevertheless, there are several reasons that prompt us to study the feasibility and benefits of executing classifier learning on the smartphone, either on the main app processor or low-power processors such as the DSP:

- Performing learning on the mobile device enables timely updates of machine learning models. As the app collects new data and ground truth labels, model can be updated on-the-fly. This will also enable timely feedback to the user.
- It also reduces network communications. Previously all new samples and labels must be uploaded to the cloud for updates of the model. With learning capability on the phone, such communication can be eliminated, which will possibly lead to energy saving and longer battery life.
- It enables the mobile device to learn or reinforce the model even without network connectivity.
- The power density of mobile SoCs has been constantly increasing in recent years and hence we come across the utilization wall or dark silicon effect [EBS11, OH05]. Modern smartphones area shipped with powerful processors, and performing learning on the phone help increase the utilization of mobile SoC, while reducing the dark silicon effect. It has been proved by previous work [RLC12] that mobile SoC can be fully utilized to perform intense parallel tasks, such as learning, for sub-second bursts.

However, even with the above benefits, the resource constraints on smartphones and mobile devices, especially limited memory, remains and will make the execution of classifier learning challenging. In this section, we describe the incorporation of on-line learning algorithms to help reduce the complexity and run-time memory footprint of learning. Specifically, we use Stochastic Gradient Descent (SGD)-class algorithm to train an SVM. Our evaluation

results have shown that SGD-based solver can significantly reduce the running time and memory footprint of learning compared to traditional SVM solvers.

3.3.4.1 Background: SVM

Recall the task to train a linear SVM, given a set of training data $\vec{x}_i$ and labels y_i. The goal is to find a hyperplane denoted by its normal vector $\vec{w}$, which can separate the training data, so that $\vec{x}_i \cdot \vec{w} - b \geq 1$ for all positive samples and $\vec{x}_i \cdot \vec{w} - b \leq -1$ for all negative samples. Specifically, the training is trying to minimzing $||\vec{w}||$ with subject to $y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1$ for all $\vec{x}_i,\ 1 \leq i \leq n$. This can be formalized as solving the following objective function:

$$\arg\min_{\vec{w},b}\{\frac{1}{2}\lambda||\vec{w}||^2 + C\sum_{i=1}^{n} L(1 - y_i(\vec{w} \cdot \vec{x}_i - b))\} \tag{3.4}$$

where the first term is the $L2$-regularization term and the second one is the $L1$-loss term using hinge loss function.

Popular implementation of SVM solver, such as libsvm [CL11], uses Sequential Minimal Optimization (SMO) to solve the above objective function. We omit the details of the SMO algorithm since it is not the focus of this work. SMO-class algorithm is a general solution for SVM and supports all types of kernel functions. However, it also suffers from high complexity with poor performance on hign-dimensional data and big datasets. In addition, the algorithm must load all training samples into the main memory, making it infeasible for execution on devices with limited RAM. In terms of model size, the SVM model trained by libsvm saves all support vectors used in the training, therefore the model can be huge for big datasets.

To improve the performance on large datasets, a variant of libsvm, liblinear [FCH08], aims at reducing the training complexity. liblinear uses Dual Coordinate Descent (DCD) [HCL08] instead of SMO. Although DCD only supports linear SVM, it greatly improves the training performance on large dataset. In addition, it reduces the model size of trained SVM by storing only the w vector, i.e. linear combination of all support vectors, instead of all the support vectors. However, liblinear is still not an on-line algorithm and it has to load all

samples to the main memory for training. Therefore it does not solve the memory footprint issues for execution on mobile devices.

3.3.4.2 Stochastic Gradient Descent based SVM

In this section we describe the use of on-line learning algorithm, Stochastic Gradient Descent (SGD), to reduce the complexity and memory footprint of linear SVM training.

3.3.4.3 Stochastic Gradient Descent

To minimize the above object function iteratively, a class of gradient descent (GD) [RHW85] based algorithms is often used. Generally, GD computes the gradient of the object function based on the entire training set, and update the weight vector $\vec{w}$ accordingly. Instead of using the full training set, another class of algorithms, Stochastic Gradient Descent (SGD) [Bot10, Bot12], uses a single randomly picked sample in each iteration to estimate the gradient of the objective function.

We describe the general steps to solve an linear SVM using SGD here. We rewrite the object function as:

$$Obj(\vec{w}) = \frac{1}{2}\lambda||\vec{w}||^2 + \frac{1}{n}\sum_{i=1}^{n} L(\vec{w}, \vec{x}_i, y_i) \tag{3.5}$$

where L is the hinge loss function:

$$L(\vec{w}, \vec{x}_i, y_i) = \max(0, 1 - y_i\vec{w}_i \cdot \vec{x}_i) \tag{3.6}$$

In each iteration i, the update of w involves the following steps:

1. A sample $(\vec{x}_i, y_i)$ arrives
2. Compute $y'_i = \vec{w}_i \cdot \vec{x}$ based on the current $\vec{w}_i$
3. Compute the value of loss function $L(\vec{w}_i, \vec{x}_i, y_i)$

4. Update weights $\vec{w}_i$ according to:

$$\vec{w}_{i+1} = (1-\lambda)\vec{w}_i - \eta \frac{\delta L}{\delta \vec{w}_i} L(\vec{w}_i, \vec{x}_i, y_i)$$

where η is the learning rate being adjusted at each step.

The SGD algorithm is originally designed for learning over very large dataset, because the update in each iteration only accesses one randomly picked sample instead of all samples, which significantly reduces the complexity and memory footprint of the algorithm. If one assumes that the input data are randomly drawn from the ground truth distribution, each new incoming sample can be used as the random picked sample, effectively enabling on-line learning on the streaming input data. However, the SGD solver may note converge to optimal solution in one pass. It often requires several passes on the same training set to achieve the accuracy of SMO/DCD solver. As we will discuss in 3.3.4.4, the performance and memory advantage over SMO/DCD still hold even if SGD has to run multiple passes.

Implementation. Because SGD eliminates the need to store the entire training set in the memory, and the model can be updated on-the-fly using one sample at each time, it is a natural fit for execution under resource constraints on mobile devices. Given this observation, we apply SGD to develop SGD-SVM, an on-line solver for linear SVM capable of executing linear SVM learning on the CPU of mobile devices or even low-power co-processor such as the DSP. Our implementation is based on existing open-source code [Bot], and can execute on the main app processor (ARM CPU) as well as the Qualcomm Hexagon DSP. The implementation is iterative so that it can work on stream of input data. We also added file I/O module to support libsvm format input file. The implementation only supports binary classification right now.

3.3.4.4 Evaluation

We compare the performance, accuracy, and memory footprint of executing SGD-based SVM learning on the Hexagon DSP, against libsvm (SMO) and liblinear (DCD). We use the

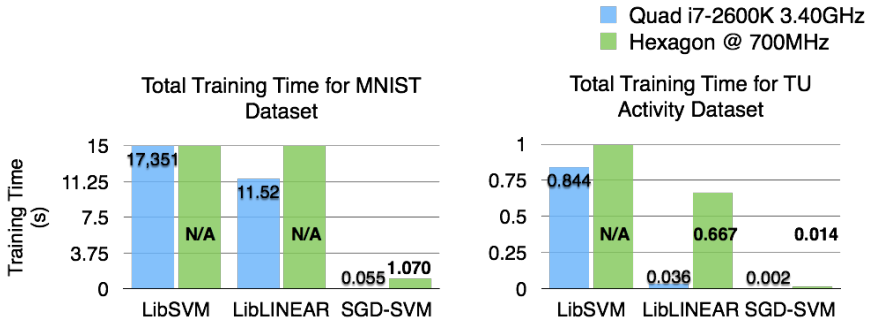


Figure 3.7: Comparison of learning time using different SVM solvers.

Hexagon DSP simulator to profile and benchmark our implementation, and the profiling of same implementation on a desktop machine with quad-core i7-2600K @ 3.4GHz is used as a baseline. We also profile unmodified implementations of libsvm and liblinear as comparisons.

Dataset. Two datasets are incorporated for the benchmark. The small dataset is the TU-darmstadt daily activity dataset described in Section 3.3.3.1. To test the feasibility of learning over larger and more-complex dataset on the DSP, we also use the MNIST handwritten digits dataset [Yan]. Each sample in the MNIST dataset has 781 vision-based features, and should be classified as one of the handwritten digits. We train the linear SVM using 60000 samples and test on 10000 samples. Both the activity classification and digit recognition are converted into binary classification problems.

Performance. Figure 3.7 reports the learning (training) time using different SVM solvers. The execution time on DSP is measured on the Hexagon simulator (700MHz). As shown in the results, it is not even feasible to execute unoptimized libsvm and liblinear on the DSP to learn over the large MNIST dataset due to the memory constraints. From the results on x86 i7 CPU processors, SGD-SVM is about 300000× faster than libsvm and 200× faster than liblinear. For the smaller activity dataset, SGD-SVM is about 47× faster than liblinear on the DSP, and 420× faster than libsvm & 330× faster than liblinear on the x86 i7 CPU. Given the great performance improvements, even if SGD-SVM requires multiple passes on the same training set, it still takes less time to train overall.

In addition, the time used by the DSP to update the model in each iteration is very

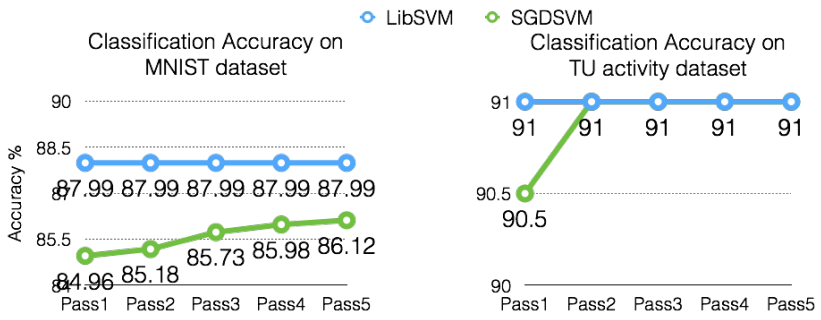


Figure 3.8: Comparison of accuracy of libsvm and SGD-SVM.

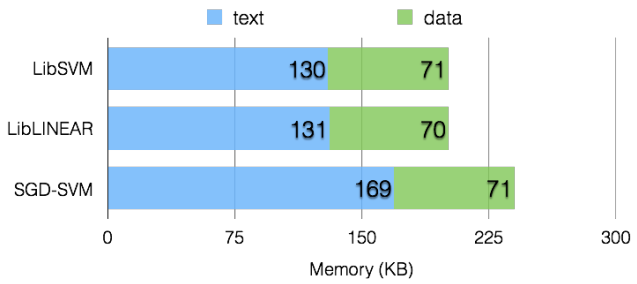


Figure 3.9: Comparison of static memory usage.

small. It takes the DSP about $1/60000s$ in the larger dataset, and $0.01/5000s$ in the smaller dataset, in each incremental update step. Note that we omit the I/O time during which the DSP simulator read data form file, since we anticipate the DSP will have direct access to audio or sensor data in a real setting.

Accuracy. Recall that the SGD algorithm may not converge to optimal solution after one pass on the training dataset. Figure 3.8 compares the model accuracy learned by SGD-SVM after different number of passes to the model accuracy trained by libsvm, which is used as optimal accuracy here. After the first pass, the SGD-SVM model accuracy is already near optimal. In addition, the model accuracy is very close or even identical to optimal accuracy after a few passes. The result here proves that using SGD-SVM instead of libsvm will only degrade the model accuracy slightly. The performance improvement of SGD-SVM will remain the same even it executes for several passes to achieve near-optimal accuracy.

Memory. We discuss the memory implication of SGD-SVM compared to libsvm and liblinear in terms of dynamic memory usage, static memory usage, and model size:

- **Dynamic memory.** Both libsvm and liblinear must store all samples in memory prior to the start of the learning. For the larger MNIST dataset with 60000 training samples and 781 features per sample, merely storing each data unit requires about $89MB$ of run-time dynamic memory even short integer is used. Obviously such great amount of run-time memory cannot be allocated on the DSP, and even on the CPU can be very expensive. For the smaller activity dataset, the required memory to store all training samples is $137KB$. On the contrary, SGD-SVM in on-line mode only needs to store one sample at each time, which is only about $1.5KB$ for the large data set ad 26 bytes for the small dataset. The great save in run-time memory footprint again justifies the benefits of using SGD based on-line learning algorithms.

- **Static memory.** Since the binary file has to be loaded into memory for execution, static memory usage also matters, including compile-time code memory and data memory. Figure 3.9 shows the comparison of binary file size using different SVM solver implementations. As of now SGD-SVM uses a bit more static memory (about $40KB$) than libsvm and libnear because it is not yet fully-optimized in memory footprint. However, the significant reduction in dynamic memory usage by SGD-SVM will easily offset its additional static memory usage.

- **Model size.** The SVM model has to be saved in the main memory for the classification later. libsvm stores all support vectors and therefore the model size depends on both the size of training set and the dimension of the feature space. Using libsvm, the model learned over the larger MNIST dataset will be about $29MB$, and the one learned over the smaller dataset will be about $86KB$. Instead of storing all support vectors, liblinear and SGD-SVM only stores the $\vec{w}$ vector, and the model size only depends on the dimension of the feature space. The model learned will be only $1.5KB$ for the larger dataset, and 26 bytes for the smaller one.

In general, SGD-SVM significantly reduces the memory footprint of SVM learning compared to libsvm and liblinear. Machine learning researchers have proposed other approach such as Truncated Descent [LLZ09] to further decrease the model size learned from SGD

based on-line learning.

3.3.4.5 Discussion

As currently the SGD-SVM takes stream of input samples and use each new incoming sample to estimate the gradient, the randomness of the picked sample is limited by the temporal correlation of samples generated by analog front end and ADC in the sensors. To address this issue, the SGD-SVM can conduct sub-sampling on the input stream and randomly drop some of the new samples. Alternatively, the SGD-SVM can buffer a limited amount of historical data, and refine the model periodically. Buffering and learning in batch will increase the randomness of the training data, therefore yield better model accuracy. For example, the learning algorithm may collect and buffer one hour of data, shuffle the dataset, and update the model every hour. In both cases, the temporal correlation among samples can be removed.

Given that SGD-SVM cannot converge to optimal model accuracy after one pass, we propose two use cases of SGD-SVM on the DSP or app processor of mobile devices:

1. **On-line learning.** The learning algorithm can buffer no historical data at all and operate in pure "on-line" mode. In this case SGD-SVM can achieve pretty good accuracy by learning over the training set in one pass. Alternatively, SGD-SVM can operate in batch mode, as described above.

2. **Accelerator.** One can also image the use of SGD-SVM as an accelerator. Such an accelerator can run continuously on the DSP to update the model to reduce the workloads on CPU, while the entire training dataset is stored in the memory accessible to the app processor (CPU). Since all training data are available, SGD-SVM can run multiple passes on the training dataset to achieve near-optimal accuracy. In this mode, the total number of memory access to all samples in SGD-SVM is still smaller compared to libsvm, even if SGD-SVM requires multiple passes. In addition, the use of SGD-SVM will reduce the complexity of learning and accelerate the computation.

It is also an interesting topic to study the memorization and generalization of SGD-trained model. For example, test if SGD-SVM can generalize to new class labels that it has not seen before (or it has only seen a limited number of samples in this class).

3.4 Wearable-Mobile Coordination

The sensing, computation, and communication capabilities of modern smartwatches make them ideal candidate to improve the inference accuracy and energy efficiency of always-on context inference. Although in the MiLift app we have considered sensors from only a smartwatch, combining sensors for the watch and the phone when both devices are available can possibly provide further optimizations opportunities. We are also motivated by the intuition that the sensing and inference computation can be alternated between the watch and the phone for maximum sensor coverage.

In this section, we investigate whether watch-phone coordinations can be leveraged to tackle the two major drawbacks associated with always-on context inference apps by studying the following key research questions:

Q1*: Can we leverage smartwatches to increase sensor coverage and inference accuracy?* We examine whether inferences running on the watch alone can provide sufficient accuracy with reasonable energy consumption (i.e. the watch can last for a full day). The execution can then be off-loaded to the watch whenever the phone is unavailable for better coverage and accuracy.

Q2*: Can smartwatches help reduce energy consumption of context inferences?* By performing inferences on the watch and replacing power-hungry phone sensors with low-power watch sensors, the energy resource of the phone can be preserved. However, it is important to balance the power and energy trade-off between the watch and the phone due to the reduced battery capacity of the watch.

Part	Apple Watch	Moto 360 Watch	Nexus 5 Phone
CPU	520MHz S1	1GHz OMAP3	2.3GHz Krait 400
RAM	512MB	512MB	2GB
Storage	8GB	4GB	16/32GB
Radio	BLE/Wi-Fi/NFC	BLE/Wi-Fi	BLE/Wi-Fi/NFC
Battery	3.8V 205mAh	3.8V 320mAh	4.3V 2300mAh
Weight	25g	49g	130g
Sensors	Accelerometer, gyroscope, pedometer, heart rate, microphone	Accelerometer, gyroscope, pedometer, heart rate, microphone, light	Accelerometer, gyroscope, pedometer, microphone, compass, proximity, light, GPS

Table 3.4: Comparison of device hardware platforms. (Showing specifications for Apple Watch 38mm Sport.)

3.4.1 Background: Smartwatch Basic Profiles

To answer the two key research questions raised above, we first compare the hardware platforms and basic energy profiles of several state-of-the-art commercial devices: the Apple Watch, the Moto 360 smartwatch and the LG Nexus 5 phone.

3.4.1.1 Hardware specifications

Table 3.4 compares the hardware specifications of popular smartwatches and smartphones. Smartwatches today have rather powerful CPUs, RAM, and radios considering their tiny and lightweight nature, enabling them to execute standalone apps without any assistance from phones. Both the Apple Watch and the Moto 360 watch provide a rich set of radios and sensors similar to smartphones today. They are also shipped with optical-based heart rate sensors which complement the phone's inability to obtain accurate human heart rate readings. However, the battery capacity of smartwatches is much smaller than smartphones because of the device volume and weight limits. Due to the similarity between the Apple Watch and the Moto 360, we have chosen the latter one as an example to benchmark our app scenarios.

Device	Power (W)	Current (mA)	Lifetime (h)
Moto 360 screen off	0.013	3.283	97.472
Moto 360 screen on	0.550	142.520	2.245
Nexus 5 screen off	0.254	58.913	37.343
Nexus 5 screen on	1.853	435.260	5.057

Table 3.5: Comparison of device power profiles.

3.4.1.2 Basic power profiles

To further analyze the feasibility of running context inferences on smartwatches, we created basic power profiles of a Moto 360 watch and a Nexus 5 phone (Table 3.5). We measured the power consumption of the watch and the phone using the experimental setup described in Section 3.4.4.1, with the two devices connected via BLE. We profiled the two devices for screen off (sleeping) and screen on, respectively, because their normal power consumptions are typically between these two extreme cases. Compared with the Nexus 5 phone, the Moto 360 watch has a lower power consumption in general. However, the power difference between sleeping and screen on is much more significant on the watch than on the phone. While the low-power nature of the smartwatch suggests the possibility for execution of context inferences, the much smaller battery capacity of the watch requires detailed studies of the power and energy trade-offs so that inference executions on the watch will not significantly reduce its battery life.

3.4.1.3 App development

Both iOS and Android have provided watch app development frameworks. Starting from the Apple watchOS 2, the watch will be able to host native apps with the WatchKit framework [wat]. Similarly, the Android SDK offers a set of Wear APIs [anda] that support watch app development and manage the communication between phone and watch. On both platforms, the programming paradigm of watch apps remains the same as mobile apps, and the SDKs support shared libraries between watch apps and phone apps.

Goal	Approach	Results
#1	Compare the inference accuracy using different watch/phone sensor combinations.	Section 3.4.4.3
#2	Create accuracy and power profiles of standalone inference executions on the watch.	Section 3.4.4.4
#3	Increase sensor coverage and inference accuracy by executing inferences on the watch when the phone is not available.	+37.4% accuracy (Section 3.4.4.5)
#4	Partition the computation across both watch/phone for better energy efficiency.	−67.3% energy (Section 3.4.4.6)
#5	Replace power-hungry phone sensors with low-power watch sensors.	−61.0% energy (Section 3.4.4.7)

Table 3.6: Summary of design goals.

3.4.2 Design Goal

We discuss how we can leverage the watch-phone coordination to improve inference accuracy and to reduce energy consumption of always-on context inferences. Table 3.6 summarizes the design goals and corresponding evaluations.

3.4.2.1 Example App Scenarios

To showcase the benefit of watch-phone collaborations we have selected two example app scenarios for benchmarks:

App 1: High-level Activity Recognition (AR). We use the MiLift's high-level activity recognition introduced in Section 2.3 as the first example app. To study the accuracy and energy implications of watch-phone collaboration, we consider accelerometers on both the watch and the phone. We compose the inference by choosing several different sensor combinations (SC):

- SC1: Watch accelerometer only.
- SC2: Phone accelerometer only.
- SC3: Watch accelerometer and phone accelerometer.
- SC4: Phone accelerometer and phone GPS.

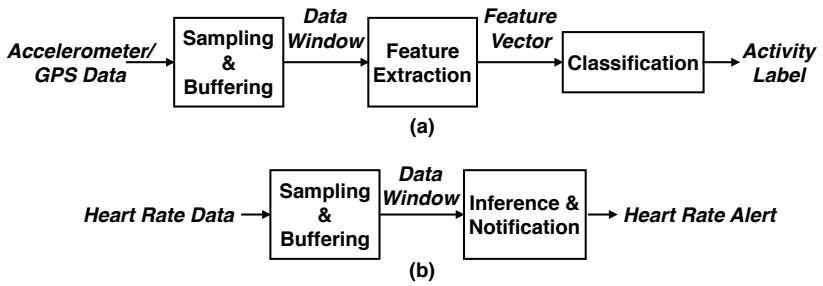


Figure 3.10: Inference pipeline of (a) the AR app, and (b) the HR Monitor app.

- SC5: Watch accelerometer, phone accelerometer, and phone GPS.

There are several reasons for selecting the above sensor combinations. First, the comparison of SC1, SC2, and SC4 shows the accuracy and energy implication of running inferences on the watch (Goal #1, #2) and demonstrates whether the watch can be used to increase the device sensor coverage (Goal #3). Second, with SC5 we study the optimal inference partition between the watch and the phone to balance the cost of computation and communication (Goal #4). Finally, the study of SC3 and SC4 suggests the possibilities of replacing high-power GPS with low-power watch accelerometers (Goal #5).

App 2: IFTTT Rules (HR Monitor) We use the IFTTT Rules app described in Section 4.2 as the second example app. In this work, we consider a specific rule in the personal health space: a heart rate monitoring app (HR Monitor). This app monitors heart rate of a user during workout sessions and generates a notification whenever the heart rate reading exceeds a certain threshold. There are two reasons that the HR monitor can benefit from executions on smartwatches: (1) most smartwatches today are shipped with optical-based heart rate sensors enabling them to accurately capture human heart rate readings; (2) because users are more likely to wear a watch than to carry a phone during a workout session, generating notifications on the watch can give better in-time feedbacks to users.

The inference pipelines of the two example apps are shown in Figure 3.10.

3.4.2.2 Prerequisite Goals

Because of their distinct locations on human bodies, sensors on smartwatches and smartphones can capture different information about user contexts and behaviors, and the accuracy of context inferences will depend on the choice of sensors. We research the effect of various watch/phone sensor combinations on inference accuracy (Goal #1). Moreover, we study whether inference executions on the watch can provide sufficient inference accuracy while not consuming excessive battery power (Goal #2).

3.4.2.3 Increasing Inference Accuracy

The research question ***Q1*** seeks to increase the sensor coverage and inference accuracy using smartwatches. The hardware comparison in Section 3.4.1.1 suggests that smartwatches today have powerful computation resources and a rich set of sensors enabling watch apps to draw inferences from the sensors. In addition, smartwatches are less intrusive wearable devices than smartphones. Unlike the smartphone usage pattern where the user can sometimes leave the phone away from human body, smartwatch users are more likely to keep wearing the watch throughout the entire day except for very specific periods such as during sleep. While phone placements can greatly affect sensor readings i.e. a phone cannot capture meaningful human movements when not carried by a user, watch sensors can provide better sensor coverage by remaining on the human wrist for longer time. If the Goal #1 and #2 are satisfied, the watch can help increase the sensor coverage and inference accuracy by continuing the always-on execution even when the phone is away from the user (Goal #3). Such situations can be identified when movements are detected by the watch but not by the phone.

3.4.2.4 Reducing Energy Consumption

The research question ***Q2*** asks how we can reduce the energy consumption of context inference executions with the help of smartwatches. Our main goal is to extend the battery life

of the phone without consuming excessive energy on the watch. While the low-power nature of the watch makes it an ideal target to perform inferences, the smaller battery capacity of smartwatches calls for a careful investigation of the watch-phone energy trade-offs.

In this work, we discuss two energy optimizations: First, context inference apps typically consist of several modules, such as sampling, feature computation, and classification. The partition of these modules across devices can not only affect the amount of computation on the watch and the phone but also change the amount of data transmitted between devices. It remains unclear how we can balance the energy trade-off of inter-device transmissions and local computation on the watch. Therefore we study the optimal partitioning of inferences for the best energy efficiency (Goal #4). While most watch apps today are merely a UI for the corresponding phone app, we have quantified the benefit of executing certain computations on the watch. Second, given that sensors on smartwatches can provide additional information about user behaviors and contexts, the inference apps can minimize its energy consumption by eliminating the use of high power sensors such as GPS (Goal #5). Moreover, spreading the execution across both the watch and the phone enables inferences to be continued on the other device if one device runs out of battery power.

3.4.3 Implementation

For both the AR app and the HR Monitor app, we have implemented a watch app on a Moto 360 smartwatch and a phone app on a LG Nexus 5 smartphone. The apps were developed under Android 5.1.1 SDK (API 22). We used the Android Wear DataMessage API for BLE data communications between devices. The watch app or phone app can execute standalone in off-line mode generating activity labels using only watch sensors or phone sensors, respectively. The two apps are also able to coordinately compose the inferences using both watch and phone sensors.

Figure 3.10 shows the inference pipeline of both apps. For the AR app, the pipeline includes sampling and buffering, feature extraction, and classification. The AR app reads accelerometer on the watch and both accelerometer and GPS on the phone, and generates

activity label every second. The HR Monitor inference pipeline includes two modules: the sampling and buffering module that captures heart rate, and the inference and notification module that detects excessive heart rate and alerts the user about possible risks. Both inference pipelines can be partitioned across the watch and the phone. The computation on each device and the amount of BLE data transmitted will change based on different partitions. Section 3.4.4.6 discusses the optimal module partition.

Note that both app implementations offer two execution modes: *always-on* and *periodic*. In always-on mode, the app continuously reads sensor data in the background and generates activity labels for notification and/or logging. In periodic mode, the user can configure the inference to execute for a certain period of time $t_{inference}$ at a given time interval $t_{interval}$. One example could be *execute activity recognition for* 10 *minutes every hour*. The implementation uses the Android AlarmManager to achieve periodical inference executions which will be leveraged to configure and optimize the duty-cycle of inferences in Section 3.4.4.4.

3.4.4 Evaluation

We evaluate the accuracy and energy implications of watch-phone collaboration for context inferences using a Moto 360 watch. The evaluations are summarized in Table 3.6.

3.4.4.1 Experimental Setup

To profile the power consumption of the watch, we tore down a Moto 360 smartwatch and placed an Adafruit INA219 High Side DC Current Sensor Breakout[2] between the battery and the main board. Using an Arduino to read the current and voltage measurements from the breakout, we then calculated the power consumption of the watch. For the Nexus 5 phone, however, we cannot have a similar setup because the phone battery prevents it from booting when extra resistance is detected. Instead, we calculated the phone power by taking current and voltage readings available in the Android file system generated by the internal power

[2] `https://www.adafruit.com/products/904`

Time	Event
8:00AM	Wake up. Turn on watch/phone. Start AR app.
8:00 - 8:30AM	Shower and breakfast.
8:30 - 9:00AM	Drive from home to work.
9:00AM - 1:00PM	Sit at office, walk around sometimes.
1:00 - 1:30PM	Lunch.
1:30 - 5:30PM	Sit at office, walk around sometimes.
5:30 - 5:45PM	Drive from work to gym.
5:45 - 7:15PM	Workout at gym. Use HR Monitor app.
7:15 - 7:30PM	Drive from gym to home.
7:30 - 9:00PM	Dinner at home.
9:00 - 10:00PM	Entertainment (e.g. watch TV/movie).
10:00 - 11:30PM	Reading.
11:30PM - 12:00AM	Shower and prepare to sleep.
12:00AM	Stop AR app. Charge watch/phone. Go to sleep.

Table 3.7: An example daily routine of a user.

gauge[3]. For both the watch and the phone we use a linear discharging model to estimate the lifetime of the battery due to the small current draw. In terms of inference accuracy, we used the scikit-learn implementation to compare the performance of classification models used in the AR app.

3.4.4.2 App Scenario Revisited

Section 3.4.2.1 describes the usage scenario of the two inference apps proposed in this work. To consider the realistic usage of the two apps in a user's daily life, we created an example daily routine of a user, including the usage of AR and HR Monitor app, shown in Table 3.7. In this daily routine, the user wears the watch in the day (16 hours) and charges it while sleeping (8 hours). The smartphone is used throughout the entire day and is normally charged at night as well. The user typically starts the AR app, both on the watch and on the phone, after he or she wakes up in the morning. The AR app will keep running until the user goes to bed at night. The user only uses the HR Monitor app during workout everyday. We made several key observations from the above daily routine:

- The watch must last for at least 16 hours so that the user does not have to charge it

[3]On an LG Nexus 5 phone, the power measurements are available in `/sys/class/power_supply/battery/`.

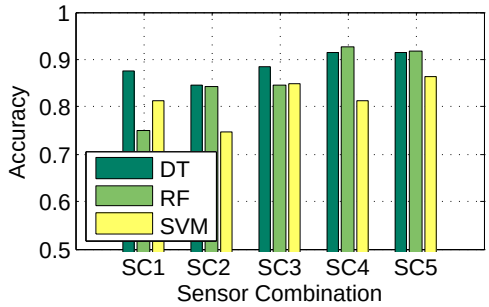


Figure 3.11: Accuracy comparison of three classification models for AR.

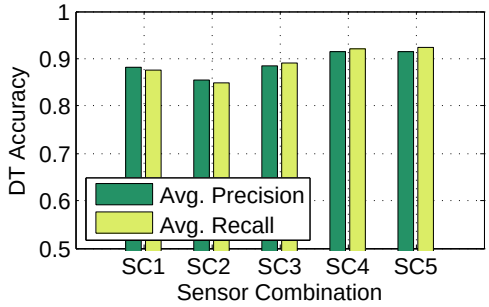


Figure 3.12: Precision and recall scores of the decision tree model for AR.

during the day.

- Since the user frequently uses the phone throughout the day, battery energy of the phone should be conserved as much as possible.
- The AR app is used for 16 hours every day.
- The HR Monitor app is used for 1.5 hours every day.

Although the daily routine shown here is only an example, we stress that a personalized routine for each user can be automatically learned by inference apps. For instance, the Optimized app [opta] discovers the daily schedule of the user during the initial usage period, and accepts user edits and tagging to improve the schedule. Inference apps can then adjust their executions based on the learned user schedule.

Scenario	Power (W)	Current (mA)	Est. Lifetime (h)
Sleep, screen off	0.013	3.283	97.472
Idle, screen on	0.550	142.520	2.245
AR app	0.213	55.929	5.721 (always-on)
HR Monitor app	0.112	28.998	11.035 (always-on)

Table 3.8: Power and battery life of a Moto 360 watch with and without inference apps running.

3.4.4.3 Micro-benchmark: Model Accuracy of the AR App

To achieve Goal #1, we compare the inference accuracy of classification models used in the AR app using different watch/phone sensor combinations described in Section 3.4.2.1. Figure 3.11 shows the accuracy comparison of models achieved by a 10-Fold cross validation using our collected activity dataset. All reported accuracy is calculated using the model trained with best parameters in scikit-learn, and we create a different model for each sensor combination. The results suggest that the accuracy of DT is mostly better than those of RF and SVM. Considering the simplicity of the decision tree compared with the other two models, we have chosen DT for the implementation on the phone and the watch. To ensure that the trained decision tree model is not biased towards any of the three classes, Figure 3.12 plots the average precision and recall scores from 10-Fold CV of DT. The precision and recall scores are consistent with the accuracy number, therefore overall accuracy will be used as the performance metrics for classification models. The model accuracy of different sensor combinations will be used to calculate the real inference accuracy proposed in Section 3.4.4.5.

3.4.4.4 Micro-benchmark: Standalone Executions on Watches

As described in Goal #2, we must ensure that the standalone inference executions on the watch will yield sufficient accuracy while at the same time not significantly reducing the battery life of the watch.

Inference Accuracy. We compare the inference accuracy of AR using only the watch with other solutions. For the AR app, according to Figure 3.11, the DT accuracy using SC1

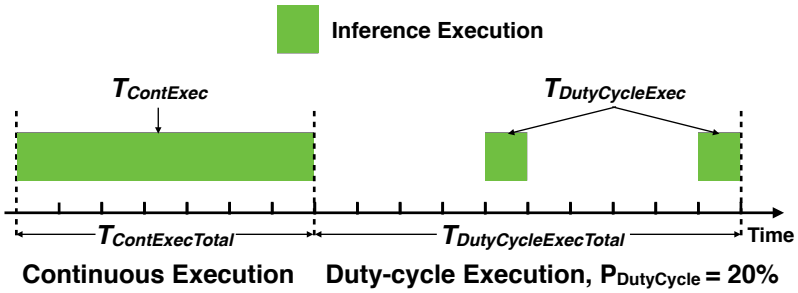


Figure 3.13: Illustration of true execution time.

- only watch accelerometer (87.50%) is better than using SC2 - only phone accelerometer (84.90%), but worse than using SC4 - phone accelerometer and GPS (91.36%). However, these accuracy metrics do not consider the fact that the phone may be away from human body. As discussed in Section 3.4.4.5, the real inference accuracy of AR using SC1 is much higher than using SC2 or SC4. For the HR Monitor app, because both the watch and the phone executions must use the same heart rate sensor on the watch, the inference accuracy will not change regardless of where it runs.

Energy Consumption and Duty-cycle Optimization. Table 3.8 shows the comparison of watch power and estimated battery life with and without always-on inferences running. All power and current numbers shown are from measurements of the entire watch. In this experiment, the AR app executes continuously in the background on the watch sampling the accelerometer at $50Hz$ and generates one activity label every second. The HR Monitor app samples the heart rate sensor at $1Hz$. The battery life of the watch significantly reduces with the inference running always-on, to about 6 hours and 11 hours respectively. Both lifetimes fall under 16 hours and will require the user to charge the watch during the day.

However, inferences do not have to be always-on throughout the day. Instead, they can be duty-cycled based on the daily routine of the user. For example, during normal working hours, the user is mostly sitting at the desk, therefore the AR app can periodically check (e.g. run for $1s$ every $5s$) and start continuous execution only when significant movements are detected. In addition, the HR Monitor app only needs to be turned on whenever the user starts working out each day, and can be switched off after exercise finishes. The *true execution*

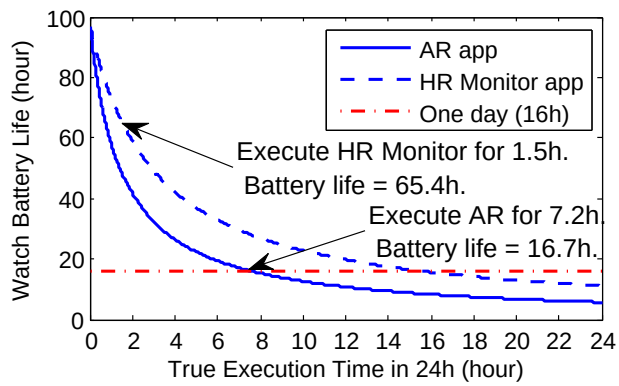


Figure 3.14: Change of watch battery life with different true execution times.

time $T_{TrueExec}$ of an app can be divided into two parts: the continuous execution part $T_{ContExec}$ and the duty-cycle execution part $T_{DutyCycleExec}$, as demonstrated in Figure 3.13. It can be modeled as follows:

$$\begin{aligned} T_{TrueExec} &= T_{ContExec} + T_{DutyCycleExec} \\ &= T_{ContExecTotal} \\ &+ P_{DutyCycle} * T_{DutyCycleExecTotal} \end{aligned} \tag{3.7}$$

Where $T_{ContExecTotal}$ is the duration of continuous execution, $T_{DutyCycleExecTotal}$ is the duration of duty-cycled execution, and $P_{DutyCycle}$ is the percentage of the duty-cycle. Based on the schedule in Table 3.7, for the AR app the execution can be duty-cycled when the user is sitting during work ($8h$), dinner ($1.5h$), and reading ($1.5h$), at a certain percentage (e.g. 20% or execute for $1s$ every $5s$), giving a total true execution time of $7.2h$. For the HR Monitor app, the execution is always-on only during the workout ($1.5h$), and remains off otherwise, resulting in a true execution time of $1.5h$. The duty-cycle schedule can be changed based on the personalized daily routine of the user and the inference history.

Figure 3.14 plots the change in battery life with different true execution times of the AR and HR Monitor app. Using the duty-cycle optimization, the AR app and the HR Monitor app execute for $7.2h$ and $1.5h$ each day, respectively. The corresponding watch battery life is $16.7h$ and $65.4h$. Both the numbers are above $16h$, and therefore the reduction in battery life caused by the inference executions is insignificant because it will not require the user to

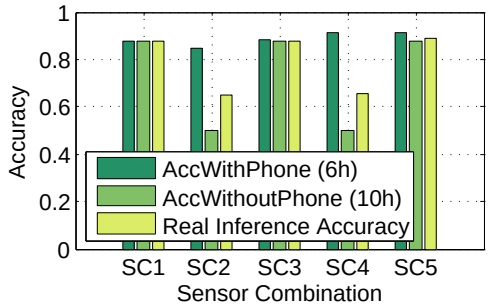


Figure 3.15: Real inference accuracy of AR with an example daily routine of a user (Table 3.7)

charge the watch during the day. In reality inference executions can be alternated across the phone and the watch, and the energy consumption of the watch can be further reduced.

3.4.4.5 Improving Accuracy: Increasing Sensor Coverage

For Goal #3, we quantify the accuracy improvement as a result of increased sensor coverage from watch-phone coordination.

We define *real inference accuracy* as a weighted average of inference accuracy numbers considering the location of the device, shown as follows:

$$\begin{aligned} Acc_{Real} = \frac{1}{T_{Total}}(&Acc_{WithPhone} * T_{WithPhone} \\ &+ Acc_{WithoutPhone} * T_{WithoutPhone}) \end{aligned} \tag{3.8}$$

If the inference samples sensor data from the watch (e.g. SC1, SC3, and SC5), $Acc_{WithPhone}$ will be the inference accuracy using sensors from *both the phone and the watch*, and $Acc_{WithoutPhone}$ will be the accuracy using *only watch sensors*. If the inference takes only phone sensor data (e.g., SC2 and SC4), $Acc_{WithPhone}$ will be the accuracy using *only phone sensors*. However, when the phone is placed elsewhere ($Acc_{WithoutPhone}$), the inference cannot capture any meaningful inertial data. Given that there are other techniques to infer the user activity, such as using GPS and time of day as heuristics, we set $Acc_{WithoutPhone}$ in these cases to be 50%. Finally, T_{Total} is the total time of the day except sleep ($16h$).

According to our example daily schedule in Table 3.7, assume the user only carries the

phone during half of the working hours ($4h$) plus during driving ($1.5h$) and lunch ($0.5h$), the phone is with the user for merely 6 out of the 16 hours in a day, and is placed elsewhere for the remaining $10h$. On the contrary, the user wears the watch throughout the entire day. Therefore the total duration that the user carries the phone ($T_{WithPhone}$) is $6h$ each day, while the total duration that the user does not carry the phone ($T_{WithoutPhone}$) is $10h$.

Figure 3.15 shows the real inference accuracy of the AR app considering the routine above. Relying solely on the phone sensors for AR will lead to a poor real inference accuracy due to the limited sensor coverage of the phone, as seen in the case of SC2 (64.78%) and SC4 (65.51%). Accuracy gets notably improved with the help of smartwatches. Using only the watch sensor (SC1) will result in a real accuracy of 87.50%, or 35.1% improvement compared with using phone only. Moreover, by fusing the watch and phone sensors together, the real accuracy of AR can be improved to 87.85% or by 35.6% without the GPS (SC3), and to 89.01% or by 37.4% with the GPS (SC5). The improvement of fusing sensors from both devices is not significant, partly because the accuracy is already rather good using only the watch sensors. For the HR Monitor app, although accuracy will remain the same whether the inference is on the watch or on the phone, the app is made possible only because of the heart rate sensor available on smartwatches.

Finally, although the values of $T_{WithPhone}$, $T_{WithoutPhone}$, and the resulting accuracy improvements all depend on the daily schedule of a user, we emphasize that a user would typically not carry the phone for an extended period of time in everyday life but would wear the watch for longer. This will lead to $T_{WithoutPhone} > T_{WithPhone}$ and therefore similar accuracy improvements as shown above.

3.4.4.6 Energy Optimization: Optimal Inference Partition

To achieve Goal #4, we discuss the optimal module partitioning of the inference pipelines shown in Figure 1.1. If the inference uses only sensor data from the watch, for example, the AR app with SC1 and the HR Monitor app, there exists several partition strategies:

AR app: 1. Execute the full pipeline on the watch, and notify the user in real time (AR-

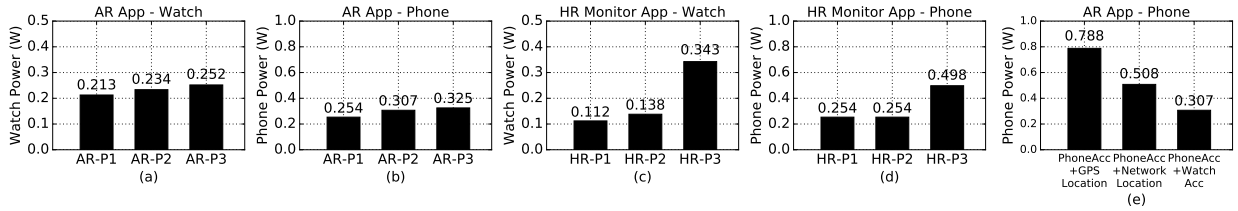


Figure 3.16: Average power consumption of **(a)** the *watch* with different partitions of the AR app; **(b)** the *phone* with different partitions of the AR app; **(c)** the *watch* with different partitions of the HR Monitor app; **(d)** the *phone* with different partitions of the HR Monitor app; **(e)** the *phone* with the AR app using and not using GPS.

P1); 2. Sample sensor data and perform feature extraction on the watch, send the calculated features to the phone, and perform classification on the phone (AR-P2); 3. Sample sensor data on the watch, send the raw data to the phone via BLE, and perform feature extraction & classification on the phone (AR-P3).

HR Monitor app: 1. Execute the full pipeline on the watch, and send the user alerts in real time (HR-P1); 2. Execute the full pipeline on the watch, but log data to SD card (HR-P2); 3. Perform sampling and buffering on the watch, send raw heart rate data to the phone via BLE, and notify the user on the phone (HR-P3).

Figure 3.16 (a) and (b) illustrate the average power consumption of the watch and the phone running the AR app respectively, with different partitions of the inference pipeline. Figure 3.16 (c) and (d) show results for the HR Monitor app. Since we assume both inferences use a $1s$ classification window, the average power consumption is equivalent to the energy consumption of the device. For AR-P1, HR-P1, and HR-P2 we use the sleeping power consumption of the phone because there is no computation or communication on the phone. For both apps executing the entire pipeline on the watch (P1) yields the optimal energy efficiency. For the AR app, the energy consumption of AR-P1 is 15.5% less on the watch and 21.8% less on the phone than AR-P3, respectively. For the HR Monitor app, the energy savings of HR-P1 on the watch and on the phone compared with HR-P3 are 67.3% and 49.0% respectively. This is mainly because energy consumed for computation is much less than the cost of transmitting data through BLE, especially when high-dimensional raw data is being used as seen in AR-P3 and HR-P3. The results also suggest that performing computation

on the watch can be more energy efficient than simply use the watch as a UI device.

When an inference uses sensor data from both the watch and the phone, as seen in the AR app with SC3 and SC5, we only need to compare AR-P2 and AR-P3 in Figure 3.16 (a) and (b) because a phone app is always required to coordinate sensor data from both devices and to complete the inference. In this case, it is more energy efficient to send calculated features from the watch to the phone (AR-P2) than sending raw data (AR-P3). The energy saving is 7.1% on the watch and 5.5% on the phone again because of less BLE data transmission.

3.4.4.7 Energy Optimization: Replacing High-power Sensor

According to Goal #5, the additional low-power sensors on the smartwatch can replace the high-power ones required before in context inferences. As shown in Section 3.4.4.5, using watch accelerometers and phone accelerometers (SC3) for the AR app can result in similar or even better real inference accuracy compared with solutions that use phone GPS (SC4 and SC5). Therefore we study the use of smartwatches in context inference to eliminate the use of phone GPS and to reduce the energy consumption of the phone. Figure 3.16 (e) shows the average power consumption of the phone when the AR app is running using different sensor combinations, with the watch sending calculated feature to the phone (AR-P2) in the last collaboration case. By replacing the phone location sensor used in the AR app with the watch accelerometer, the phone consumes 61.0% and 35.5% less energy compared with the cases where GPS locations and network locations are used, respectively.

Although our experiments did not consider the power consumed by other workloads and cellular radios, the reduced energy consumption of inferences can nevertheless extend the battery life of both the watch and the phone enabling them to execute more workload during the day.

3.5 Summary

In this chapter we quantify the benefit of using deep learning and heterogeneous devices and processors for pervasive context inferences:

- We have achieved comparable inference accuracy as traditional models and acceptable latency using deep learning without hand-picking features.
- We propose that certain stages of the inference pipeline can be off-loaded from main app processors to CPUs and DSPs. We have shown up to 30× latency speed-up from running deep learning tasks on mobile GPUs and up to 60% energy saving from off-loading sensing tasks to mobile DSPs.
- We also propose that context inferences can be executed across smartwatches and smartphones. From the watch-phone coordination, with two example inferences we demonstrate an accuracy improvement of up to 37% from the increased device sensor coverage. We also showcase a 67% energy saving from partitioning inferences across watches and phones, and a 61% saving as a result of replacing energy-hungry sensors with watch inertial sensors.

CHAPTER 4

Putting It Together: A Programming Framework for Context Inferences

Motivated by the MiLift app and our observations from optimizing the accuracy and energy consumption of context inference apps, we realize that ad-hoc optimizations and algorithms may limit the adoption by developers at a larger scale. In this chapter, we propose a framework with a set of programming abstractions and an associated runtime for the development and execution of context inference apps. The framework helps app developers compose inferences in a modular and systematic fashion while tackling a set of runtime challenges.

4.1 Design Challenge and Contribution

Connected sensing devices, such as cameras, thermostats, in-home motion, door-window, energy, water sensors [ama], collectively dubbed as the *Internet of Things* (IoT), are rapidly permeating our living environments [bcc11], with an estimated 50 billion such devices in use by 2020 [Eva11]. In theory, they enable a wide variety of apps spanning security, efficiency, healthcare, and others. But in practice, developing IoT apps is arduous because the tight coupling of apps to specific hardware requires each app to implement the data collection logic from these devices and the logic to draw inferences about the environment or the user.

Unfortunately, this monolithic approach where apps are tightly coupled to the hardware, is limiting in two important ways. First, for app developers, this complicates the development process, and hinders broad distribution of their apps because the cost of deploying their specific hardware limits user adoption. Second, for end users, each sensing device they install is limited to a small set of apps, even though the hardware capabilities may be useful

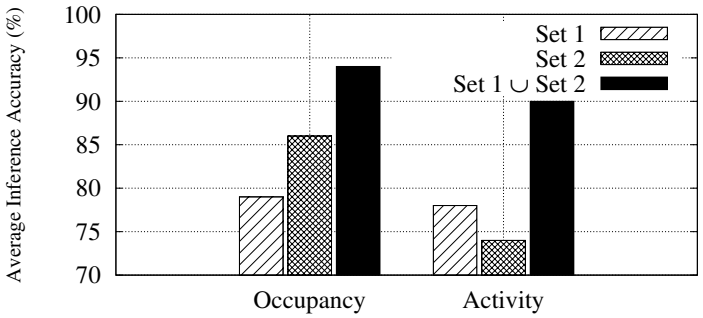


Figure 4.1: Improvement in occupancy and activity inference accuracy by combining multiple devices in a lab deployment. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 ={PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit [fit]}.

for a broader set of apps. How do we break free from this monolithic and restrictive setting? Can we enable apps to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address the problem of monolithic app development for connected devices, we start from an insight that many inferences required by apps can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred by either detecting motion or recognizing people in images, with data sampled from motion sensors (such as those in security systems or Nest [nes]), cameras (e.g. Dropcam [dro], Simplicam [sim]), microphone, smartphone GPS, or using a combination of these sensors, since each may have different sources of errors. *We posit that inference logic, traditionally left up to apps, ought to be abstracted out as a system service, thus decoupling "what is sensed and inferred" from "how it is sensed and inferred".* Such decoupling enables apps to work in heterogeneous environments with different sensing devices while at the same time benefiting from shared and well trained inferences. Consequently, there are three key challenges in designing such a service:

Device selection: The service must be able to select the appropriate devices in a deployment that can satisfy an app's inference request (including inference accuracy). Device selection helps apps to run in heterogeneous deployments. It also helps apps to operate in

settings with user mobility where the set of usable devices may change over time. Moreover, apps can leverage multiple available devices to improve inference accuracy, as shown in Figure 4.1.

Efficiency: For inferences that are computationally expensive to run locally on user devices, or to support deployments that span geographical boundaries, the service should be able to offload computation to remote servers. In doing so, the service should partition computation while efficiently using network bandwidth.

Disconnection tolerance: The service should be able to handle dynamics that can arise due to device disconnections and user mobility.

To address these challenges concretely, we propose *Beam*, an app framework and associated runtime which provides apps with inference-based programming abstractions [SSP15a, SSP15b, SSP16]. It introduces the key abstraction of an **inference graph** to not only decouple apps from the mechanics of sensing and drawing inferences, but also directly aid in addressing the challenges identified above. apps simply specify their inference requirements, while the Beam runtime bears the onus of identifying the required sensors in the given deployment and constructing an appropriate inference graph.

Inference graphs are made up of modules which are processing units that encapsulate inference algorithms; modules can use the output of other modules for their processing logic. Beam introduces three simple building blocks that are key to constructing and maintaining the inference graph: typed inference data units (IDUs) which guide module composability, channels that abstract all inter-module communications, and coverage tags that aid in device selection. The Beam runtime instantiates the inference graph by selecting suitable devices and assigning computational hosts for each module. Beam also mutates this assignment by partitioning the graph at runtime for efficient resource usage. Beam's abstractions and runtime together provide disconnection tolerance.

Our implementation of the Beam runtime works across Windows PCs, tablets, and phones. Using the framework, we develop two realistic apps, eight different types of inference modules, and add native support for many different types of sensors. Further, Beam

supports all device abstractions provided by HomeOS [DMA12], thus enabling the development of a variety of inference modules. We find that for these apps: 1) using Beam's abstractions results in up to 4.5× fewer development tasks and 12× fewer source lines of code with negligible runtime overhead; 2) inference accuracy is 3× higher due to Beam's ability to select devices in the presence of user mobility; and 3) network resource usage due to Beam's dynamic graph partitioning matches hand-optimized versions for the apps.

4.2 Beam Overview

In this section, we first describe two representative classes of apps and distill the challenges an inference framework should address. Next, we describe the key abstractions central to Beam's design in addressing the identified challenges.

4.2.1 Example Apps

Our motivation for designing Beam are data-driven-inference based apps, aimed at homes [sma, nes], individual users [quaa, mapa, WCC14, WCB11, RAP11] and enterprises [KAB05, RPG07, BNJ11, sho, iBe]. We identify the challenges of building an inference framework by analyzing two popular app classes in detail, one that infers environmental attributes and another that senses an individual user.

Rules: A large class of popular apps is based on the 'If This Then That (IFTTT)' pattern [ift, UMP14]. IFTTT enables users to create their own rules connecting sensed attributes to desired actions. We consider a particular rules app which alerts a user if a high risk appliance, e.g., electric oven, is left on when the home is unoccupied [SKB13]. This app uses the appliance-state and home occupancy inferences.

Quantified Self (QS) [quaa, mapa, MBM08, FFO12, ABS05] disaggregates a user's daily routine by tracking her physical activity (walking, running, etc), social interactions (loneliness), mood (bored, focused), computer use, and more.

Using these two popular classes of apps we address three important challenges they pose:

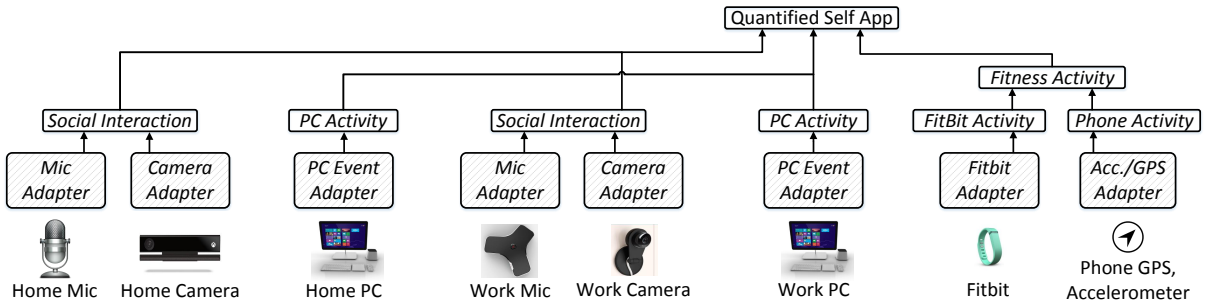


Figure 4.2: Inference graph of modules for the Quantified Self (QS) app. Adapters are device driver modules.

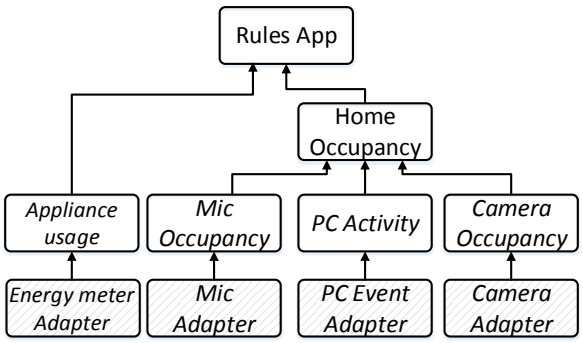


Figure 4.3: Inference graph for the Rules app.

device selection, efficiency, and disconnection tolerance, as detailed in Section 4.1. Next, we explain the key abstractions in Beam aimed at tackling these challenges.

4.2.2 Beam Abstractions

In Beam, *app developers* only specify their desired inferences. To satisfy the request, Beam bears the onus of identifying the required sensors and inference algorithms in the given deployment and constructing an inference graph.

Inference Graphs are directed acyclic graphs that connect devices to apps. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of *inference data units (IDUs)* between modules. While these abstractions are described in more detail below, Figure 4.2 shows an example inference graph for the QS app that we later build and evaluate. The graph uses eight different devices spread

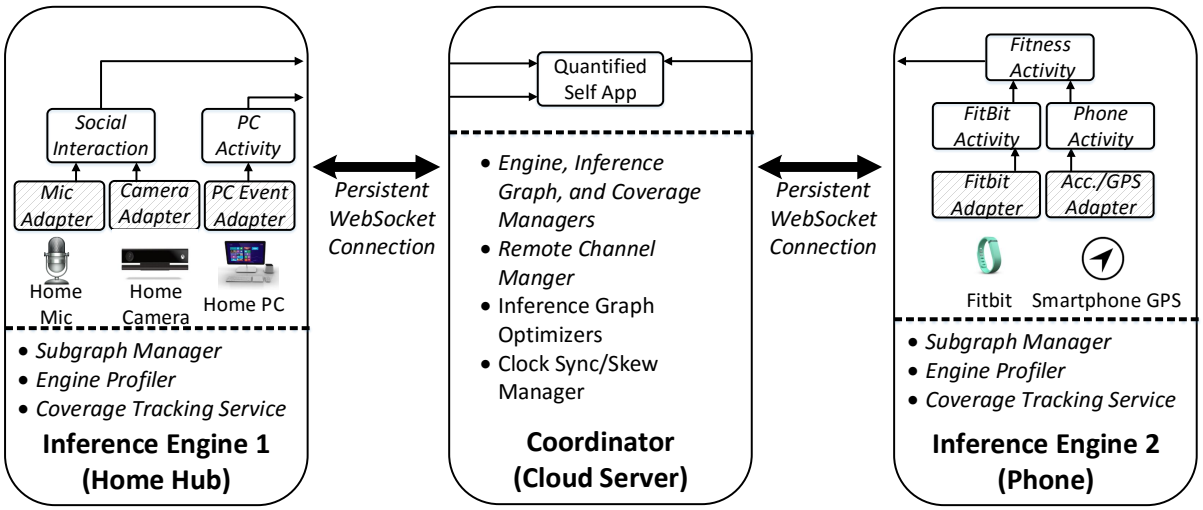


Figure 4.4: Overview of different Beam components in a deployment with 2 Engines.

across the user's home and workplace, and includes mobile and wearable devices. The app requests a top-level inference as an IDU and Beam dynamically selects the modules that can satisfy this inference based on the devices available. For example, in Figure 4.2, to satisfy the app's request for inferences pertaining to fitness activities Beam uses a module that combines inferences drawn separately from a user's smartphone GPS, accelerometer, and Fitbit device, thus forming part of the inference graph for QS. Figure 4.3 shows the inference graph for the Rules app.

Composing an inference as a directed graph enables sharing of data processing modules across apps and other modules that require the same input. In Beam, each computing device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the **Engine**. Engines are computational hosts for inference graphs. Figure 4.4 shows two engines, one on the user's home hub and another on her phone; the inference graph for QS (shown in Figure 4.2) is split across these engines, while the QS app runs on a cloud server. For simplicity, we do not show another engine that may run on the user's work PC.

IDU: An *Inference data unit (IDU)* is a typed inference, and in its general form is a tuple <t,e,s>, which denotes any inference with state information s, generated by an inference algorithm at time t and error e. The types of the inference state s, and error e, are specific to the inference at hand. For instance, s may be of a numerical type such as a double

```
<Spec>
  <ControlParameters> <!-- Module parameters -->
    <Param name="sampleSize" type="int" value="5"/>
  </ControlParameters>

  <Output> <!-- Output channel IDU spec -->
    <Inference type="Beam.IDU.HomeOccupancyIDU"/>
  </Output>

  <Input> <!-- Input channels -->
    <InputBlock type="OR">
      <InputChannel Mode="FreshPush">
        <Module type="Beam.Modules.PCActivity"/>
        <Module type="Beam.Modules.MicOccupancy"/>
        <Module type="Beam.Modules.CameraOccupancy"/>
      </InputChannel>
    </InputBlock>
  </Input>
</Spec>
```

Listing 4.1: Module specification of Home Occupancy.

(e.g., inferred energy consumption), or an enumerated type such as high, medium, or low. Similarly, error e may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred". The latter is handled by inference modules, which we describe next.

Inference Modules: Beam encapsulates inference algorithms into modules. Inference modules consume IDUs from one or more modules, perform certain computation using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters are device drivers that decouple "what is sensed" from "how it is sensed". *Inference developers* specify (i) a module's input dependencies (either as IDU types or as modules), (ii) the IDU type it generates, and (iii) its configuration parameters. Modules have complete autonomy over how and when to output an IDU, and can maintain arbitrary internal states. Listing 4.1 shows a specification for the Home Occupancy inference module in the Rules inference graph (Figure 4.3). It lists (i) input dependencies of PC Activity *OR* Mic Occupancy *OR* Camera Occupancy, (ii) $HomeOccupancyIDU$ to be the type of output it generates, and (iii) a control parameter, *sampleSize*, that specifies the temporal size of input samples (in seconds) to

consider in the inference logic. app developers request the local engine for desired inferences, for example:

```
engineInstance.Request(
    Beam.Modules.ModHomeOccupancy,
    tags, Mode.FreshPush);
```

These are satisfied by inference modules implemented by inference developers, and apps receive IDUs via a callback.

Channels: To ease inference composition, *channels* link modules to each other and to apps, abstracting away the complexities of connecting modules across different devices. Channels provide support for disconnections tolerance and enable optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. Remote channels enable apps and inference modules to seamlessly use remote devices or modules. Channels can be either configured to deliver IDUs to the reader as soon as the writer pushes it (*FreshPush*, as seen in Listing 4.1 line 12), or to deliver IDUs in batches thus amortizing the cost of computation and network transfers.

Coverage Tags: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags which describes what the sensor is sensing. For example, a location string tag can indicate a coverage area such as "home" and a remote monitoring app can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This gives apps complete flexibility in defining new tag types. Adapters are assigned tags by the respective engines at setup time, and are updated at runtime to handle dynamics (Section 4.3.1).

Beam's runtime also consists of a **Coordinator** which interfaces with all engines in a deployment and runs on a replicated server that is reachable from all engines. The coordi-

nator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent web-socket connection, and instantiate and manage the parts of inference graphs local to them.

4.3 Beam Runtime

In this section, we describe how the Beam runtime uses the inference graph to aid in device selection, efficient graph partitioning, and handling device disconnections.

4.3.1 Device Selection

Beam simplifies app development by automatically selecting devices that match its inference request in heterogeneous deployments and in the presence of user mobility. Beam leverages the device discovery mechanism in HomesOS [DMA12] to discover and instantiate adapter modules for available sensors in the deployment.

apps request their local Beam engines for all inferences they require, including the coverage associated with each inference. All app requests are forwarded to the coordinator. Using inference module specifications and devices with matching coverage tags available in the deployment [1], the coordinator recursively resolves all required inputs of each module. A module's coverage tag set includes tags from the downstream modules it processes data from.

Handling environmental dynamics: Movement of users and devices can change the set of sensors and devices that satisfy an app's requirement. For instance, consider an app that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, work PC, or smartphone. In such scenarios, the inference graph

[1] The requested tag must match one of the adapter tags.

needs to be updated dynamically. Beam updates the coverage tags to handle such dynamics. Tags of *location* type (e.g., "home") are assumed to be static and are only edited by the user. For tags of type *user*, the sensed subject is mobile and hence the sensors that cover it may change. The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines.

The user tracking service updates the coverage tags as the user moves. When a user leaves home for work, the tracking service removes the *user* tag from device adapters on the home PC and adds them to adapters on her smartphone. When she arrives at work, the tracking service removes the user tag from her smartphone and add them to adapters on her work PC. The user tracking service relies on device interactions. When a user interacts with a device, it updates the tags of all sensors on the device to include the user's tag.

Finally, changes in coverage tags (e.g., due to user movements) or device availability (e.g., device disconnections and re-connections) will result in the coordinator reselecting devices for requested inferences and recreating the graph accordingly.

4.3.2 Inference Partitioning for Efficiency

Beam uses the inference graph for partitioning computation across devices and optimizing for efficiency.

Graph creation and partitioning: The Beam coordinator maintains a set of inference graphs in memory as an *incarnation*. When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, re-using already running modules, and merges inference graphs if needed. Once the coordinator finishes resolving all required inputs for each module in the inference graph, it determines where each module should run using the optimization schemes described next. The coordinator then initializes remote channels and partitions the graph into engine-specific subgraphs which are sent to the engines. Whenever the tracking service updates coverage tags, e.g. due to user movements, the coordinator re-computes the inference graphs and sends updated subgraphs to the affected engines. Next, the engines receive their respective subgraphs, compare each

subgraph to existing ones, and update them by terminating deleted channels and modules before initializing new ones. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

Optimizing resource usage: In Beam, optimizations are either performed *reactively*, i.e., when an app issues/cancels an inference request, or *proactively* at periodic intervals.

Beam's default reactive optimization determines where each module should run by partitioning the inference graph to minimize the number of remote channels. Let $G(V, E)$ be an inference graph, where V represents the nodes (inference modules), and E represents its adjacency matrix. In E, e_{ij} is the cost of the edge (channel) connecting module i to module j; $e_{ij} = 0$ if two modules are not connected directly. Beam's optimizer determines potential partitions of the inference graph and picks the partition with the minimum cost. To determine a partition $P_{|V|\times|D|}$, Beam assigns each module $i \in V$ to run on a device $d \in D$. That is, $p_{id} = 1$ if module i runs on device d and $p_{id} = 0$ otherwise. We define the cost matrix of a partition P of the inference graph as $C_{|D|\times|D|} = P^T EP$, where $c_{d_1 d_2}$ denotes the sum of the cost of all channels from device d_1 to device d_2. Since the reactive optimizer aims at minimizing the number of remote channels, here $e_{ij} = 1$ for all connected modules i and j in the graph. An adapter module runs on a device co-located with the sensor, and an app runs on the device requested by the user. Beam solves the following linear program to find P with the minimum cost:

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{\forall d_1, d_2 \in D, d_1 \neq d_2} c_{d_1 d_2} \\
\text{subject to} \quad & \sum_{d \in D} p_{id} = 1 \qquad \forall i \in V \\
& p_{id} \in \{0, 1\} \qquad \forall i \in V,\ \forall d \in D
\end{aligned}
$$

Beam's default proactive optimization minimizes the amount of data transferred over remote channels by solving the same linear program but using the data rate profile of each edge as e_{ij}. Engines profile their subgraphs, and report profiling data (e.g., per-channel data rate or estimated per-module CPU utilization) to the coordinator periodically. Other po-

tential optimizations can minimize CPU/memory usage at engines, or IDU delivery latency. Beam allows for modular replacement of optimizers. The coordinator applies optimizations by re-configuring inference graphs and remapping the engine on which each inference module runs.

Scatter node optimization: The coordinator further optimizes the inference graph by finding remote channels which have the same writer module, and whose readers reside on a common engine (R_e). For each such set of edges (E), it adds a single remote channel edge from the writer to a new *scatter* node at R_e. The scatter node is then set as the writer for all edges in E, in effect, replacing multiple remote channels with one and reducing the amount of *wide-area* network transfers by a factor of $|E|$.

4.3.3 Disconnection Tolerance

Beam's remote channels always go through the coordinator and support reader/writer disconnections by using buffers at the coordinator. Thus, a channel is split into three logical components: writer-side, reader-side, and coordinator-side (present only in remote channels). A channel's writer-side and coordinator-side component buffer IDUs. Channels offer two guarantees: i) readers do not receive duplicate IDUs, and ii) readers receive IDUs in FIFO timestamp order. Beam specifies a default size for remote channel buffers but also allows app developers to customize buffer sizes based on deployment scenarios, e.g., network delays and robustness.

Internally, channels assign sequence numbers to IDUs. They are used for reader-writer flow control, and in remote channels for applying back-pressure on the writer-side component when the coordinator-side buffer is full, e.g., when a reader is disconnected. Currently, the writer-side and coordinator-side buffers use the drop-tail policy to minimize data transfer from writer to coordinator in the event of a disconnected/lazy reader (as opposed to drop head). This design implies that after a long disconnection a reader will first receive old inference values followed by recent ones.

Channels and modules do not persist data. If necessary, apps and modules may use a

Adapter	Inference Module
PC Event	PC Activity
PC Input	PC Occupancy
Phone GPS	Semantic Location
Accelerometer Fitbit	Fitness Activity
Energy Meter (HomeOS)	Appliance Usage
Camera (HomeOS)	Camera Occupancy
PC Mic/Tablet Mic	Mic Occupancy
PC Mic/Tablet Mic	Social Interaction

Table 4.1: Sample adapters and inference modules.

temporal data store, such as Bolt [GSP14], to make inferences durable.

4.4 Implementation

Our Beam prototype is implemented in C# as a cross-platform portable service that can be used by .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 apps. The Beam inference library has sample implementations for 8 inference modules and 9 adapters (listed in Table 4.1). It also includes a HomeOS-adapter that allows Beam to leverage various other device abstractions provided by HomeOS [DMA12], such as the camera and energy meter device drivers used by some of our sample inferences. Each Beam module has a single data event queue and a thread to deliver received IDUs (akin to the actor model [Arm10, BBG14, BGK11]). All communication between the coordinator and engine instances uses the SignalR [sig] library, and Json.NET [jso] is used for data serialization. The engine library, coordinator, sample adapters, and tracking service are implemented in 6614, 952, 1824, and 219 (total=9609) source lines of code respectively.

4.4.1 Sample apps

We implement the motivating apps described in Section 4.2.1 in Beam. Inference graphs of Rules and Quantified Self (QS) are shown in Figure 4.3 and Figure 4.2, respectively. Device adapters such as Microphone, Camera, and PC Event adapters are shared by both inference

graphs. For common inference modules such as the PC Activity inference, Beam instantiates only one of them across these graphs. Changes in coverage tags and device availability caused by user mobility prompt Beam to re-select appropriate devices for inference graphs. For instance, PC Activity for QS might either be drawn from the home PC or the work PC depending on the user's current location.

4.4.1.1 Rules app

The Rules app requires the Appliance Usage and Home Occupancy inferences implemented as follows.

The *Appliance Usage* inference module reads aggregated power consumption of a home from a whole-home power meter, or a utility smart-meter, and disaggregates it to determine the set of appliances that are on at any given instant, using the CO algorithm from [Har92], configured with 10 commonly owned home appliances [BKP14]. The whole-house power readings are generated using our power-sensor adapter, which interfaces with an Aeon ZWave whole-house meter [aeo].

The *Mic Occupancy* inference module reads audio samples using the PC Microphone adapter at a sampling rate of 8 kHz (in 4 second frames), and filters out background noise (such as wind, fans, etc.) [HXZ13]. If after filtering, the audio sample still indicates sound is present, the inference output is 'occupied'.

The *PC Activity* module infers the current activity a user is performing on a PC (described in Section 4.4.1.2).

The *Camera Occupancy* module receives streaming video input from an adapter provided by the HomeOS web-cam driver. The input video is of 640×480 resolution and streams at a frame rate of 1 fps. The module compares consecutive frames in the video. If any significant difference indicating possible human movement is detected [BJM13], the inference output is 'occupied'.

The *Home Occupancy* module combines Mic Occupancy, Camera Occupancy, and PC Activity modules, to produce a Home Occupancy inference, outputting 'occupied' if one of

the following is true: Mic Occupancy, Camera Occupancy, or PC Activity $\neq$ No activity.

4.4.1.2 Quantified Self (QS) app

QS tracks a user's fitness activities, social behaviors, and computing activities on a PC. It is implemented as a Windows Azure web app. Users view plots of their data at leisure on the QS webpage. The inference modules used by this app are described as follows.

The *Social Interaction (Is Alone)* module detects the presence of human voice, outputting 'user not alone' when human voice is present (likely due to conversations with others, though false positives may arise due to TV sounds and background noises). It computes the mel-frequency cepstral coefficients (MFCC) [DM80, Mer76] over a 200 ms window of the microphone adapter data at 44.1 kHz and uses a decision tree [Qui86] to classify if human voice is present. The module also incorporates movement detection by analyzing video streams from the camera.

The *PC Activity* inference module reads the name of the currently active desktop window from the PC-event adapter using a Win32 system call. It then classifies the name into one of the known PC activity categories (coding, web browsing, social networking, emailing, reading etc.) using a pre-configured mapping. It also infers the psychological state of the user (bored vs. focused) using the features proposed in [MIC14], including window switches, web page switches, time spent browsing Facebook.com, and time spent using e-mail.

The *Fitness Activity* module implements the algorithm from [RMB10] to infer human transportation modes (still, walking, driving) using the phone accelerometer. It also uses the Fitbit [fit] API to fetch users' FitBit activity logs, and combines it with accelerometer-based inferences.

4.4.2 APIs

Listings 4.1 and 4.2 show how app and inference developers leverage the Beam APIs using the Home Occupancy inference as an example.

Inference developers provide an XML specification for each inference module (Listing 4.1) configuring its parameters as well as the input and output channel IDU types. They then implement the module using Beam's APIs (Listing 4.2, line 1-19) extending the `InferenceModuleBase` helper class. The module is first initialized with control parameters (line 5). It receives inputs in the `DataReceived` callback (line 9), performs the implemented inference logic (line 11), and sends result IDUs to output channels (line 14-16).

```
1  // Inference developers implement module logic
2  public class ModHomeOccupancy:InferenceModuleBase {
3    // Read parameters from the specification XML file
4    public override void Initialize(ModuleSpec spec) {
5      this.paramList = spec.getControlParams();
6      // set state and initialize using parms ...
7    }
8    // Callback to receive IDUs from input channel(s)
9    public override void DataReceived(IChannel channel, List<IIDU>
       inputSignals) {
10     // Compute occupancy based on input
11     HomeOccupancyIDU inferenceResult =
12                     computeOccupancy(inputSignals);
13     // Push result IDUs to output channel(s)
14     if (!changedSinceLastPush(inferenceResult))
15       foreach (IChannel ch in outputChannels)
16         ch.Push(inferenceResult);
17   }
18   // ...
19 }
20 // App developer: request inferences from engine
21 public class QSApp : InferenceModuleBase {
22   void startInference() {
23     // Get an instance of the local engine
24     Beam.Engine engine = Beam.Engine.Instance;
25     // Prepare coverage tags
26     List<CoverageTag> tag = new List<CoverageTag>();
27     tag.Add(new PersonCoverageTag("User1"));
```

```
28      // Register for inference notifications
29      engine.Request(Beam.Modules.ModHomeOccupancy, tag, Mode.FreshPush,
            this);
30    }
31    // Callback to receive IDUs from input channel(s)
32    public override void DataReceived(IChannel channel, List<IIDU>
          occupancyInferences ) {
33      // Perform actions based on IDUs received ...
34    }
35  }
```

Listing 4.2: Example usage of the Beam API.

App developers simply request a specific inference module, e.g. Home Occupancy (Listing 4.2, line 20-35). The app specifies coverage tags (line 26-27), and invokes the local engine's `Request` method (line 29) to register for inference notifications. Beam then instantiates the required inference graph and returns a channel to the app with the requested module as writer. Result IDUs are received by the app via the `DataReceived` callback (line 32).

4.5 Evaluation

We evaluate how Beam's inference graph abstraction simplifies app development, benchmark its performance, and evaluate its efficacy in addressing the three key challenges identified in Section 4.1. Our evaluation uses micro-benchmarks as well as the two motivating apps from Section 4.4.1.

First, in Section 4.5.2 we quantify how Beam's abstractions simplify app development and evaluate the overhead of graph creation. Then, in Section 4.5.3, we evaluate how Beam's device selection in a real-world deployment with user mobility improves inference accuracy. Next, in Section 4.5.4, we show the impact of Beam's inference graph partitioning to optimize for efficient resource usage. Finally, in Section 4.5.5 we showcase Beam's ability to handle device disconnections.

For our experiments, the Beam coordinator runs on a Windows Azure VM (with AMD Opteron Processor, 7 GB RAM, 2 virtual hard disks, running Windows Server 2008 R2); the engines run on desktop machines (with AMD FX-6100 processor, 16 GB RAM, running Windows 8.1) and a Windows Phone (Nokia Lumia). Both sample apps, Rules and Quantified Self (QS), run on the same VM as the coordinator; local engines run in the cloud, a home PC, phone, and a work PC.

4.5.1 Development Approaches

To quantify the reduction in development effort achieved by Beam, we explore different approaches that a developer may adopt to design such apps.

Monolithic-All Cloud (M-AC). In this approach, the app is developed as a monolithic silo without the use of any framework. All app logic is tightly coupled to the sensing devices, and all collected data is relayed to cloud services, as is the case with Xively [xiv] and SmartThings [sma]. The cloud service runs the app's data processing and inference logic.

Monolithic-Cloud and Device (M-CD). In this approach, an app developer hard-codes the division of inference logic across the cloud VM and end devices [opta, WCC14]. Thus, sensor values are processed to some degree on the end device before being uploaded to the cloud VM which hosts the remainder of the app logic. Depending on the deployment and resource constraints, the developer may need to hand-optimize the resource usage (e.g., CPU, memory, or network usage).

Monolithic-using inference libraries (M-Lib). This approach is similar to the previous one (M-CD), except that app developers may use libraries of inference algorithms tuned by domain experts, thus leading to some reduction in development effort [CLL11, NDA13a, KSB13].

Monolithic-using sensor hub systems (M-Hub). Platforms such as HomeOS [DMA12], Homeseer [hom], and others [rev], facilitate the development of apps by providing homogeneous device-based programming abstractions. Typically, these platforms implement sensor drivers and regulate access to different sensors; apps still implement inference logic.

Beam. In this approach, an app on any of the user's devices simply presents its inference

Application components and their description
Sensor driver: *Handled by M-Hub and Beam* One driver per sensor type.
Inference logic: *Handled by M-Lib and Beam* For each inference an application requires, at least one inference component is needed, e.g., incorporating feature extraction techniques, inference algorithm, learning model, etc.
Parameter tuning: *Simplified by Beam* An application must also incorporate logic to match its inference logic with the underlying sensors (for a range of sensors), e.g. configuring sensor-specific parameters such as sampling rate, frame rate for cameras, sensitivity level for motion sensors, etc.
Cloud service: *Simplified by Beam* Depending on the development approach, an application may require several cloud services, e.g., a storage service for data archival, an execution environment for hosting inference logic, authentication services, etc.
Device disconnection tolerance: *Handled by Beam* Since devices such as smartphones, tablets, may have intermittent connectivity, developers need to appropriately handle disconnections.
User interface (UI): *Simplified by Beam* Typical applications require certain UI components, e.g., to allow configuration of sensors for data collection, or for users to view results.

Table 4.2: Components of inference-based applications.

requests to the local Beam instance. Using the inference graph abstraction, Beam bears the onus of device selection, optimizing for efficiency, and handling disconnections. Note that using Beam does not preclude the M-Hub approach where all sensing and inference logic run locally on a single hub device (e.g., a home hub). We refer to such scenarios built using Beam's inference abstractions as Beam-Hub, with the engine and coordinator running locally without needing an external network connection.

4.5.2 Evaluation of Inference Abstraction

In this section we highlight the saving in app development effort using Beam's inference graph abstraction and quantify the overhead of graph creation.

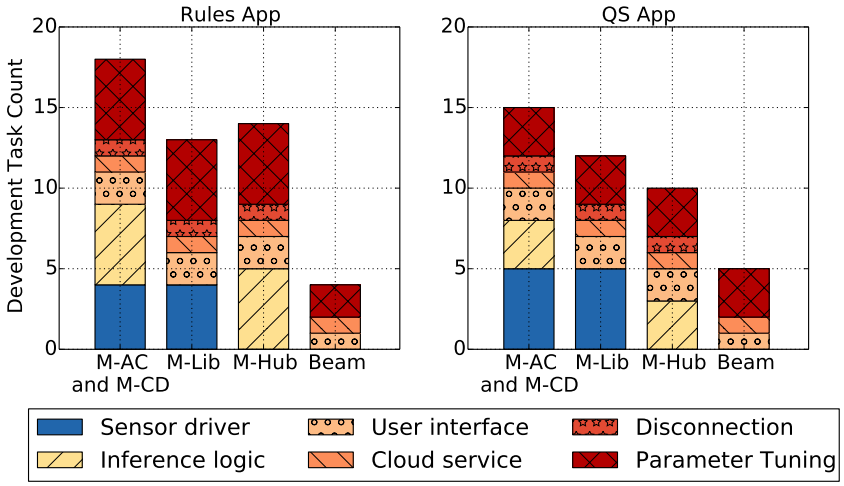


Figure 4.5: Development tasks using different development approaches in the two apps (Rules, QS)

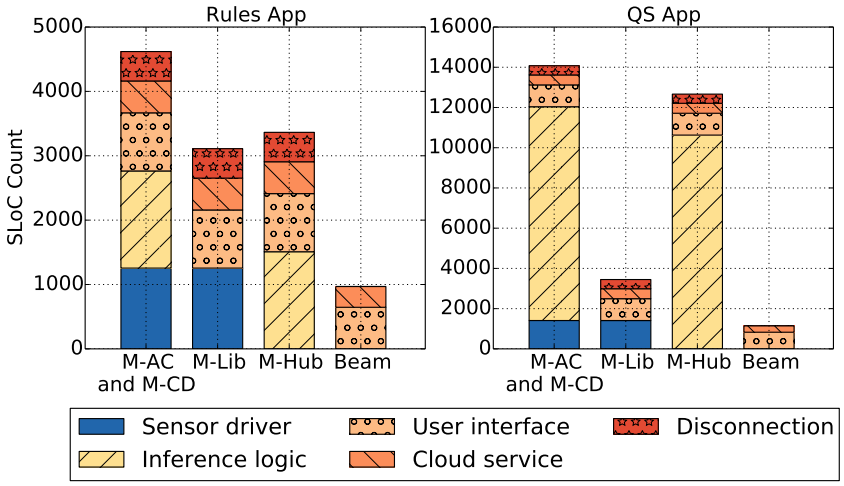


Figure 4.6: SLoC using different development approaches in the two apps (Rules, QS)

4.5.2.1 Comparison of Development Effort

We implement our representative apps using the different development approaches described above and present a quantitative comparison of the development effort using two metrics: (i) number of development tasks and (ii) number of source lines of code (SLoC). *Number of development tasks* is defined as the number of architectural components that need to be designed, implemented, and maintained for a complete functioning app. To analyze development effort in greater depth, these components can further be categorized based on the

function they perform (Table 4.2). This metric captures the diverse range of tasks developers of apps for connected devices are required to handle. Although comparing the number of tasks provides insight into the development effort required for each approach, different components often require varying levels of implementation efforts. Thus, to distinguish individual components, we also measure the *number of source lines of code (SLoC)* required for the components in each approach.

Figures 4.5 and 4.6 show the number of development tasks and number of SLoC, respectively, for the Rules and QS apps using the different development approaches. We observe that for the Rules app, Beam reduces the number of development tasks by 4.5×, and the number of SLoC by 4.8×, compared with M-AC and M-CD. Similarly, for the QS app, Beam reduces the number of development tasks by 3×, and the number of SLoC by 12×, compared with M-AC and M-CD.

Number of development tasks: As shown in Figure 4.5, the approaches of Monolithic-All Cloud (M-AC) and Monolithic-Cloud and Device (M-CD) have similar number of development tasks for both the Rules (on left) and the QS app (on right). M-CD requires developers to hard-code the division of tasks between end-point devices and cloud servers, thus statically optimizing for better resource usage than M-AC (Section 4.5.4).

Compared with M-AC and M-CD, the M-Lib approach reduces developer effort. It leverages existing libraries which provide implementations of inference algorithms and also handle their training and tuning. Similarly, in the M-Hub approach, developer effort is reduced due to existing sensor driver implementations provided by the platform. Finally, when using Beam, app developers do not need to design or implement sensor drivers, inference logic, tuning timing parameters, or handling disconnections. app developers only need to decide their required inferences, and develop app-specific components, e.g., user interface, third-party authentication, etc.

Number of SLoC: As shown in Figure 4.6, we observe that for all approaches, the SLoC count is generally proportional to the development task count. For most approaches SLoC is dominated by tasks of developing sensor drivers and inference logic. For instance, the

	Sample scenario 1 (local inference)		Sample scenario 2 (remote inference)		
	App1's request	**App2's request**	**App1's request**	**App2's request**	**Reevaluation**
Total	232.54 ± 1.63	246.71 ± 16.62	237.43 ± 12.76	230.24 ± 3.53	-
Request and subgraph transfer	230.35 ± 1.68	246.24 ± 16.62	236.13 ± 12.75	229.73 ± 3.47	-
Coordinator (graph creation)	1.05 ± 0.14	0.16 ± 0.01	0.90 ± 0.04	0.20 ± 0.07	0.12 ± 0.01
Coordinator (split graphs)	0.06 ± 0.01	0.12 ± 0.01	0.06 ± 0.01	0.15 ± 0.07	0.11 ± 0.01
Engine (instantiate subgraphs)	1.05 ± 0.13	0.16 ± 0.03	0.30 ± 0.08	0.12 ± 0.04	0.40 ± 0.10

Table 4.3: Inference graph setup times (in ms) in two sample scenarios, with one standard deviation.

Social Interaction inference in QS contributes more than 9796 SLoC. Both Beam and M-Lib help alleviate this complexity. Beam improves upon M-Lib by handling the complexity of implementing sensor drivers, disconnection tolerance, and optimizing resource usage, etc.

4.5.2.2 Overhead of Inference Graph Creation

We study the time taken by Beam to satisfy requests for a single Mic Occupancy inference, which in turn uses the PC Mic adapter. We consider two sample scenarios, 1) apps request for a local inference, and 2) apps request for a remote inference. In both cases, app 1 initiates a request first, followed by app 2, with the same coverage tag.

In both scenarios, the overhead of instantiating and maintaining the inference graph at end-points is minimal and dwarfed by the latency of transferring the request to the coordinator and receiving back the subgraphs.

Table 4.3 shows the overhead of graph creation for each of the scenarios. In both cases, the second request uses less time for graph creation at the coordinator, since much of the graph already exists when the second request arrives (e.g., module specifications are not re-read). Likewise, in both scenarios, time spent at the engine(s) in applying the subgraph is lower for the second request as compared to the first request. Further, it is lower in scenario 2 because the inference graph is split across two engines. Lastly, the coordinator performs a periodic re-evaluation based on the channel data rates and applies the proactive optimization

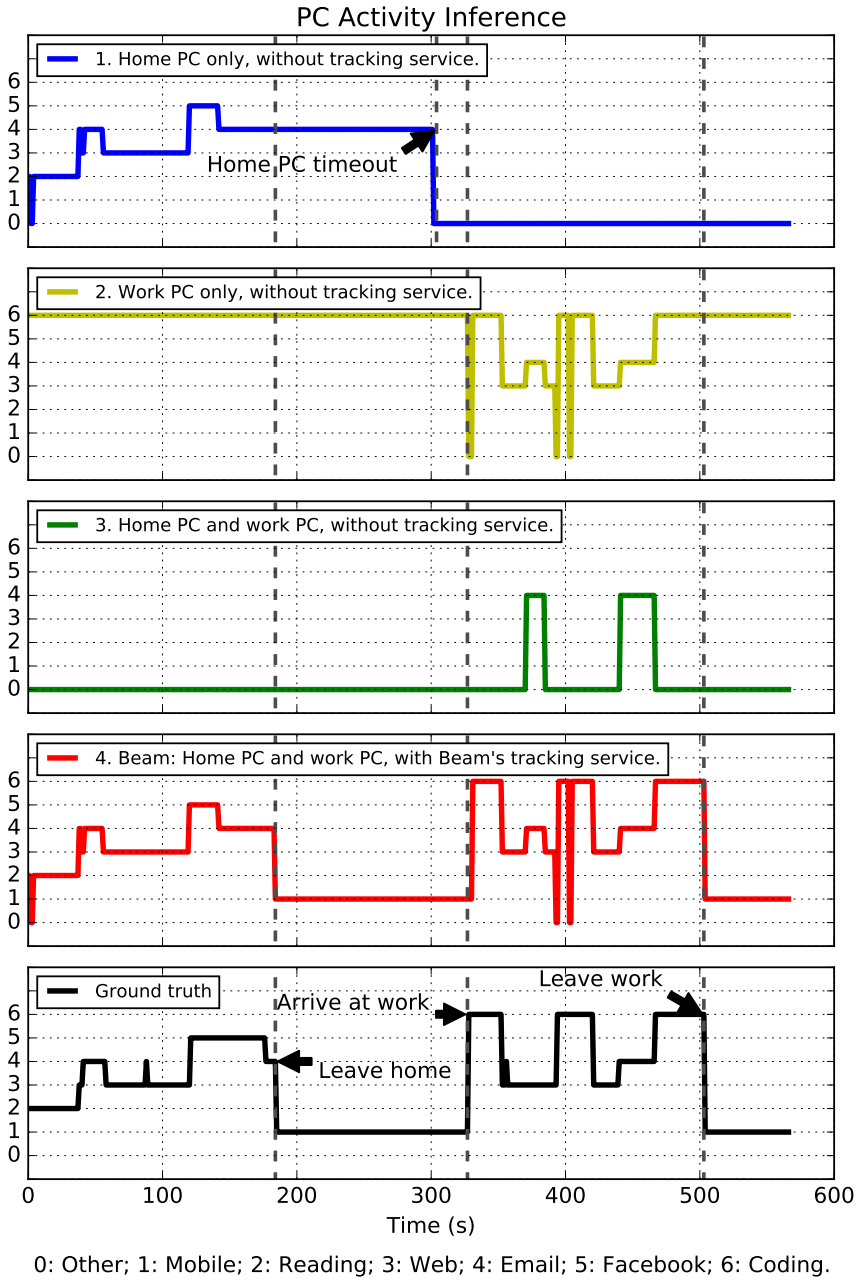


Figure 4.7: Beam's tracking service improves inference accuracy (measured against ground truth) significantly over other approaches all of which fail to select devices in the presence of user mobility.

discussed in Section 4.3.2. The time taken to perform the re-evaluation is minimal.

4.5.3 Device Selection

Unlike other approaches described in Section 4.5.1, the inference graph in Beam can select devices for apps even in heterogeneous environments with user mobility, resulting in increased inference accuracy. We demonstrate this using the PC Activity inference in the context of the QS app (inference graph in Figure 4.2).

We perform an experimental lab deployment with two locations - a lab which acts as 'home' and an office. Movements from home to office are used to simulate user commuting. We compute Beam's inference accuracy against manually-collected ground truth data from the deployment, and compare it to three other development approaches that may be used in the absence of a Beam-like tracking service. The first approach performs the PC Activity inference using only inputs from the home PC, while the second approach uses only inputs from the work PC. We assume that the home PC goes into sleep after a certain period of user inactivity, while the work PC remains on even after the user leaves. In the third approach, the inference is drawn using simultaneous inputs from both the home and work PCs. However, when the two inputs conflict, the output is set to 'Other'.

Figure 4.7 shows a comparison of inference accuracy for these different schemes over a ten minute interval of using the QS app. Inferring PC-based activities using only the home PC works accurately until the user leaves home, but deviates significantly from ground truth once the user has left. Similarly, using only the work PC can only accurately compute the PC-based activities of the user after the user arrives at work. On the other hand, using both the work and home PC without a tracking service often produces conflicting results, for instance, when home PC and work PC both generate PC Activity inferences during user commuting. Beam's tracking service correctly identifies the location of the user and triggers the inference graph to re-select appropriate devices, achieving inference accuracy 3× higher than the best performing scheme above. Using the tracking service, Beam's smartphone engine can also correctly indicate that the user is 'Mobile' while commuting. Table 4.4 summaries these accuracy improvements.

Although the above experiments are performed in a lab setting with a simulated commut-

Setup	Accuracy
Home PC only, without tracking service	29.68%
Work PC only, without tracking service	26.94%
Home PC and work PC, without tracking service	4.59%
Home PC and work PC, with Beam's tracking service	88.16%

Table 4.4: Accuracy of PC Activity Inference compared to ground truth (a summary of Figure 4.7).

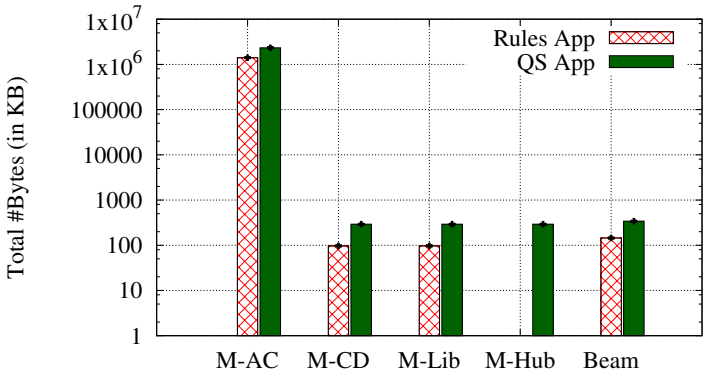


Figure 4.8: Total bytes transferred over the wide area for a 60 minute run of the Rules and QS apps using different approaches. Y-axis is in log scale.

ing scenario, having a longer commuting time will only reduce the accuracy of non-Beam approaches, since only Beam with the tracking service can infer user commuting and all other approaches will yield incorrect results. Finally, we expect to observe a similar accuracy improvement for other inferences that require handling of sensor coverage, e.g. the Social Interaction inference in the QS app.

4.5.4 Efficient Resource Usage

Next, we illustrate that Beam can match the resource usage of hand-optimized apps by partitioning the inference graph across devices. We also evaluate different optimization schemes used in Beam. Although we consider network usage to benchmark Beam in this dissertation, we expect similar optimizations can be performed on other resources such as CPU usage, latency, and energy.

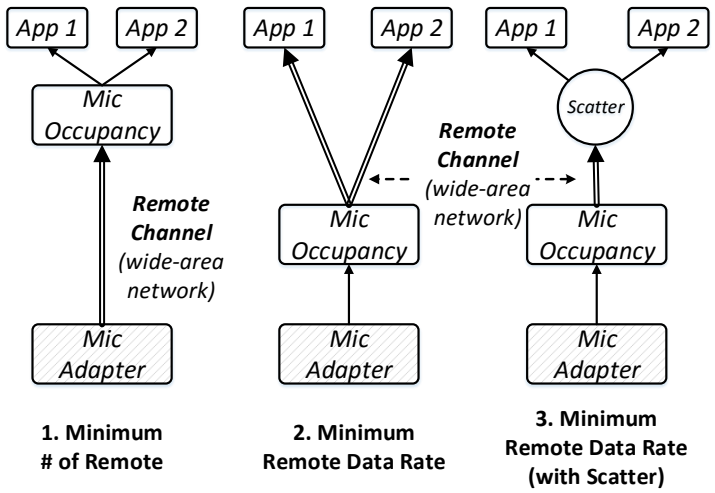


Figure 4.9: Sample configurations of the Mic Occupancy inference, with different optimization goals.

Graph partitioning: For the Rules and QS apps, we compare Beam's data transfer overhead (i.e., number of bytes transferred over the wide area) with that of different approaches (M-AC, M-CD, M-Lib, M-Hub). Figure 4.8 shows the total number of bytes transferred over the wide area in one hour, for the sample apps using different approaches. Medians and standard deviations across three runs are reported. M-AC incurs the largest overhead, because it transfers all sensor data from the device to a cloud VM for processing. On the other hand, the M-CD, M-Lib, and M-Hub approaches are optimized to perform most of their processing at the edges before transferring data to the cloud VM. Beam automatically partitions the inference graph using both reactive and proactive optimizations and comes close to matching the network transfer overhead incurred by M-CD; it incurs a slightly higher overhead for transferring *control messages* such as forwarding the app's request to the coordinator, receiving the part of the inference graph to instantiate, sending channel data rates to coordinator (for proactive optimization), acknowledgments, etc. Note that, when the M-Hub approach is used for the Rules app, there is no wide area IDU transfers because all required sensors are present locally at home.

Optimization schemes: Next, we evaluate the effect of different optimization schemes in Beam. We focus on a simple inference graph, where two apps running on cloud servers

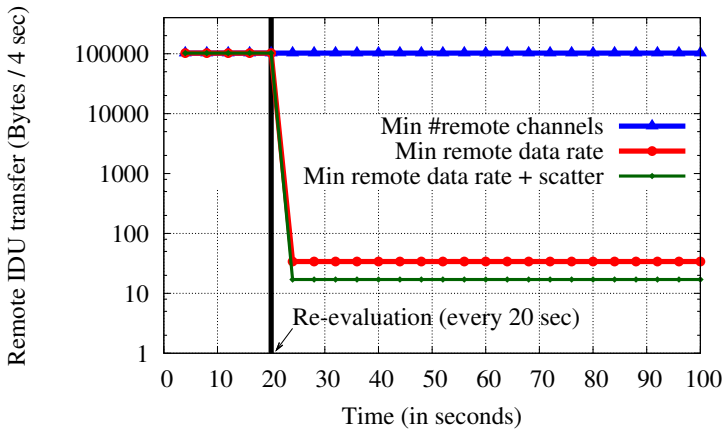


Figure 4.10: Network resource consumption over a 100 seconds interval for configurations in Figure 4.9. Y-axis is in log scale. IDUs are generated every 4 seconds.

subscribe to the Mic Occupancy inference. Figure 4.9 shows the three configurations that result from Beam optimizations in isolation and Figure 4.10 shows their network resource consumption over a 100-second interval. Beam's default reactive optimization (Figure 4.9 #1) minimizes the number of remote channels resulting in a large amount of microphone data being transmitted over the wide area. Beam's proactive optimization notices these large uploads and uses channel data rates to re-evaluate and re-partition the inference graph (Figure 4.10 at 20 s), thus moving the Mic Adapter closer to the edge (Figure 4.9 #2), and reducing wide area transfers significantly. Finally, enabling Beam's scatter node optimization (Figure 4.9 #3) halves the network overhead, for two consumer apps, compared with the proactive optimization without the scatter node.

4.5.5 Handling Disconnections

In this section we quantify the ability of the Beam inference graph to handle device disconnections. Remote channels in Beam buffer data at both the coordinator and at the writer endpoints to tolerate reader and writer disconnections. The size of these buffers and the writer's sending rate determine the time window for which disconnections are lossless, and can be sized as per the deployment scenario. Figure 4.11 shows a time series plot of the number of IDUs and data messages received by a reader over a 100 seconds interval. The

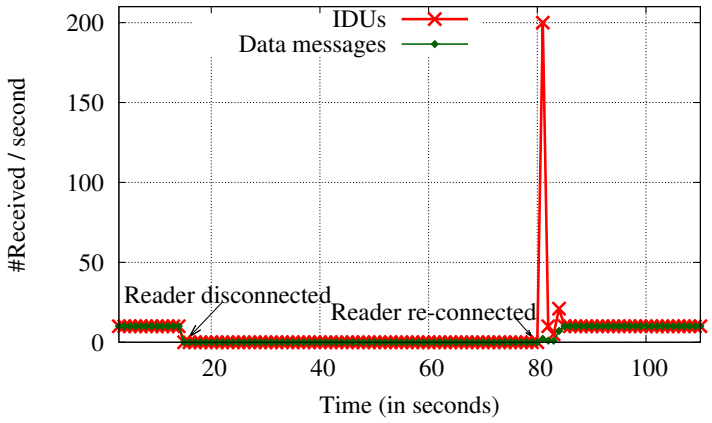


Figure 4.11: Remote channel time trace. Write rate is 10 values per second, and writer buffer and coordinator buffer are sized at 100 values each.

writer produces ten IDUs every second. Each IDU produced is pushed out in a separate data message until a reader disconnection at t=15s results in data buffering, first at the coordinator, and then at the writer. We constrain the channel buffers at the writer and coordinator ends to 100 IDUs each, thus supporting buffering of only 20 seconds worth of IDUs in this configuration, forcing the remaining IDUs to be dropped. When the reader reconnects at t=80 s, the 200 buffered IDUs are batched in a small number of data messages and delivered to the reader, showing Beam's support for tolerating device disconnections.

4.6 Discussion

We discuss potential improvements to Beam.

Error and error propagation: Beam currently supports typed errors such as probability distributions (e.g. mean and standard deviation), and error margin (e.g. center and radius). Although error propagation has been studied in the field of artificial intelligence (e.g. neural network [RHW88]), there is no prior work on error propagation in mobile context sensing. We are investigating techniques to enable inference module developers to implement customized error propagation functions for specific inferences, so that Beam can propagate the error from a module's inputs to its output.

Actuation and timeliness: Many in-home devices possess actuation capabilities, such as locks, switches, cameras, and thermostats. apps and inference modules in Beam may want to use such devices. If the inference graph for these apps is geo-distributed, timely propagation and delivery of such actuation commands to the devices becomes important and raises interesting questions of what is the *safe* thing to do if an actuation arrives "late".

Data archival and correlation mining: Prior work has shown that exploiting the correlation among inferences can effectively reduce sensing cost [Nat12]. While Beam modules do not currently store data either at the engines or the coordinator, apps and modules may use a temporal datastore, such as Bolt [GSP14], to make inferences durable. Storing and querying archived inference data will allow inference developers to perform correlation mining to improve inferences.

Data privacy: While we do not address privacy concerns in our work, we believe the use of inferences can enable better data privacy controls [CSR14]. For example, users may allow an app to access the occupancy inference (using a camera) instead of the raw image data used for drawing the inference. This prevents the leakage of private information by preventing other inferences that can be drawn using the raw data. Moreover, Beam's coverage tags allow the user to define fine-grained controls, for instance, allowing an app to access activity inference only for a certain user tag.

4.7 Summary

Context inference apps using connected sensing devices are difficult to develop today because they must incorporate all the data sensing and inference logic, even as devices move or are temporarily disconnected. We design and implement Beam, a framework and runtime for context inference apps using connected devices. Beam introduces the inference graph abstraction which is central to decoupling apps, inference algorithms, and devices. Beam uses the inference graph to address the challenges of device selection, efficient resource usage, and device disconnections. Using Beam, we develop two representative apps (Rules and QS), where we show up to 4.5× lower number of tasks and 12× lower source line of code in app

development effort, with negligible runtime overhead. Moreover, Beam results in up to 3× higher inference accuracy due to its ability to select devices in heterogeneous environments, and Beam's dynamic optimizations match hand-optimized apps for network resource usage.

CHAPTER 5

Context Awareness Toolkit

Based on the above findings on composing and running context inferences across multiple devices, we have created a suite of open-source toolkit apps to assist developers in different stages of building context inferences apps, including:

- The **Data Collector** that helps developers collect sensor data and ground truth labels from users as part of data collection campaigns.
- The **Inference Composer** that automatically composes and exports an inference pipeline from a collected dataset, including the training and tuning of machine learning models. It also supports manual configuration of an inference.
- The **Inference Executor** that takes an exported inference pipeline from Inference Composer and performs runtime optimizations of inference executions by partitioning a pipeline across multiple devices.

The above apps are open-source and available on Github [too].

5.1 Data Collector

Because most context inference apps use data-driven machine learning models to make decisions, the first task of building inferences is always to collect data and ground truth from a set of users, such as the user study seen in MiLift. The data collection campaigns in various situations have the same or very similar workflow, that is, collecting sensor data and saving them for model training. However, the exact configuration of data collections may vary in a number of aspects, such as the type of devices, type of sensors, sampling frequencies, etc.

Therefore for each new data collection campaign, developers often need to write a different data collection app and cannot re-use previous code. Moreover, there are very limited support from the mobile operating systems and existing solutions for collecting ground truth labels.

To address these problems, we propose Data Collector, an Android app for helping developers collect a set of sensor data from users, including the ground truth label.

5.1.1 Design

The basic workflow of Data Collector is described as follows:

- Data Collector provides a set of APIs for app developers to specify the set of sensors to be considered, including device types, sensor types, sensor configurations, as well as ground truth configurations.
- Target users then install Data Collector, which collects data based on the developer's configuration. The app runs on the target user's phone, continuously sampling sensor data and periodically querying ground truth labels (if necessary) until the desired amount of data has been collected.
- Finally, Data Collector outputs the data in a serialized or structured human-readable format, and developers can save the data for later use.

The schema used by Data Collector to represent sensor data and ground truth labels is shown in Figure 5.1. The main abstractions of a dataset include:

- `DataType` represents both the `DeviceType` (e.g. smartwatch, smartphone, Fitbit, etc.) and the `SensorType` (e.g. accelerometer, gyroscope, GPS location, etc.).
- `DataInstance` is a single instance of sensor data which contains a `timestamp` and an array of `value` representing the different channels of sensors.
- `LabelType` represents the type of a ground truth label collected from users. Each label could be a real number, an integer, a string (nominal value), or a sensor type.

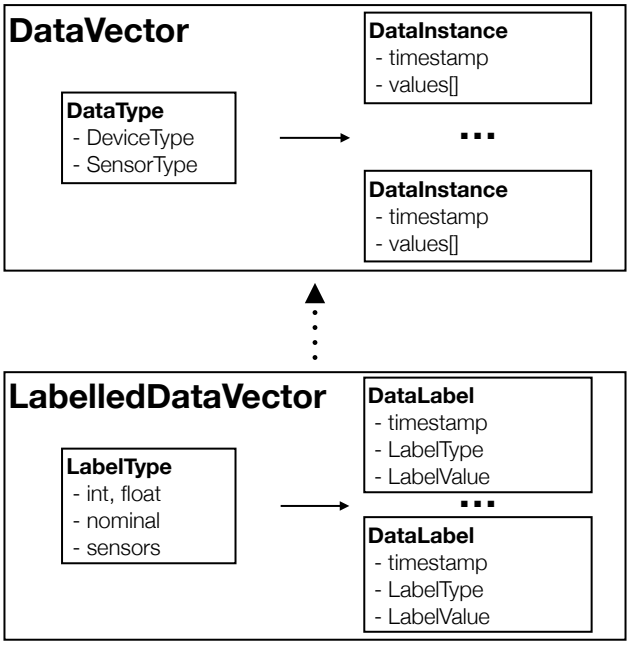


Figure 5.1: Schema of data representation used by Data Collector.

- Each `DataLabel` is an instance of a ground truth label, identified by a `LabelType` and the corresponding value `LabelValue`.
- Each `DataVector` is a basic entry in a dataset. It contains a hash map from a `DataType` to a list of `DataInstance`.
- Each `LabelledDataVector` inherits the `DataVector` class but also stores a list of ground truth labels using a hash map from a `LabelType` to a list of `DataLabel`.

5.1.2 Implementation

Data Collector is implemented as an Android app. It provides a `Data Collection Configurator` for developers to configure a data collection and then calls a background service `Data Collection Service` for sampling and saving sensor data as a `DataVector`. The ground truth label collection is implemented using the Android Alarm and BroadcastReceiver and periodically queries labels from users.

```
1 // Configure type of sensor data to collect
2 DataCollectionConfigurator configurator =
3     new DataCollectionConfigurator();
4 configurator.addSensorTypeToVector(
5     DeviceType.ANDROID_PHONE,
6     SensorType.ANDROID_ACCELEROMETER);
7 configurator.addSensorTypeToVector(
8     DeviceType.ANDROID_PHONE,
9     SensorType.ANDROID_GRAVITY);
10
11 // Configure label collection
12 LabelType labelType = new LabelType(
13     "ground_truth",
14     LabelDataType.NOMINAL);
15     labelType.setInterval(10);
16     labelType.addCandidateNominalValue("Activity A");
17     labelType.addCandidateNominalValue("Activity B");
18     labelType.addCandidateNominalValue("Activity C");
19
20 // Pass Configurator to DataCollectionService
21 DataCollectionService.configureDataCollection(
22     DeviceType.ANDROID_PHONE,
23     configurator.getDataVector());
24
25 // Start the data collection
26 DataCollectionService.startDataCollection();
27
28 // Stop the data collection and obtain the data as CSV
29 DataVector result = DataCollectionService.stopDataCollection();
30 result.dumpAsCSV("path_to_data.csv");
```

Listing 5.1: API example of the Data Collector.

Listing 5.1 shows an example of using Data Collector's APIs to configure a data collection. Developers invoke the configurator and specify the types of sensor data to be collected (line 2-9). They then request labels of nominal type (strings) to be collected and specify the possible candidate strings (line 12-18). Finally, developers use the `Data Collection Service` to start the data collection (line 26), and write the collected data to a CSV file after stopping the collection (line 29-30).

5.2 Inference Composer

Next, we describe the design and implementation of Inference Composer, a Python program that provides high-level APIs for training machine learning models from data generated

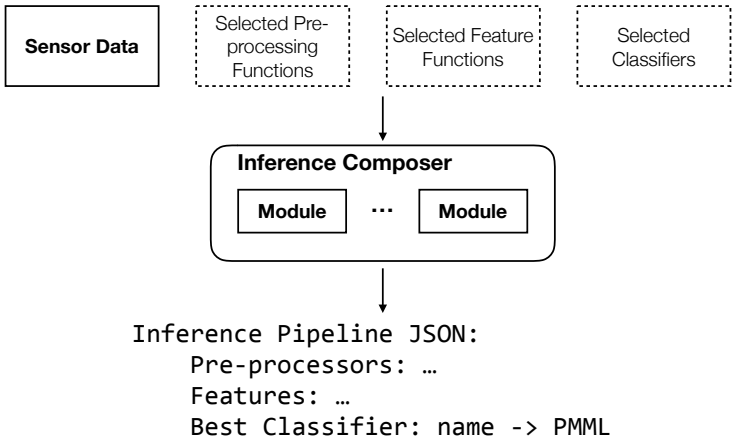


Figure 5.2: Workflow of Inference Composer.

by Data Collector. While Beam tackles the challenges of creating inference abstractions and leveraging heterogeneous devices at runtime, this app complements Beam by providing assistance on model training.

Compared with prior frameworks for mobile sensing and inference, such as Kobe [CLL11], Auditeur [NDA13a], and Senergy [KSB13], our Inference Composer app offers two new features. First, it eases the inference model training by offering an automatic training mode, where a set of default classifiers are trained on a dataset and the model with the best performance is returned to the developer. Second, it closes the gap between popular machine learning frameworks (e.g. scikit-learn [PVG11], Tensorflow [ten], etc.) and mobile operating systems such as Android. Inference Composer can export an inference pipeline trained by these frameworks (in Python) to a JSON file that can be parsed, initialized, and executed on Android using the Inference Executor app described in Section 5.3. In doing so it saves trained classification models in the form of Predictive Model Markup Language (PMML) [pmm] which can then be loaded by the Inference Executor.

5.2.1 Design

The workflow of Inference Composer is shown in Figure 5.2. *Machine learning experts* (i.e. inference developers) create Beam-style inference modules by implementing a `ModuleBase`

interface. Each inference module must implement two methods:

- `process()` takes a data vector, performs inference computation, and returns a data vector.
- `export()` saves the module itself in JSON format, including transforming scikit classifiers to PMML.

Inference Composer is shipped with a module library which includes default inference modules such as pre-processing functions, feature calculation functions, and classifiers.

App developers then feed sensor data into Inference Composer. We offer two modes of composing inferences:

Manual mode: App developers use the API to specify the sensors to be considered, pre-processing functions, feature functions, and classification models (given that these modules exist in the current module library). By supplying a dataset obtained from Data Collector, an inference pipeline can be trained based on the specifications and saved to a JSON file.

Automatic mode: App developers use the API to pick a specific inference and specify certain criteria, for example, a thresholding inference accuracy or simply requesting a classifier with the best accuracy. Based on the dataset, Inference Composer automatically selects the right classifier and feature algorithms (if required) from the current module library to meet the criteria, and returns the trained inference pipeline.

After an inference pipeline is created, it can be exported to a JSON file by invoking the `export()` method of each of its member inference module. An example of the JSON file is also shown in Figure 5.2.

5.2.2 Implementation

Inference Composer is implemented in Python and provides wrapper functions for both scikit-learn [PVG11] and Tensorflow [ten], enabling it to leverage both traditional machine learning models and deep neural networks. It provides a set of default inference modules

```
1 # Pre-processing
2 pre_processor = Preprocess.Preprocess(
3   _window_size=1,
4   _data_columns=data_columns,
5   _operators=[Preprocess.MOVING_AVG_SMOOTH]
6 )
7 processed_data = pre_processor.process(raw_data)
8
9 # Feature calculation
10 feature_calculator = Feature.Feature(
11   _window_size=1,
12   _data_columns=data_columns,
13   _features=features
14 )
15 feature_vector = feature_calculator.process(processed_data)
16
17 # Train default classifiers and show performance
18 classifiers = Classifier.Classifier(
19    _feature_mapper=feature_calculator.get_mapper(),
20     _model_path=MODEL_PATH,
21     _cross_validation=True,
22     _cv_fold=10
23 )
24 classifiers.add_default_classifiers()
25 classifiers.process(feature_vector)
26
27 # Export the trained pipeline to JSON,
28 # including automatically transforming model to PMML
29 export_inference([pre_processor, feature_calculator, classifiers])
```

Listing 5.2: API example of Inference Composer.

but also allows inference developers to add new modules using the `ModuleBase` interface. In order to export scikit classifiers into PMML, Inference Composer uses tools provided by the open-source jpmml project, including jpmml-model [jpmb] and sklearn2pmml [jpmc]. Listing 5.2 illustrates the use of Inference Composer APIs to create and export an inference pipeline. Developers configure pre-processing (line 2-7), feature calculation (line 10-15), and classifiers (line 17-25) using existing inference modules. They can export the entire inference as a JSON file after specifying each modules (line 29).

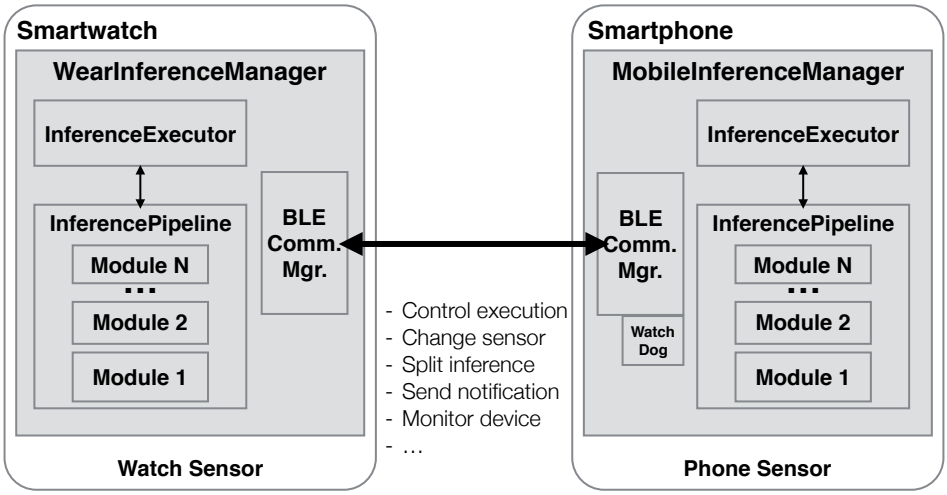


Figure 5.3: Architecture of Inference Executor.

5.3 Inference Executor

Inference Executor is an Android app for actualizing an inference pipeline generated by Inference Composer and optimizing its execution across devices. The runtime loads a saved inference pipeline from a JSON file, including loading classifiers from PMMLs, and instantiates the pipeline using Java implementations of inference modules available in a module library. The runtime partitions modules in a pipeline and considers different devices as targets for the execution. In this work, we consider the specific case of executing an inference between a smartwatch and a smartphone, similar to the scenario proposed in Section 3.4.

5.3.1 Design

Figure 5.3 shows the system architecture of two Inference Executor apps running on a smartphone and a smartwatch, respectively. An `Inference Manager` (mobile or wear) runs on each device and is responsible for coordinating and managing the execution between devices. On each device, `Inference Manager` invokes an `Inference Executor` to load an `Inference Pipeline` from a JSON file, and to instantiate required `Inference Module` based on loaded parameters as well as existing modules in the current library. `Inference Executor` also

serves as a runtime container for performing the actual sensing and inference workloads. Finally, a `Communication Manager` coordinates the communication between devices, including but not limited to controlling module placements, managing sensor sampling, sending notifications, etc. Specifically, a dedicated part called `Watch Dog` monitors other devices for connectivity and availability to support runtime optimizations.

In the case of watch-phone coordination, the `Mobile Inference Manager` running on a smartphone loads an `Inference Pipeline` and splits it into a phone part and a watch part. It then uses the `Communication Manager` to send the watch part to the `Wear Inference Manager`. Both managers then coordinates the execution of this inference.

5.3.2 Implementation

Inference Executor uses the Android Alarm and BroadcastReceiver for periodic inference execution. Inference pipelines are parsed using the JSON library of Java, and we use the jpmml-evaluator [jpma] open-source library to parse and evaluate the PMML classification models exported by Inference Composer. Finally, the communication between devices is achieved using Android's Wear API [anda], which opportunistically uses either Bluetooth LE of WiFi for data transmission.

Similar to Inference Composer, Inference Executor has a set of default inference modules in its Java module library. It also provides an interface for creating new modules.

To perform an inference in Inference Executor, app developers specify a pointer to the JSON pipeline file, the interval and duration of the current inference execution, and an optimization goal as one of the following:

- *Phone only:* Always perform the inference on a smartphone.
- *Watch only:* Always perform the inference on a smartwatch.
- *Maximize coverage:* Start the entire inference on a smartwatch and use a phone to monitor the connectivity and significant motions on the watch. Upon the detection of watch not moving or disconnected, move the entire inference to phone. As shown by

our experiments in Section 3.4.4.5, this approach maximizes the inference accuracy by using whichever device available at a time.

- *Minimize transmission:* Start with a random module partition across the two devices. Monitor the amount of data transferred between devices. If the data rate is greater than a certain threshold, re-partition the inference pipeline and restart the inference execution. Similar to Beam's default optimizer shown in Section 4.5.4., this approach minimizes resource usage by reducing the amount of remote data transfer.

CHAPTER 6

Conclusion

In this dissertation we have made three key research contributions towards improving the development and execution of context-aware inference apps on pervasive connected devices:

- By proposing MiLift as an example app of running context inferences on a non-smartphone wearable device, we prove that context inferences have the potential of leveraging the unique characteristics of heterogeneous devices for better inference accuracy and energy efficiency. MiLift achieves 90% classification accuracy for both cardio and weightlifting exercises while extending the watch battery life by up to 19 hours compared with prior approaches. It performs fully autonomous workout tracking and requires no manual input from users. MiLift is also used to motivate our following work, such as optimizing the inference accuracy and energy consumption of context inference apps, and the multi-device inference framework.

- We perform a set of optimizations for context inference apps that can be adopted by developers today. We have achieved comparable inference accuracy as traditional models and acceptable latency using deep learning without hand-picking features, up to 30× latency speed-up of deep learning tasks using mobile GPUs and up to 60% energy saving of off-loading inference tasks from the CPU to the DSP, as well as up to 37% accuracy improvement and up to 67% less energy consumption for context-aware apps from watch-phone coordinations.

- We design and implement Beam, a framework and runtime for context inference apps using connected devices. Beam fills the gap between raw sensor data and high-level contexts by introducing the inference graph abstraction which is central to decoupling

apps, inference algorithms, and devices. Beam uses the inference graph to address the challenges of device selection, efficient resource usage, and device disconnections. Beam helps reduce development tasks by up to 4.5× and source lines of code by up to 12×, while achieving 3× higher inference accuracy by handling environmental dynamics.

- We release a suite of open-source toolkit apps to assist developers in data collection, inference composition, and inference execution on Android.

References

[ABS05] Ernesto Arroyo, Leonardo Bonanni, and Ted Selker. "Waterbot: Exploring Feedback and Persuasive Techniques at the Sink." In *Proc. ACM CHI*, 2005.

[ABY14] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. "Rio: A System Solution for Sharing I/O between Mobile Systems." In *Proc. ACM MobiSys*, 2014.

[aeo] "Aeon Labs Z-Wave Smart Energy Switch." `http://aeotec.com/`.

[ama] "Amazon Home Automation Store." `http://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011`.

[anda] "Building Apps for Wearables." `https://developer.android.com/training/building-wearables.html`.

[andb] "Composite sensor type summary - Android Developer." `https://source.android.com/devices/sensors/sensor-types.html#composite_sensor_type_summary`.

[appa] "Apple - Apple Watch." `https://www.apple.com/watch/`.

[appb] "Core Motion Framework Reference - iOS Developer Library." `https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Reference/`.

[arm] "ARM big.LITTLE." `https://www.arm.com/products/processors/technologies/biglittleprocessing.php`.

[Arm10] Joe Armstrong. "Erlang." *CACM*, **53**:68–75, Sept 2010.

[asu] "ASUS ZenWatch 2." `https://www.asus.com/us/ZenWatch/ASUS_ZenWatch_2_WI501Q/`.

[atl] "Atlas Wristband." `https://www.atlaswearables.com/`.

[ban] "Guided Workout: Microsoft Band." `https://www.microsoft.com/microsoft-band/en-us/support/health-and-exercise/guided-workouts/`.

[BBG14] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. "Orleans: Distributed Virtual Actors for Programmability and Scalability." Technical Report MSR-TR-2014-41, March 2014.

[bcc11] "The US Market for Home Automation and Security Technologies." Technical report, BCC Research, IAS031B, 2011.

[BDV03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. "A neural probabilistic language model." *journal of machine learning research*, **3**(Feb):1137–1155, 2003.

[BGK11] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. "Orleans: Cloud Computing for Everyone." In *Proc. ACM SOCC*, 2011.

[BHS07] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B. Srivastava. "SensorWare: Programming Sensor Networks Beyond Code Update and Querying." *Pervasive Mob. Comput.*, **3**(4):386–412, August 2007.

[BJM13] A.J. B. Brush, Jaeyeon Jung, Ratul Mahajan, and Frank Martinez. "Digital Neighborhood Watch: Investigating the Sharing of Camera Data amongst Neighbors." In *Proc. ACM CSCW*, 2013.

[BKP14] Nipun Batra, Jack Kelly, Oliver Parson, Haimonti Dutta, William J. Knottenbelt, Alex Rogers, Amarjeet Singh, and Mani Srivastava. "NILMTK: An open source toolkit for non-intrusive load monitoring." In *Proc. ACM e-Energy*, 2014.

[BL16] Sourav Bhattacharya and Nicholas D Lane. "From smart to deep: Robust activity recognition on smartwatches using deep learning." In *Proc. IEEE PerCom Workshops*, 2016.

[BNJ11] Rajesh Krishna Balan, Khoa Xuan Nguyen, and Lingxiao Jiang. "Real-time Trip Information Service for a Large Taxi Fleet." In *Proc. 9th ACM MobiSys*, Bethesda, Maryland, USA, June 2011.

[Bot] Léon Bottou. "Stochastic Gradient Descent (version 2)." `http://leon.bottou.org/projects/sgd`.

[Bot10] Léon Bottou. "Large-scale machine learning with stochastic gradient descent." In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

[Bot12] Léon Bottou. "Stochastic Gradient Descent Tricks." In *Neural Networks: Tricks of the Trade*, pp. 421–436. Springer, 2012.

[BR15] Tony Beltramelli and Sebastian Risi. "Deep-Spying: Spying using Smartwatch and Deep Learning." *arXiv preprint arXiv:1512.05616*, 2015.

[CBV15] Meredith A Case, Holland A Burwick, Kevin G Volpp, and Mitesh S Patel. "Accuracy of smartphone applications and wearable devices for tracking physical activity data." *JAMA*, **313**(6):625–626, 2015.

[CCC07] Keng-Hao Chang, Mike Y. Chen, and John Canny. "Tracking Free-weight Exercises." In *Proceedings of the 9th International Conference on Ubiquitous Computing*, UbiComp '07, 2007.

[CDS05] Michael L Cotterman, Lynn A Darby, and William A Skelly. "Comparison of muscle force production using the Smith machine and free weights for bench press and squat exercises." *The Journal of Strength & Conditioning Research*, **19**(1):169–176, 2005.

[CH10] Aaron Carroll and Gernot Heiser. "An analysis of power consumption in a smartphone." In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ATC '10, pp. 21–21, 2010.

[CHS05] Edmund Cauza, Ursula Hanusch-Enserer, Barbara Strasser, Bernhard Ludvik, Sylvia Metz-Schimmerl, Giovanni Pacini, Oswald Wagner, Petra Georg, Rudolf Prager, Karam Kostner, et al. "The relative benefits of endurance and strength training on the metabolic factors and muscle function of people with type 2 diabetes mellitus." *Archives of physical medicine and rehabilitation*, **86**(8):1527–1533, 2005.

[CL11] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: a library for support vector machines." *ACM Transactions on Intelligent Systems and Technology (TIST)*, **2**(3):27, 2011.

[CLL11] David Chu, Nicholas D. Lane, Ted Tsung-Te Lai, Cong Pang, Xiangying Meng, Qing Guo, Fan Li, and Feng Zhao. "Balancing Energy, Latency and Accuracy for Mobile Sensor Data Classification." In *Proc. 9th ACM SenSys*, Seattle, Washington, USA, November 2011.

[crf] "CRF++: Yet Another CRF toolkit." `https://taku910.github.io/crfpp/`.

[CSG13] Heng-Tze Cheng, Feng-Tso Sun, Martin Griss, Paul Davis, Jianguo Li, and Di You. "Nuactiv: Recognizing unseen new activities using semantic attribute-based learning." In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pp. 361–374. ACM, 2013.

[CSR14] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. "ipShield: a framework for enforcing context-aware privacy." In *Proc. 11th USENIX NSDI*, April 2014.

[cud] "Nvidia CUDA." `http://www.nvidia.com/object/cuda_home_new.html`.

[CZW15] David Chu, Zengbin Zhang, Alec Wolman, and Nicholas Lane. "Prime: A Framework for Co-located Multi-device Apps." In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pp. 203–214, 2015.

[DM80] Steven B Davis and Paul Mermelstein. "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences." *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **28**(4):357–366, 1980.

[DMA12] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. "An operating system for the home." In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012.

[dro] "Dropcam - Super Simple Video Monitoring and Security." `https://www.dropcam.com/`.

[DSY15] Han Ding, Longfei Shangguan, Zheng Yang, Jinsong Han, Zimu Zhou, Panlong Yang, Wei Xi, and Jizhong Zhao. "FEMO: A Platform for Free-weight Exercise Monitoring with RFIDs." In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, 2015.

[DYN10] Kyle Dorman, Marjan Yahyanejad, Ani Nahapetian, Myung-kyung Suh, Majid Sarrafzadeh, William McCarthy, and William Kaiser. "Nutrition Monitor: A Food Purchase and Consumption Monitoring Mobile System." In *Mobile Computing, Applications, and Services*, pp. 1–11. Springer, 2010.

[EBS11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling." In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376. IEEE, 2011.

[ESK11] Emre Ertin, Nathan Stohs, Santosh Kumar, Andrew Raij, Mustafa al'Absi, and Siddharth Shah. "AutoSense: unobtrusively wearable sensor suite for inferring the onset, causality, and consequences of stress in the field." In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pp. 274–287. ACM, 2011.

[Eva11] Dave Evans. "The Internet of Things: How the next evolution of the Internet is changing everything." *CISCO white paper*, 2011.

[Faw06] Tom Fawcett. "An introduction to ROC analysis." *Pattern recognition letters*, **27**(8):861–874, 2006.

[FCH08] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. "LIBLINEAR: A library for large linear classification." *The Journal of Machine Learning Research*, **9**:1871–1874, 2008.

[FFO12] Jon Froehlich, Leah Findlater, Marilyn Ostergren, Solai Ramanathan, Josh Peterson, Inness Wragg, Eric Larson, Fabia Fu, Mazhengmin Bai, Shwetak Patel, and James A. Landay. "The Design and Evaluation of Prototype Eco-feedback Displays for Fixture-level Water Usage Data." In *Proc. ACM CHI*, 2012.

[fit] "Fitbit." `https://www.fitbit.com/`.

[FN93] Ken-ichi Funahashi and Yuichi Nakamura. "Approximation of dynamical systems by continuous time recurrent neural networks." *Neural networks*, **6**(6):801–806, 1993.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning." Book in preparation for MIT Press, 2016.

[GE03] Isabelle Guyon and André Elisseeff. "An introduction to variable and feature selection." *Journal of machine learning research*, **3**(Mar):1157–1182, 2003.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. "Macro-programming wireless sensor networks using Kairos." In *Distributed Computing in Sensor Systems*, pp. 126–140. Springer, 2005.

[GLR14] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. "DSP.Ear: Leveraging Co-processor Support for Continuous Audio Sensing on Smartphones." In *Proc. ACM SenSys*, 2014.

[GLR16] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. "LEO: Scheduling Sensor Inference Algorithms across Heterogeneous Mobile Processors and Network Resources." In *Proc. 2016 ACM MobiCom*, 2016.

[gooa] "Google Awareness API." `https://developers.google.com/awareness/`.

[goob] "Google Cloud Dataflow." `https://cloud.google.com/dataflow/`.

[gooc] "Google Fit." `https://developers.google.com/fit/`.

[GSP14] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. "Bolt: Data management for connected homes." In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014.

[GTV14] Juan M García-Gómez, Isabel de la Torre-Díez, Javier Vicente, Montserrat Robles, Miguel López-Coronado, and Joel J Rodrigues. "Analysis of mobile health applications for a broad spectrum of consumers: A user experience approach." *Health informatics journal*, **20**(1):74–84, 2014.

[gym] "Gymwolf." `http://www.gymwolf.com/`.

[Har92] G.W. Hart. "Nonintrusive appliance load monitoring." *Proceedings of the IEEE*, 1992.

[HBC15] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B. Nightingale. "WearDrive: Fast and Energy-Efficient Storage for Wearables." In *Proc. USENIX ATC*, 2015.

[HCL08] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S Sathiya Keerthi, and Sellamanickam Sundararajan. "A dual coordinate descent method for large-scale linear SVM." In *Proceedings of the 25th international conference on Machine learning*, pp. 408–415. ACM, 2008.

[hea] "Apple HealthKit." `https://developer.apple.com/healthkit/`.

[hex] "Qualcomm Hexagon DSP." `https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor`.

[HFS08] Tâm Huynh, Mario Fritz, and Bernt Schiele. "Discovery of activity patterns using topic models." In *Proceedings of the 10th international conference on Ubiquitous computing*, UbiComp '08, pp. 10–19. ACM, 2008.

[HHP16] Nils Y Hammerla, Shane Halloran, and Thomas Ploetz. "Deep, Convolutional, and Recurrent Models for Human Activity Recognition using Wearables." *arXiv preprint arXiv:1604.08880*, 2016.

[HNT] Samuli Hemminki, Petteri Nurmi, and Sasu Tarkoma. "Accelerometer-based Transportation Mode Detection on Smartphones." In *Proc. of ACM SenSys 2013*.

[hom] "HomeSeer." `http://homeseer.com/`.

[HXZ13] Tian Hao, Guoliang Xing, and Gang Zhou. "iSleep: Unobtrusive Sleep Quality Monitoring Using Smartphones." In *Proc 11th ACM SenSys*, 2013.

[iBe] "iOS: Understanding iBeacon." `http://support.apple.com/kb/HT6048`.

[IBY07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed data-parallel programs from sequential building blocks." In *Proc. EuroSys*, Lisboa, Portugal, March 2007.

[ift] "IFTTT: connect the apps you love." `https://ifttt.com/`.

[jah] "jahmm: An implementation of Hidden Markov Models in Java." `https://code.google.com/p/jahmm/`.

[JdT95] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. "Rover: A Toolkit for Mobile Information Access." In *Proc. 15th ACM SOSP*, Copper Mountain, CO, December 1995.

[jef] "JEFIT Workout Tracker." `https://www.jefit.com/`.

[JLY12] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. "SymPhoney: A Coordinated Sensing Flow Execution Engine for Concurrent Mobile Sensing Applications." In *Proc. 10th ACM SenSys*, Toronto, Ontario, Canada, November 2012.

[jpma] "jpmml-evaluator." `https://github.com/jpmml/jpmml-evaluator`.

[jpmb] "jpmml-model." `https://github.com/jpmml/jpmml-model`.

[jpmc] "sklearn2pmml." `https://github.com/jpmml/sklearn2pmml`.

[JSD14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding." *arXiv preprint arXiv:1408.5093*, 2014.

[jso] "Json.NET." `http://www.newtonsoft.com/json`.

[KAB05] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. "Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea." In *Proc. 3rd ACM SenSys*, San Diego, California, USA, November 2005.

[Kar15] Andrej Karpathy. "The Unreasonable Effectiveness of Recurrent Neural Networks." 2015.

[KJ97] Ron Kohavi and George H John. "Wrappers for feature subset selection." *Artificial intelligence*, **97**(1):273–324, 1997.

[KKE10] Donnie H. Kim, Younghun Kim, Deborah Estrin, and Mani B. Srivastava. "SensLoc: Sensing Everyday Places and Paths Using Less Energy." In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pp. 43–56, 2010.

[KLJ08] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. "SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments." In *Proc. ACM MobiSys*, 2008.

[KLL10] Adil Mehmood Khan, Young-Koo Lee, Sungyoung Y Lee, and Tae-Seong Kim. "A triaxial accelerometer-based physical-activity recognition via augmented-signal features and a hierarchical recognizer." *Information Technology in Biomedicine, IEEE Transactions on*, **14**(5):1166–1172, 2010.

[KMC00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click modular router." *ACM Transactions on Computer Systems*, **18**(3):263–297, August 2000.

[KMH13] Matthias Kranz, Andreas Möller, Nils Hammerla, Stefan Diewald, Thomas Plötz, Patrick Olivier, and Luis Roalter. "The mobile fitness coach: Towards individualized skill assessment using personalized mobile devices." *Pervasive and Mobile Computing*, **9**(2):203–215, 2013.

[KSB13] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. "The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing." In *Proc. ACM OOPSLA*, Indianapolis, Indiana, USA, November 2013.

[LAH06] Liqian Luo, Tarek F Abdelzaher, Tian He, and John A Stankovic. "Envirosuite: An environmentally immersive programming framework for sensor networks." *ACM Transactions on Embedded Computing Systems (TECS)*, **5**(3):543–576, 2006.

[LBG16] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices." In *Proc. 15th ACM/IEEE IPSN*, 2016.

[Lee] Benjamin N Lee. "librf: C++ random forests library." `http://mtv.ece.ucsb.edu/benlee/librf.html`.

[LFR12] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T Chittaranjan, Andrew T Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. "StressSense: Detecting stress in unconstrained acoustic environments using smartphones." In *Proc. ACM UbiComp*, 2012.

[LGM15] Nicholas D. Lane, Petko Georgiev, Cecilia Mascolo, and Ying Gao. "ZOE: A Cloud-less Dialog-enabled Continuous Sensing Wearable Exploiting Heterogeneous Computation." In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pp. 273–286, 2015.

[LGQ15] Nicholas D Lane, Petko Georgiev, and Lorena Qendro. "DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning." In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 283–294. ACM, 2015.

[lgw] "LG G Watch R." `http://www.lg.com/us/smart-watches/lg-W110-lg-watch-r`.

[LHL14] Farley Lai, Syed Shabih Hasan, Austin Laugesen, and Octav Chipara. "CSense: A Stream-processing Toolkit for Robust and High-rate Mobile Sensing Applications." In *Proc. 13th ACM/IEEE IPSN*, 2014.

[LKL15] Luyang Liu, Cagdas Karatas, Hongyu Li, Sheng Tan, Marco Gruteser, Jie Yang, Yingying Chen, and Richard P. Martin. "Toward Detection of Unsafe Driving with Wearables." In *Proc. of ACM WearSys*, 2015.

[LLL13] Robert LiKamWa, Yunxin Liu, Nicholas D Lane, and Lin Zhong. "Moodscope: Building a mood sensor from smartphone usage patterns." In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013.

[LLN11] Sungyoung Lee, Hung Xuan Le, Hung Quoc Ngo, Hyoung Il Kim, Manhyung Han, Young-Koo Lee, et al. "Semi-Markov conditional random fields for accelerometer-based activity recognition." *Applied Intelligence*, **35**(2):226–241, 2011.

[LLZ09] John Langford, Lihong Li, and Tong Zhang. "Sparse Online Learning via Truncated Gradient." *Journal of Machine Learning Research*, **10**(777-801):65, 2009.

[LRW05] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. "Design and implementation of a single system image operating system for ad hoc networks." In *Proc. ACM MobiSys*, 2005.

[LWL12] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. "Reflex: using low-power processors in smartphones without knowing them." In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12. ACM, March 2012.

[LWZ12] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. "Supporting distributed execution of smartphone workloads on loosely coupled heterogeneous processors." In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '12, October 2012.

[LWZ14] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. "K2: A Mobile Operating System for Heterogeneous Coherence Domains." In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 285–300, New York, NY, USA, 2014. ACM.

[mapa] "Map My Fitness." `http://www.mapmyfitness.com/`.

[mapb] "MapMyRun." `http://www.mapmyrun.com/`.

[MBM08] Dan Morris, A.J. Bernheim Brush, and Brian R. Meyers. "SuperBreak: Using Interactivity to Enhance Ergonomic Typing Breaks." In *Proc. ACM CHI*, 2008.

[mem] "Memkite: Deep Learning for iOS." `http://memkite.com/`.

[Mer76] Paul Mermelstein. "Distance measures for speech recognition, psychological and instrumental." *Pattern recognition and artificial intelligence*, **116**:374–388, 1976.

[Met78] Charles E Metz. "Basic principles of ROC analysis." In *Seminars in nuclear medicine*, volume 8, pp. 283–298. Elsevier, 1978.

[MF94] Steven T McCaw and Jeffrey J Friday. "A Comparison of Muscle Activity Between a Free Weight and Machine Bench Press." *The Journal of Strength & Conditioning Research*, **8**(4):259–264, 1994.

[MIC14] Gloria Mark, Shamsi T. Iqbal, Mary Czerwinski, and Paul Johns. "Bored Mondays and Focused Afternoons: The Rhythm of Attention and Online Activity in the Workplace." In *Proc. 32nd ACM CHI*, April 2014.

[mil] "Workout Tracking, NESL Github." `https://github.com/nesl/WorkoutTracking`.

[MKB10] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. "Recurrent neural network based language model." In *Interspeech*, volume 2, p. 3, 2010.

[MLN15] Frank Mokaya, Roland Lucas, Hae Young Noh, and Pei Zhang. "Myovibe: Vibration based wearable muscle activation detection in high mobility exercises." In *Proc. ACM UbiComp*, 2015.

[MLN16] Frank Mokaya, Roland Lucas, Hae Young Noh, and Pei Zhang. "Burnout: A Wearable System for Unobtrusive Skeletal Muscle Fatigue Estimation." In *Proc. 15th ACM/IEEE IPSN*, 2016.

[mota] "Moto 360." `http://www.motorola.com/us/products/moto-360`.

[motb] "Moto 360 Sport." `https://www.motorola.com/us/products/moto-360-sport`.

[mov] "Moves app." `https://www.moves-app.com/`.

[Moz89] Michael C Mozer. "A focused back-propagation algorithm for temporal pattern recognition." *Complex systems*, **3**(4):349–381, 1989.

[MP96] David Mosberger and Larry Peterson. "Making Paths Explicit in the Scout Operating System." In *Proc. 2nd USENIX OSDI*, Seattle, WA, October 1996.

[MPA14] Bobak Jack Mortazavi, Mohammad Pourhomayoun, Gabriel Alsheikh, Nabil Alshurafa, Sunghoon Ivan Lee, and Majid Sarrafzadeh. "Determining the Single Best Axis for Exercise Repetition Recognition and Counting on SmartWatches." In *Proc. IEEE BSN*, 2014.

[mrf] "mRandomForest - NESL Github." `https://github.com/chenguangshen/mRandomForest`.

[msb] "Microsoft Band." `http://www.microsoft.com/microsoft-band/en-us`.

[MSG14] Dan Morris, T. Scott Saponas, Andrew Guillory, and Ilya Kelner. "RecoFit: Using a Wearable Sensor to Find, Recognize, and Count Repetitive Exercises." In *Proc. 32nd ACM CHI*, 2014.

[msh] "Microsoft Health." `https://www.microsoft.com/microsoft-health/`.

[MSS06] Uwe Maurer, Asim Smailagic, Daniel P Siewiorek, and Michael Deisher. "Activity recognition and monitoring using multiple sensors on different body positions." In *Proc. IEEE BSN*, 2006.

[msv] "mSVM - NESL Github." `https://github.com/chenguangshen/mSVM`.

[MVS15] Verena Majuntke, Sebastian VanSyckel, Dominik Schafer, Christian Krupitzer, Gregor Schiele, and Christian Becker. "COMITY: coordinated application adaptation in multi-platform pervasive systems." In *Proc. of IEEE PerCom*, 2015.

[MWM06] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. "Flask: A language for data-driven sensor network programs." *Harvard Univ., Tech. Rep. TR-13-06*, 2006.

[Nat12] Suman Nath. "ACE: Exploiting Correlation for Energy-efficient and Continuous Context Sensing." In *Proc. ACM MobiSys*, 2012.

[NDA13a] Shahriar Nirjon, Robert F. Dickerson, Philip Asare, Qiang Li, Dezhi Hong, John A. Stankovic, Pan Hu, Guobin Shen, and Xiaofan Jiang. "Auditeur: A Mobile-cloud Service Platform for Acoustic Event Detection on Smartphones." In *Proc. 11th ACM MobiSys*, Taipei, Taiwan, June 2013.

[NDA13b] Hyduke Noshadi, Foad Dabiri, Shaun Ahmadian, Navid Amini, and Majid Sarrafzadeh. "HERMES: Mobile system for instability analysis and balance assessment." *ACM Trans. Embed. Comput. Syst.*, **12**(1s):57:1–57:24, March 2013.

[NDH10] Ehsan Nazerfard, Barnan Das, Lawrence B Holder, and Diane J Cook. "Conditional random fields for activity recognition in smart environments." In *Proceedings of the 1st ACM International Health Informatics Symposium*, pp. 282–286. ACM, 2010.

[nes] "Nest." `http://www.nest.com/`.

[NGG15] Shahriar Nirjon, Jeremy Gummeson, Dan Gelb, and Kyu-Han Kim. "TypingRing: A Wearable Ring Platform for Text Input." In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, 2015.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. "The Regiment Macroprogramming System." In *Proc. 6th ACM/IEEE IPSN*, 2007.

[NVI] NVIDIA. "Tegra." `http://goo.gl/zmlZj`.

[OD08] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[OH05] Kunle Olukotun and Lance Hammond. "The future of microprocessors." *Queue*, **3**(7):26–29, 2005.

[opta] "Optimized App." `http://optimized-app.com/`.

[optb] "OptiTrack Prime 13." `https://www.optitrack.com/products/prime-13/`.

[Pea89] Barak A Pearlmutter. "Learning state space trajectories in recurrent neural networks." *Neural Computation*, **1**(2):263–269, 1989.

[PFB00] Michael L Pollock, Barry A Franklin, Gary J Balady, Bernard L Chaitman, Jerome L Fleg, Barbara Fletcher, Marian Limacher, Ileana L Piña, Richard A Stein, Mark Williams, et al. "Resistance exercise in individuals with and without cardiovascular disease benefits, rationale, safety, and prescription an advisory from the committee on exercise, rehabilitation, and prevention, council on clinical cardiology, American Heart Association." *Circulation*, **101**(7):828–833, 2000.

[PGR08] Kevin Patrick, William G Griswold, Fred Raab, and Stephen S Intille. "Health and the mobile phone." *American journal of preventive medicine*, **35**(2):177–181, 2008.

[PHK13] Igor Pernek, Karin Anna Hummel, and Peter Kokol. "Exercise repetition detection for resistance training based on smartphones." *Personal and ubiquitous computing*, **17**(4):771–782, 2013.

[Pin87] Fernando J Pineda. "Generalization of back-propagation to recurrent neural networks." *Physical review letters*, **59**(19):2229, 1987.

[PLL11] Bodhi Priyantha, Dimitrios Lymberopoulos, and Jie Liu. "Littlerock: Enabling energy-efficient continuous sensing on mobile phones." *Pervasive Computing, IEEE*, **10**(2):12–15, 2011.

[pmm] "PMML Wikipedia." `https://en.wikipedia.org/wiki/Predictive_Mode_Markup_Language`.

[poc] "Step tracking made easy with the M7." `http://blog.runkeeper.com/326/step-tracking-made-easier-with-the-m7/`.

[PPC12] Jun-geun Park, Ami Patel, Dorothy Curtis, Seth Teller, and Jonathan Ledlie. "Online pose classification and walking speed estimation using handheld devices." In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pp. 113–122, 2012.

[PVG11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, **12**:2825–2830, 2011.

[quaa] "Quantified Self." `http://quantifiedself.com/`.

[QUAb] QUALCOMM. "Snapdragon 800 MDP Mobile Development Platform." `http://goo.gl/3UYXI8`.

[Qui86] J. Ross Quinlan. "Induction of decision trees." *Machine learning*, **1**(1):81–106, 1986.

[RAP11] Md. Mahbubur Rahman, Amin Ahsan Ali, Kurt Plarre, Mustafa al'Absi, Emre Ertin, and Santosh Kumar. "mConverse: Inferring Conversation Episodes from Respiratory Measurements Collected in the Field." In *Proc. 2nd Wireless Health*, New York, NY, USA, 2011. ACM.

[ren] "Renderscript." `https://developer.android.com/guide/topics/renderscript/compute.html`.

[rev] "Revolv." `http://revolv.com/`.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning internal representations by error propagation." Technical report, DTIC Document, 1985.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors." *Cognitive modeling*, **5**, 1988.

[Rit84] Dennis M Ritchie. "A stream input-output system." In *AT&T Bell Laboratories Technical Journal*, 1984.

[RLC12] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. "Computational sprinting." In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12. IEEE, 2012.

[RMB10] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. "Using mobile phones to determine transportation modes." *ACM Transactions on Sensor Networks (TOSN)*, 2010.

[RMM10] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth, and Andrius Aucinas. "EmotionSense: A Mobile Phones Based Adaptive Platform for Experimental Social Psychology Research." In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, pp. 281–290. ACM, 2010.

[RPG07] H. Ramamurthy, B. S. Prabhu, R. Gadh, and AM. Madni. "Wireless Industrial Monitoring and Control Using a Smart Sensor Platform." *Sensors Journal, IEEE*, **7**(5):611–618, May 2007.

[RPK12] Moo-Ryong Ra, Bodhi Priyantha, Aman Kansal, and Jie Liu. "Improving energy efficiency of personal sensing applications with heterogeneous multi-processors." In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pp. 1–10. ACM, 2012.

[RRP14] James A Ross, David A Richie, Song J Park, Dale R Shires, and Lori L Pollock. "A case study of OpenCL on an Android mobile GPU." In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pp. 1–6. IEEE, 2014.

[RSD10] Sasank Reddy, Katie Shilton, Gleb Denisov, Christian Cenizal, Deborah Estrin, and Mani Srivastava. "Biketastic: sensing and mapping for better biking." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pp. 1817–1820. ACM, 2010.

[run] "RunKeeper." `https://runkeeper.com/`.

[SBS15] Muhammad Shoaib, Stephan Bosch, Hans Scholten, Paul JM Havinga, and Ozlem Durmaz Incel. "Towards detection of bad habits by fusing smartphone and smartwatch sensors." In *Proc. IEEE PerCom Workshops*, 2015.

[SBV11] Christian Seeger, Alejandro Buchmann, and Kristof Van Laerhoven. "myHealthAssistant: a phone-based body sensor network that captures the wearer's

exercises throughout the day." In *Proceedings of the 6th International Conference on Body Area Networks*, pp. 1–7. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[SCC14] Chenguang Shen, Haksoo Choi, Supriyo Chakraborty, and Mani Srivastava. "Towards a rich sensing stack for IoT devices." In *Proc. IEEE/ACM ICCAD*, San Jose, CA, 2014.

[SCR] Chenguang Shen, Supriyo Chakraborty, Kasturi Rangan Raghavan, Haksoo Choi, and Mani B. Srivastava. "Exploiting processor heterogeneity for energy efficient context inference on mobile phones." In *Proc. ACM HotPower 2013*.

[sho] "ShopKick Shopping App." `http://shopkick.com/`.

[sig] "ASP.NET SignalR." `http://signalr.net/`.

[sim] "Simplicam Home Surveillance Camera." `https://www.simplicam.com/`.

[SKB13] Rayman Preet Singh, S. Keshav, and Tim Brecht. "A cloud-based consumer-centric architecture for energy data analytics." In *Proc. ACM e-Energy*, 2013.

[SM06] Charles Sutton and Andrew McCallum. "An introduction to conditional random fields for relational learning." *Introduction to statistical relational learning*, pp. 93–128, 2006.

[sma] "SmartThings." `http://www.smartthings.com/`.

[sna] "Qualcomm Snapdragon." `https://www.qualcomm.com/products/snapdragon/processors`.

[SSP15a] Rayman Preet Singh, Chenguang Shen, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. "A Case for Ending Monolithic Apps for Connected Devices." In *Proc. USENIX HotOS XV*, 2015.

[SSP15b] Rayman Preet Singh, Chenguang Shen, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. "Its Time to End Monolithic Apps for Connected Devices." *USENIX ;login:*, **40**(5), 2015.

[SSP16] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. "Beam: Ending Monolithic Applications for Connected Devices." In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[str] "Strava." `https://www.strava.com/`.

[tan] "Project Tango." `https://get.google.com/tango/`.

[ten] "TensorFlow." `https://www.tensorflow.org/`.

[the] "Theano." `http://deeplearning.net/software/theano/`.

[TIa] TI. "OMAP." `http://processors.wiki.ti.com/index.php/OMAP`.

[TIb] TI. "Pandaboard." http://goo.gl/ujdiL.

[tid] "TIDIGITS Dataset." https://catalog.ldc.upenn.edu/ldc93s10.

[TKN10] Arsalan Tavakoli, Aman Kansal, and Suman Nath. "On-line Sensing Task Optimization for Shared Sensors." In *Proc. 9th ACM/IEEE IPSN*, 2010.

[too] "ContextAwarenessToolkit, NESL Github." https://github.com/nesl/ContextAwarenessToolkit.

[tor] "Torch." http://torch.ch/.

[Tsc15] Fabian Tschopp. "Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems." *arXiv preprint arXiv:1509.03371*, 2015.

[UMP14] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. "Practical Trigger-action Programming in the Smart Home." In *Proc. ACM CHI*, 2014.

[vim] "VimoFit." http://www.vimofit.com/.

[VKP06] Max Van Kleek, Kai Kunze, Kurt Partridge, et al. "OPF: a distributed context-sensing framework for ubiquitous computing environments." In *Ubiquitous Computing Systems*, pp. 82–97. Springer, 2006.

[VOI] VOICEBOX. "Speech Processing Toolbox for MATLAB." http://goo.gl/wakDY.

[VSM15] S Vigneshwaran, Sougata Sen, Archan Misra, Satyadip Chakraborti, and Rajesh Krishna Balan. "Using infrastructure-provided context filters for efficient fine-grained activity sensing." In *Proc. IEEE PerCom*, 2015.

[vul] "Vulkan." https://developer.nvidia.com/Vulkan.

[VVL07] Douglas L Vail, Manuela M Veloso, and John D Lafferty. "Conditional random fields for activity recognition." In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, p. 235. ACM, 2007.

[wat] "WatchKit - Apple Developers." https://developer.apple.com/watchkit/.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." In *Proc. 18th ACM SOSP*, Banff, Canada, October 2001.

[WCB11] Danny Wyatt, Tanzeem Choudhury, Jeff Bilmes, and James A. Kitts. "Inferring Colocation and Conversation Networks from Privacy-sensitive Audio with Implications for Computational Social Science." *ACM Trans. Intell. Syst. Technol.*, **2**(1), January 2011.

[WCC14] Rui Wang, Fanglin Chen, Zhenyu Chen, Tianxing Li, Gabriella Harari, Stefanie Tignor, Xia Zhou, Dror Ben-Zeev, and Andrew T. Campbell. "StudentLife: Assessing Behavioral Trends, Mental Well-being and Academic Performance of College Students using Smartphones." In *Proc. ACM Ubicomp*, 2014.

[Wer88] Paul J Werbos. "Generalization of backpropagation with application to a recurrent gas market model." *Neural Networks*, **1**(4):339–356, 1988.

[wor] "Workout Labs." `http://workoutlabs.com/`.

[WZ89] Ronald J Williams and David Zipser. "A learning algorithm for continually running fully recurrent neural networks." *Neural computation*, **1**(2):270–280, 1989.

[WZL06] Kamin Whitehouse, Feng Zhao, and Jie Liu. "Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data." In *Proc. EWSN*, 2006.

[xiv] "xively by LogMein." `https://xively.com/`.

[XLL13] Chenren Xu, Sugang Li, Gang Liu, Yanyong Zhang, Emiliano Miluzzo, Yih-Farn Chen, Jun Li, and Bernhard Firner. "Crowd++: Unsupervised Speaker Count with Smartphones." In *Proc. ACM UbiComp*, 2013.

[XSW11] James Y Xu, Yuwen Sun, Zhao Wang, William J Kaiser, and Greg J Pottie. "Context guided and personalized activity classification system." In *Proceedings of the 2nd Conference on Wireless Health*, p. 12. ACM, 2011.

[Yan] Christopher J.C. Burges Yann LeCun, Corinna Cortes. "THE MNIST DATABASE of handwritten digits." `http://yann.lecun.com/exdb/mnist/`.

[YTS14] Longqi Yang, Kevin Ting, and Mani B Srivastava. "Inferring occupancy from opportunistically available sensor data." In *Pervasive Computing and Communications (PerCom), 2014 IEEE International Conference on*, pp. 60–68. IEEE, 2014.

[ZMN10] Shumei Zhang, Paul McCullagh, Chris Nugent, and Huiru Zheng. "Activity Monitoring Using a Smart Phone's Accelerometer with Hierarchical Classification." In *Intelligent Environments (IE), 2010 Sixth International Conference on*, pp. 158–163. IEEE, 2010.

[ZSV14] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D Gribble, Arvind Krishnamurthy, and Henry M Levy. "Customizable and extensible deployment for mobile/cloud applications." In *Proc. USENIX OSDI*, 2014.