Università degli Studi di Cagliari

Ph.D. DEGREE

ELECTRONIC AND COMPUTER ENGINEERING

Cycle XXXIV

TITLE OF THE Ph.D. THESIS

ADAPTIVE COGNITIVE SENSOR NODES FOR THE INTERNET OF MEDICAL THINGS

Scientific Disciplinary Sector(s)

ING-INF/01

Ph.D. Student:	MATTEO ANTONIO SCRUGLI
Supervisor	PAOLO MELONI

Final exam. Academic Year 2020/2021

Thesis defence: April 2022 Session

Università degli Studi di Cagliari
Department of Electrical and Electronic Engineering (DIEE)
Ph.D. degree, XXXIV cycle

Adaptive cognitive sensor nodes for the internet of medical things

Matteo Antonio Scrugli
December 2021

Advisor: Prof. Paolo Meloni

“One of my most productive days was throwing away 1,000 lines of code.”
Ken Thompson

Acknowledgements

This work was supported by EU Commission for funding ALOHA Project (H2020) under Grant Agreement n. 780788. This work was also supported by the joint research and development project F/050395/01-02/X32, INSIEME: INtelligent Systems for Integrated hEalth ManagEment, CUP:B28I17000060008, funded by Italian MISE (Ministero dello Sviluppo Economico), D.M. 01/06/2016, Axis 1, action 1.1.3. of the National Operative Program «Imprese e Competitività»2014-2020 FESR, Horizon 2020 – PON I&C 2014-20.

Abstract

The Internet of Medical Things (IoMT) paradigm is becoming mainstream in multiple clinical trials and healthcare procedures. It relies on novel, very accurate and compact sensing devices, network and communication infrastructures, opening previously unmatched possibilities of implementing data collection and continuous patient monitoring. Nevertheless, to fully exploit the potential of IoMT, some steps forward are needed. First, the edge-computing paradigm must be added to the picture. A certain level of near-sensor processing has to be enabled, to improve the scalability, portability, reliability and responsiveness of the IoMT nodes. Second, novel, increasingly accurate data analysis algorithms, such as those based on artificial intelligence and deep learning, must be exploited. To reach these objectives, designers, and programmers of IoMT nodes, have to face challenging optimization tasks, in order to execute fairly complex computing processes on low-power wearable and portable processing systems, with tight power and battery lifetime budgets. In this thesis, the implementation on resource-constrained computing platforms of a cognitive data analysis algorithm based on a convolutional neural network was explored. The treatment of cardiovascular disease and fitness tracking were chosen as use cases within the IoMT context to validate our approach. To minimize power consumption, an adaptivity layer has been added, the latter dynamically manages the hardware and software configuration of the device to adapt it at runtime to the required operating mode. The experimental results show that adapting the node setup to the workload at runtime can save up to 60% power consumption. The optimized and quantized neural network reaches an accuracy value higher than 97% for arrhythmia disorders detection and more than 97% for detecting some specific physical exercises on a wobble board.

Contents

1 Introduction **1**

2 State of the art **5**

2.1 IoT for healthcare solutions . . . 6
2.1.1 Integration of cognitive processing on IoMT devices . . . 6
2.2 Sensory nodes for cardiovascular disease treatment . . . 7
2.2.1 Cognitive approach for cardiovascular disease detection . . . 8
2.3 Cognitive IoMT node for sensorimotor exercise detection . . . 10
2.4 Edge-computing paradigm & runtime adaptivity on nodes with low or high computational capabilities . . . 12
2.4.1 Runtime adaptivity . . . 13
2.4.2 CNN workload partitioning and mapping . . . 15
2.4.3 Dynamic Voltage and Frequency Scaling . . . 15
2.5 Current scenario summary and proposed novelties . . . 17

3 Methodologies, techniques and architectures **19**

3.1 Hardware platform . . . 21
3.1.1 SensorTile . . . 21
3.1.2 Orlando . . . 24
3.2 Middleware & Operating System . . . 27
3.2.1 Common Microcontroller Software Interface Standard . . . 27
3.2.2 FreeRTOS . . . 36
3.2.3 Orlando middleware . . . 38
3.3 PyTorch framework & quantization . . . 40
3.4 Electrocardiogram configuration . . . 41
3.4.1 MIT−BIH Arrhythmia Database . . . 43
3.5 Application model & ADAptive runtime Manager . . . 44
3.5.1 ADAptive runtime Manager algorithm . . . 47

4 Cardiovascular disease detection on electrocardiogram **52**
4.1 Operating modes . . . 53
4.1.1 Operating mode: Raw data . . . 53
4.1.2 Operating mode: Peak detection . . . 55
4.1.3 Operating mode: CNN processing . . . 55
4.2 The peak detection algorithm . . . 56
4.3 Designing the CNN: training and optimization . . . 58
4.3.1 Model exploration . . . 61
4.3.2 Post-deployment degradation and refinement with Augmentation . . . 66
4.4 Experimental results . . . 67
4.4.1 Pre-deployment CNN accuracy . . . 67
4.4.2 Post-deployment CNN accuracy . . . 69
4.4.3 Power consumption measures . . . 72
4.4.4 Power model & operating mode power consumption estimation 74
4.5 Work comparison . . . 77

5 Sensorimotor exercise detection using a wobble board **79**
5.1 Operating modes . . . 80
5.1.1 Operating mode: raw data . . . 80
5.1.2 Operating mode: basic balance . . . 82
5.1.3 Operating mode: CNN . . . 82
5.2 Neural network design . . . 83
5.2.1 Data augmentation & generalization . . . 86
5.3 Experimental results . . . 87
5.3.1 Neural network accuracy . . . 87
5.3.2 Power consumption measures . . . 87
5.3.3 Power model & operating mode power consumption estimation 89

6 Runtime reconfiguration on multi-core platforms **91**
6.1 Adaptivity in Advanced Multi-core Hardware Platforms . . . 92
6.1.1 ADAM for Multi-Cores . . . 92
6.2 Splitting policies . . . 94
6.2.1 Splitting model on Orlando . . . 95
6.3 Operating mode . . . 98
6.4 Experimental results . . . 101
6.4.1 Experimental setup . . . 101
6.4.2 Single Channel . . . 103
6.4.3 Multi-Channel . . . 105

7 Conclusion 108

Repositories 110

Bibliography 111

Figures

3.1 General overview of the proposed system. 20
3.2 IoMT node architecture overview. 21
3.3 SensorTile. 22
3.4 Block diagram of the SensorTile board. 24
3.5 Additional board. 25
3.6 SoC top-level block diagram. 25
3.7 V_{dd} measurement and approximation for each system frequency. . . 27
3.8 Example of im2col on a 2D image with a 3x3 kernel, padding size of 1 and stride size of 2 [1]. 28
3.9 Convolution of three-dimensional (height, width, and channel) data [1]. 29
3.10 Experiment with different data layouts, CHW and HWC. The matrix-multiplication runtime is the same for both data layout styles. HWC has a shorter im2col runtime [1]. 30
3.11 Einthoven Triangle. 43
3.12 Electrodes placement according to the modified limb lead method [2]. 44
3.13 10-second ECG trace, recording number 108 in the MIT−BIH Arrhythmia dataset. In the upper part is present the MLLII trace, in the middle part the notes and in the lower part the modified lead V1 trace. 45
3.14 Simple task chain. 45
3.15 Two possible configurations of a generic system. 47

4.1 Frequency response of the circuit configuration chosen for the AD8232. 53
4.2 ECG application model. 54
4.3 Filtering block diagram. 57
4.4 Raw signal and filtered one from two different recordings. 57
4.5 CNN structure and two possible classes of labels. 62

4.6 Exploration of the chosen neural network model, the name comes from *labels_ conv1OutputFeatures_ conv2OutputFeatures_ fc1Outputs*. 63
4.7 Results obtained from the training of the model having: 20 output features for *Conv1*, 20 output features for *Conv2* and 100 output for *Fc1*. 64
4.8 For the most accurate models, the energy consumption for a single CNN task call is shown. The dotted line represents the maximum allowable drop in accuracy (0.5% with respect to the most accurate model) for NLRAV (red line) and NSVFQ (blue line) classes. The models marked with an "×" do not respect the constraints imposed on the minimum necessary accuracy value. 65
4.9 Qualitative example of augmentation. 67
4.10 ROC curve and AUC value for *NLRAV_4_4_100* and *NSVFQ_4_4_100* models with augmentation support. 68
4.11 Confusion matrix for *NLRAV_4_4_100* and *NSVFQ_4_4_100*. . 69
4.12 False positives and false negatives cases resulting from the peak detection algorithm and classification with remaining true positive cases for NLRAV and NSVFQ classes using CNNs trained with augmentation techniques. 70
4.13 Taking into consideration the true positive peaks obtained with a tolerance equal to 50 samples, the statistical distribution of the accuracy values for each ECG recording, obtained from the classification on the validation set, is represented. The floating-point and fixed point models are tested, inference with centered and non-centered peaks is also tested. In orange, the median value. 71
4.14 Circuit used to measure power consumption. 72
4.15 The graph summarizes the energy consumption for different heartbeat rates, when data sending is enabled (Tx) or not (No Tx). *Raw OM* does not depend on heartbeat nor on the threshold settings and the threshold task is disabled, so only one value is shown. 73
4.16 Estimation of energy consumption for each task of each operating modes at 60 bpm. 76

5.1 Wobble board used to validate our approach. 80
5.2 Application model. Top *raw OM*, middle *balance OM*, bottom *CNN OM*. 81

5.3 The four common wobble board exercises recommended by physiotherapist Anders Heckmann [3]. (a) Balance while keeping as steady as possible. (b) Move the board back and forth. (c) Move the board from sideto side. (d) Clockwise and counterclockwise circular movement. The Figure was extracted from Nilsson et al. [4]. 82
5.4 CNN structure and classes description. 85
5.5 Example graph of single acquisition of the "two leg tilts" exercise. The two axes in the lower plane represent the raw data acquired by the sensor. The vertical axis represents time in milliseconds. 86
5.6 Validation set confusion matrix, to the left the model with floating point weights and to the right fixed point weights. 88
5.7 Power consumption for each OM. 89
5.8 Estimation of energy consumption for each task of each OM. 90

6.1 One core for each convolutional layer. 92
6.2 Subdivision of the workload given by the *blocks* into several cores. 93
6.3 Example of balanced pipeline (grey shadows show the potential execution of successive pipelined computations of the same application). 95
6.4 Example of redistribution of the workload over multiple cores (grey shadows show the potential execution of successive pipelined computations of the same application). 97
6.5 ECG application model for the CNN processing operating mode [5] 99
6.6 Neural network structure. 100
6.7 The baseline setup for the selected use case on Orlando. 101
6.8 Dataflow on cores with a balanced pipeline and medium workload (ADAM-IF with maximum 100 *bpm*). 103
6.9 Comparison of power consumption considering different adaptation policies (with or without ADAM) and different workloads. 104
6.10 Comparison of power consumption in different ADAM configurations with high workloads (processing of six ECG streams). 106

Tables

2.1 Structure of some devices proposed in literature. 11
2.2 Qualitative comparison with CNN workload partitioning and mapping studies. 16
2.3 Qualitative comparison of some works in literature using DVFS techniques. 17

3.1 SensorTile current consumption in different operating states. 23
3.2 Core activation and deactivation functions. 39
3.3 Synchronization functions, which are mainly used to read/write on the same FIFO in a mutually exclusive way. 39
3.4 Call functions, used by the ADAM system to manage the execution of tasks on the cores. 39

4.1 True positives, false positives, and false negatives of our peak detection algorithm with a tolerance of 50 samples for each ECG recording on MIT−BIH arrhythmia database. 59
4.2 Classes distribution over the dataset or the false negative subset. . . 60
4.3 Hyperparameters used during the training phase. 61
4.4 Parameters used for *NLRAV_4_4_100* and *NSVFQ_4_4_100* training phase. 68
4.5 Summary of consumption and execution time for each task. 75
4.6 Summary of consumption of peripherals. 75
4.7 Results in terms of accuracy value on MIT-BIH dataset (see classes names in Figure 5.4). 78

5.1 Hyperparameters used during the training phase. 84
5.2 Model parameters. 85
5.3 Augmentation parameters. 87
5.4 Summary of consumption and execution time for each task. 89
5.5 Summary of consumption of peripherals. 90

6.1 rpc_call(...) function arguments. 98

Listings

3.1 arm_nn_mat_mult_kernel_q7_q15() source code with our modification. 31
3.2 arm_convolve_HWC_q7_basic_nonsquare() source code with our modification. 32
3.3 arm_fully_connected_q7_opt() source code with our modification. 33
3.4 arm_maxpool_q7_HWC_1D() source code with our modification. 34
3.5 compare_and_replace_if_larger_q7() source code with our modification. 36
3.6 FreeRTOSConfig.h file. FreeRTOS original source code. 37
3.7 port.c file. FreeRTOS original source code. 37

Algorithms

3.1 ADAM algorithm. 51

6.1 ADAM-FF policy algorithm. 96
6.2 ADAM-IF policy algorithm 96

Acronyms

1D	1-Dimensional 30
2D	2-Dimensional 29
3D	3-Dimensional 22, 29
5G	5th Generation 5
AAMI	Association for the Advancement of Medical Instrumentation 109
ACAP	Adaptive Compute Acceleration Platform 12
ACC	Accuracy 61
ADAM	ADAptive runtime Manager ix, x, 3, 14, 19, 21, 26, 39, 46–49, 77, 91–95, 97–99, 101–108
ADC	Analog to Digital Converter 52, 55
AI	Artificial Intelligence 2, 6, 7, 12, 13, 91
ANN	Artificial Neural Network 7
ANSI	American National Standards Institute 109
API	Application Programming Interface 20, 23, 27, 38, 77, 91
AR	Augmented Reality 22
ASIC	Application Specific Integrated Circuit 7, 16
AUC	Area Under Curve 67
BHD	British Hypertension Society 109
BIH	Beth Israel Hospital 9, 11, 43, 44, 52, 56, 58, 108
BLE	Bluetooth Low Energy 19, 22, 49, 75, 79, 90
BSD	Berkeley Software Distribution 40

CHW	Channel-Width-Height 29
CMSIS	Common Microcontroller Software Interface Standard 12, 27–30, 36, 60, 77, 83, 84
CNN	Convolutional Neural Network vii–x, 3, 8–11, 14–17, 27, 54, 56, 57, 60–62, 65–67, 69, 70, 72, 74, 75, 77, 82–84, 89, 92, 94, 97, 99, 101, 102, 108
CPU	Central Processing Unit 7, 11, 16, 28
CS	Compressive Sensing 8
CVD	Cardiovascular Diseases 1, 2, 8
DBN	Deep Belief Network 11
DC	Direct Current 57
DFSDM	Digital Filters for external Sigma-Delta Modulators 22
DL	Deep Learning 2, 17
DNN	Deep Neural Network 11, 16
DSP	Digital Signal Processor 26, 38, 39, 95, 97, 103
DVFS	Dynamic Voltage and Frequency Scaling x, 14–17
ECG	Electrocardiogram viii, ix, 2–5, 7–10, 18, 42–44, 52, 53, 55, 56, 58, 66, 69–71, 74, 75, 98, 99, 105, 106, 108, 109
EIS	Electronic Image Stabilization 22
ES	Early Stopping 61, 84
ESD	ElectroStatic Discharge 24
FAIR	Facebook's Artificial Intelligence Research 40
FF	Frequency First xiii, 94–96, 102–106
FIFO	First In First Out x, 3, 38, 39, 44, 46–51, 92, 93
FN	False Negative 58–61
FP	False Positive 58, 59, 61
FPGA	Field Programmable Gate Array 8, 10, 12
FPU	Floating-Point Unit 22

FreeRTOS Free Real-Time Operating System xii, 36–38, 79
FT Fixed Topology 102–104, 106

GPIO General Purpose Input/Output 23
GPU Graphics Processing Unit 7, 12, 16

HWC Height-Width-Channel 29, 32, 34

I^2C Inter Integrated Circuit 24
IBM International Business Machines 13
IF Idle First ix, xiii, 94–96, 102–106
IMU Inertial Measurement Unit 22
IO Input/Output 26
IoMT Internet of Medical Things 1, 3, 6, 108
IoT Internet of Things 1–3, 5, 6, 8, 9, 13, 91

KNN K-Nearest Neighbors 8, 9

LA Left Arm 42
LCD Liquid Crystal Display 22
LDA Linear Discriminant Analysis 7
LDO Low-DropOut 24
LL Left Leg 42
LP Low Pass 57
LSA Latent Semantic Analysis 9, 11

MCT Medtronic SEEQ Mobile Cardiac Telemetry 6
MCU MicroController Unit 22
MEMS Micro Electro-Mechanical Systems 22
microSD micro Secure Digital 24
MIT Massachusetts Institute of Technology 8, 9, 11, 43, 52, 56, 58, 108
MLL Modified Limb Lead 43
MLLII Modified Limb Lead II 43, 44, 52

NN	Neural Network 2, 12, 28, 36, 60, 83, 84
OIS	Optical Image Stabilization 22
OM	Operating Mode viii, ix, 3, 56, 57, 72–75, 77, 80–82, 87–90
OS	Operating System 20, 37, 91
OTG	On-The-Go 22
PCA	Principal Component Analysis 7
PPV	Positive Predictive Value 58
RA	Right Arm 42
RAM	Random Access Memory 7, 26, 36
RBM	Restricted Boltzmann Machine 11
ReLU	Rectified Linear Unit 30, 99
RF	Random Forest 9
ROC	Receiver Operating Characteristic 67
RPC	Remote Procedure Call 38, 97
RTOS	Real-Time Operating System 14, 36–38
SIMD	Single Instruction Multiple Data 29, 36, 77
SoC	System on a Chip 12, 48, 49, 51
SoM	Systems on a Module 12
SPI	Serial Peripheral Interface 49
SQA	Signal Quality Assessment 8
SRAM	Static Random Access Memory 26, 38, 97
SSF	Static System Frequency 102–106
SVM	Support-Vector Machine 8
SWD	Serial Wire Debug 23, 24
TN	True Negative 58, 61
TP	True Positive 58, 59, 61
TPR	True Positive Rate 58
USB	Universal Serial Bus 22, 24

VR	Virtual Reality 22
WBAN	Wireless Body Area Network 9, 10
WHO	World Health Organization 2, 10

1
Introduction

The Internet of Things (IoT) paradigm, declined in the so-called Internet of Medical Things (IoMT), enables seamless collection of a wide range of data streams, that can be analyzed to extract relevant information about the patient's condition. However, in order to make IoMT really ubiquitous and effective, a step forward is needed to improve scalability, responsiveness, security and privacy. Most of the efforts aiming in this direction focus on the adoption of an edge-computing approach. Data streams, acquired by sensors, can be processed, at least partially, at the edge, before being sent to the cloud, on adequate portable/wearable processing platform. This provides several advantages. First, it reduces bandwidth requirements. Near-sensor processing can extract from raw data more compact information. In this way, less communication bandwidth is required to the centralized server, and, at the same time, the energy consumption related to wireless data transmission is drastically reduced. Second, near-sensor processing can improve reliability. Monitoring must not rely necessarily on connection availability and, if immediate feedback to the user and/or local actuation is needed, the delays through the network can be avoided. Moreover, pre-processed information can be delivered to the cloud, preserving user privacy avoiding the propagation of sensitive data.

An extremely important field of application of IoMT is related to the treatment of Cardiovascular Diseases (CVD), a major public health problem that generates millions of deaths yearly and impacts significantly on health-related pub-

lic costs. As an example, in 2016, $\approx$ 17.6 million (95% CI, 17.3−18.1 million) deaths were attributed to CVD globally, representing an increase of 14.5% (95% CI, 12.1%–17.1%) since 2006 [6]. In Europe, the CVD impact on the economy is estimated to be around €210 billion [7]. It is commonly accepted that machine learning and IoT are important for creating a novel assisted living methodology [8], this is also true for CVD monitoring. In Maskeliūnas et al. [8], many aspects regarding assisted living and how to improve the quality of this category of devices are discussed, some of them are: the introduction of machine learning can allow the device to adapt autonomously to the environment and reduces manual interventions by an operator; the combination of Artificial Intelligence (AI) and IoT leads to improvements from the point of view of comfort and energy saving, allows constant monitoring of the environment and learning from its behavior. CVD treatment with remote monitoring involves in most cases analysis of Electrocardiogram (ECG) signals.

The World Health Organization (WHO) 2010 guidelines state that, excluding special cases, an average adult should engage in moderate-intensity physical activity for at least 150 minutes per week and high-intensity physical activity for 75 minutes per week. Due to their ease of use, accuracy, and portability, fitness tracker devices have grown in popularity in recent years. In addition to the choice of CVD treatment as the main use case, physical activity tracking is one of the use cases chosen in this work to support our thesis.

Creating embedded platforms implementing such kind of analysis is promising, but, at the same time, very challenging, for several reasons:

- **Requires edge computing at low energy/cost budget**: Sensor nodes must be wearable and affordable to implement ubiquitous patient monitoring. Given the high data rate produced by the sensors, wireless raw data transmission requires an energy budget that cannot be negligible when the task is implemented in a portable and inexpensive computing device.

- **Requires cognitive computing**: state-of-the-art events detection/tracking tasks are often based on the analysis of manually designed features with are hard to craft and extract online from the original signal. Thus the community is shifting focus to techniques based on Neural Networks (NNs) and Deep Learning (DL), that rely on automatically learned features. However, existing approaches that use DL for the events recognition, rarely pay attention to energy consumption to be deployed on low-power processing systems. Thus, pretty often do not take into account workload reduction and post-deployment accuracy evaluation.

- **Requires adaptivity**: Intensity of the processing workload is very depen-

> dent on the needed level of detail and also intrinsically data-dependent. For example, in the case of ECG analysis, the information to be analyzed is usually contained in waveform shape of peaks, thus the rate of sample frames to be analyzed is directly dependent on the patient's heartbeat rate. This paves the way to energy consumption reduction by means of an adaptive management of the system, that reconfigures itself on the basis of the detected data and on the chosen Operating Mode (OM).

In this thesis, the implementation of a system for at-the-edge cognitive processing of ECG data and fitness activity was explored. A hardware/software setup for the processing system inside the IoMT node has been designed. Two reference platforms were selected in order to validate the implemented system on both a single-core and a multi-core platform. For the single-core case SensorTile platform was used, a compact processing microcontroller-based device developed by STMicroelectronics. As a multi-core platform, Orlando board was chosen, also developed by STMicroelectronics. The implemented system makes use of a quantized Convolutional Neural Network (CNN), that has been validated in post-deployment and recovers accuracy drops that arise in real online utilization. Moreover, a step further in hardware/software optimization using adaptivity has been taken, allowing the system to reconfigure itself, to suit different operating modes and data processing rates. To this aim, besides executing the tasks that implement sensor monitoring and on-board processing, the system includes a component called ADAptive runtime Manager (ADAM), able to dynamically manage the hardware/software configuration of the device optimizing power consumption and performance. ADAM creates and manages a network of processes that communicate with each other via First In First Outs (FIFOs) queues. The morphology of the process network varies to match the needs of the operating mode in execution. ADAM can be triggered by re-configuration messages sent by the external environment or by specific workload-related variables in the sampled streams (e.g. patient's heartbeat pace). When triggered, ADAM changes the morphology of the process network, switching on or off processes, and reconfigures the inter-process FIFOs. Moreover, depending on the new configuration it changes the hardware setup of the processing platform, adapting power-relevant settings such as clock frequency, supply voltage, peripheral gating.

The remainder of this thesis is as follows: Chapter 2 shows what the current IoT challenges are as applied to continuous monitoring devices in the health care domain; in Chapter 3 the hardware platforms used and the tools exploited and implemented to support our thesis are described. Additionally, the ADAM component developed, capable of dynamically managing the device configuration, is

explained; Chapter 4 presents the implementation case related to the detection of anomalies in ECG trace, shows how the application model was adapted to this use case, the techniques exploited and how the cognitive analysis was designed. Finally, it is shown how the system handles post-deployment non-idealities; Chapter 5 discusses how the system was designed to be able to identify the execution of some simple sensorimotor exercises on a common wobble board; in Chapter 6 design changes are described to adapt the system seen in Chapter 4 to run on a multi-core platform. A multi-core platform provides greater computational capacity, but the ability of such devices to handle parallel computing and pipelining capabilities must be managed; finally, Chapter 7 outlines our conclusions.

2
State of the art

The IoT consists of a network containing a huge number of interconnected devices, as a result, devices are becoming smarter and the quality of information available from the network is increasingly high. On the other hand, the IoT network is still trying to take its own shape, however, its technological development has already started to take giant steps. The real expansion will presumably take place with the advent of the 5th Generation (5G) network, the standards of this network allow the IoT to develop throughout the territory in a widespread manner. In order to design a reference platform with the purpose of exploiting it as an IoT node, it is necessary to study what the current technology makes available, understand what are the advantages but also the limitations. This chapter will mention some of the works proposed in literature that offer IoT solutions for health care. Many of these solutions involve collecting and analyzing data only on the cloud. The cognitive edge-computing paradigm on the other hand introduces many advantages in terms of responsiveness, accuracy and data security. We will focus on works dealing with ECG monitoring and anomalies detection but also on sensorimotor exercise detection, showing how deep learning-based analysis achieves detection accuracy values equal to or better than more traditional methods. Finally, we compare methodologies used in the literature to manage workload and energy consumption in high computational capabilities edge-devices.

2.1 IoT for healthcare solutions

Multiple solutions involving the use of sensor networks in hospitals or at home and the IoMT are proposed in literature [9, 10, 11]. Most of these studies exploit a cloud-based analysis: data is usually encapsulated in standard formats and sent to remote servers for data mining. Most research work takes into account wearability and portability as main objectives when developing IoMT-based data sensing architectures, thus devices available on the market can guarantee autonomy for days or weeks [12, 13].

Commercial devices such as Medtronic SEEQ Mobile Cardiac Telemetry (MCT) System and ViSi Mobile System have been designed to fit non-invasively into the skin and body, both devices allow vital sign monitoring, in particular, the first one, focuses on cardiac activity. These devices provide for raw data streaming to the cloud and arrhythmias recognition, however, power consumptions in different operating modes are not specified, and no data relating to the detection accuracy are available.

2.1.1 Integration of cognitive processing on IoMT devices

Cognitive IoT for domestic and hospital scenarios are adaptive, interactive and contextual. Adaptive describes a system that can learn and adapt to diverse contexts without user involvement, since a home environment, user requests and needs fluctuate in real time. Interactive references a system that can interact with humans, services, tools and devices; the system can also ask a user questions, if a situation under analysis is still unclear. A contextual system can recognize time, temperature, pollution, noise, human body language, requests and needs [8]. Although AI is not a new concept, its application still has critical issues [14]. This is due to the extremely dynamic nature of smart environments, as well as the massive amounts of hard real-time data generated by residents and the environment itself [15]. On the other hand, IoT technologies provide a solid foundation for distributed data collection and environment control via various actuators and appliances. The combination of AI and IoT opens up new possibilities, allowing for new types of intelligent pervasive systems and platforms [16, 17], providing the highest level of comfort, energy savings, and new personalized services for residents of intelligent environments. To really use cognitive computing at the edge, more complex and accurate algorithms, such as those exploiting artificial intelligence or deep learning, must be targeted. Their efficiency has been widely demonstrated on high-performance computing platforms in medical and health care fields. Some examples are [18], where an NVIDIA GeForce GTX 1080 Ti (11 GB) is used [19], that

uses a 3.5 *GHz* Intel Core i7-7800X CPU, RAM 32 GB, and a GPU NVIDIA Titan X (Pascal, 12 GB), or [20], based on an i7-4790 CPU at 3.60 *GHz*. However, how to map state-of-the-art cognitive computing on resource-constrained platforms is still an open question. There is an ever-increasing number of approaches focusing on machine learning and artificial intelligence to identify specific events in physiological data. In Tabal et al. [21] and Magno et al. [22] authors exploit Artificial Neural Network (ANN) to detect specific conditions from the proposed data. In Magno et al. [22], an ANN is used to identify the emotional states (happiness or sadness) of the patient. However, network topologies are still very basic, highly tuned and customized to fit on the target device.

2.2 Sensory nodes for cardiovascular disease treatment

As already described in Chapter 1, cardiovascular diseases are one of the most frequent causes of death worldwide, which is why there is also a strong interest among the scientific community to address this issue. There are several works that implement ECG monitoring on customized chips, it is shown that with low energy consumption it is possible to classify cardiac anomalies in real-time even using AI methods. In Lee et al. [23] wavelet theory was adopted to perform features extraction and classification, an accuracy of 97.25% was achieved on arrhythmias recognition. In Chen et al. [24] an excellent job of researching the compromise between complexity and performance on different classifiers with different lead configurations was made. In particular, they obtained good results with a Linear Discriminant Analysis (LDA) using the spectral energy of the PQRST complexes as features. In Bayasi et al. [25] a Naive Bayes classifier is exploited, they obtained an accuracy equal to 86% on arrhythmias recognition using the PQRST points detected with a Pan-Tompkins algorithm as features. In Ince et al. [26] a wavelet-based algorithm is used to extrapolate morphological and temporal features, a Principal Component Analysis (PCA) is used to reduce the dimensionality and redundancy of the features. An accuracy of 97.4% was obtained with an evolutionary ANN. In Amirshahi and Hashemi [27] and Kolağasioğlu [28] possible implementations of arrhythmia classification algorithms based on spiking neural networks are shown.

Despite the choice of using Application Specific Integrated Circuits (ASICs) leads to the development of ultra low-power devices, but is not a very versatile solution. A large number of works focus on implementing efficient off-the-shelf

commercial devices to facilitate easier community adoption of these techniques. Several target technologies have been used in the literature, such as Field Programmable Gate Array (FPGA) or microcontroller-based boards. A substantial number of research works are dedicated to studying IoT devices in the medical field, in particular ECG monitoring and anomalies detection. In Deshmukh and Chaskar [29] and Arun et al. [30], authors deal with simple ECG monitoring on wearable devices. In Satija et al. [31] and Xu [32], authors treat with particular attention the aspect of Signal Quality Assessment (SQA), identifying the signal quality level is useful for knowing when to ignore the input data or even when to put the device into a sleep state. In Natarajan and Vyas [33] a system that uses Compressive Sensing (CS) to compress bio-signals in a power-efficient way is proposed. In Spanò et al. [34], authors propose a monitoring device with a particular focus on low energy consumption.

2.2.1 Cognitive approach for cardiovascular disease detection

In some works proposed in the literature dealing with ECG signal monitoring, local processing is used only for implementing easy checks on raw data and/or marshaling tasks for wrapping the sensed data inside standard communication protocols [35, 36, 37, 38]. On the other hand, there are numerous works in the literature that exploit cognitive computing for CVD detection, even at the edge. To really use cognitive computing at the edge, more complex and accurate algorithms, such as those exploiting artificial intelligence or deep learning, must be targeted.

The cognitive approach that involves the use of CNNs shows promise in terms of accuracy in detecting ECG signal arrhythmias compared to other traditional strategies based or not on artificial intelligence algorithms [39, 40], Moreover, in most cases, the use of CNNs allows to classify an ECG signal even if not preprocessed. The most common strategies present in many state-of-the-art works that allow to improve the efficiency of these IoT nodes are: moving the inference operations at the edge, choosing a low-power devic and quantization techniques to speed up the network execution of the inference stage.

In Deshmane and Madhe [41] it's shown how with CNNs approach it's possible to obtain higher accuracy values for patient ECG-based authentication (81.33%, 96.95%, 94.73% and 92.85% on MIT, FANTASIA, NSRDB and QT database respectively) if compared with other traditional methods such as K-Nearest Neighbors (KNN) and Support-Vector Machines (SVMs). Unlike our work, In Deshmane and Madhe [41] signal filtering techniques are used to clean up noise and features extraction, these techniques require a considerable amount of resources and lead

to a consequent increase in power consumption. The CNN input signal we are considering is a raw signal without any pre-processing.

In Walinjkar and Woods [42] the importance of real-time monitoring is discussed, they propose a system capable of recognizing anomalies on ECG, testing their methodology with various machine learning techniques.

Another interesting work was presented In Rani Roopha Devi et al. [39], in addition to the comparison with other techniques used to analyze the ECG trace, Latent Semantic Analysis (LSA) techniques were used to improve the accuracy of the network. Both training and inference take place on the cloud side, our aim is to move the inference to the edge of a low-power device in order to reduce latency times and reduce energy consumption due to wireless communication.

In Xitong et al. [43] excellent results were obtained for ventricular arrhythmias and supraventricular arrhythmias classification: 99.6% and 99.3% for accuracy value, 98.4% and 90.1% for sensitivity value, 99.2% and 94.7% for positive predictive value, respectively. In Xitong et al. [43] a double CNN is used, one of them takes as input the frequency domain information of the ECG signal (a fast Fourier transform is performed). This methodology, despite the excellent results in terms of accuracy, was not taken into consideration in our case because it's particularly expensive to perform on a microcontroller.

In Naz et al. [44], the authors obtain excellent results in terms of accuracy, they used a method similar to the one used In Ma et al. [45] to combine the output features of three independent neural networks (VGG19, AlexNet, and Inception-v3) and then classify them as a single output. The methodology used allows a significant increase in terms of accuracy (97.6%) compared to the use of neural networks chosen individually, however, this method is too resource-expensive in terms of memory and computational load to be executed in a microcontroller such as the one we selected. In Sakib et al. [40], again, there is proof of how neural networks obtain good results if compared with methods such as KNN and Random Forest (RF) (95.98% on MIT-BIH Supraventricular Arrhythmia Database) and the inference occurs directly from the IoT node but the power consumption remain relatively high once again, they are used in fact non-low power devices such as Raspberry Pi 4 or Jetson Nano. Always In Sakib et al. [40], a good job of research has been done on the morphology of the CNN network that was more suitable for inference on ECG signals, the network we used provides a structure very similar to the one chosen In Sakib et al. [40].

In Azimi et al. [46], good results are obtained in terms of accuracy for arrhythmia classification using the MIT Arrhythmia dataset (96% using MIT Arrhythmia dataset), the inference does not occur on the cloud side, nor on the sensorial device at the edge, but from a device located within the same Wireless Body Area

Network (WBAN) network. The sensory device, therefore, remains in constant communication with the outside to send the raw data of the signal. An approach similar to [47] was chosen, an embedded device was chosen that is able to perform the inference directly on the node. In Burger et al. [47], a study was made on the variation of accuracy as a function of different quantization levels, they choose a precision of 12-bit with an accuracy of 97%, but it's visible that already from 6-bit upwards the accuracy levels exceed the 90%. Power consumption is around $200\,mW$ during computation and the node is based on FPGA technology.

In Yeh et al. [48] ResNet, AlexNet, and SqueezeNet neural network are used, an accuracy of 97% is achieved. In addition to the use of much more expensive computational neural networks, there is an overhead due to encoding the raw ECG signal into a JPEG image. The platforms chosen are Arduino UNO for signal acquisition and Raspberry Pi 3B+ to process the raw signal, making the power consumption considerably higher than a solution based only on microcontroller.

Other works with which we are confronted are [49, 50, 51, 52]. Table 2.1 lists and describes the main works we were confronted with.

2.3 Cognitive IoMT node for sensorimotor exercise detection

The guidelines of the WHO in 2010 document state that an average adult should engage in physical activity of moderate intensity for at least 150 minutes per week and 75 minutes per week at high intensity, excluding special cases [53]. Tracking and encouraging good levels of physical activity can improve people's health [54]. Fitness tracker devices have had a rapid development in recent years, due to their ease of use, accuracy, and portability. Events in a trackable signal, although seemingly simple, can be difficult to identify and recognise within the data stream.

In our use case, a CNN is used to identify and recognize simple physical exercises performed on a wobble board. In Nilsson et al. [4] and Blažica and Krivec [55], the authors propose the use of a wobble board in creative way, to entice people to its use, as it is very important in ankle rehabilitation. In a review of the concept of patient motivation [56], the authors describe how motivation has been considered in relation to rehabilitation associated with strokes, fractures, rheumatic disease, aging, and cardiac and neurological issues. The limited motivation of some individuals may, at least in part, be ascribed to the tedious nature of the ankle exercises and the inability to monitor one's improvement throughout the course of the train-

Reference	*Node technology*	*Processing placement*	*Power consumption*	*Accuracy and Dataset*	*Classification method*
Xitong et al. [43]	–	–	–	MIT-BIH ventricular arrhythmias and supraventricular arrhythmias classification: 99.6% and 99.3% for accuracy value, 98.4% and 90.1% for sensitivity value, 99.2% and 94.7% for precision value, respectively	CNN
Rani Roopha Devi et al. [39]	–	training and inference on cloud	–	94% for sensitivity vale, 99,3% for accuracy value on custom dataset	CNN + LSA method
Mathews et al. [49]	–	–	–	accuracies of 93.63% for ventricular ectopic beats and 95.57% for supraventricular ectopic beats on MIT Arrhythmia dataset	RBM and DBN
Sannino and De Pietro [50]	–	–	–	MIT-BIH dataset, NSVFQ classes: 99,09%, 98,55% and 99,52% for accuracy, sensitivity and specificity value, respectively	DNN
Naz et al. [44]	–	–	–	97.6% accuracy on MIT-BIH dataset for ventricular tachyarrhythmia desease	CNN
Kiranyaz et al. [51]	Intel I7-4700MQ at 2.4 GHz (eight CPUs) and 16-Gb memory, but designed to run on cheaper and less powerful architectures	inference on edge	–	accuracy up to 99% for ventricular ectopic beats and up to 97.6% for supraventricular ectopic beats on MIT Arrhythmia dataset	adaptive implementation of 1-D CNNs
Azimi et al. [46]	hierarchical structure	training on cloud, inference on edge	–	96% for accuracy value on MIT Arrhythmia dataset	CNN
Burger et al. [47]	FPGA	inference on edge	$200\,mW$ [I]	97% accuracy value for NLRAV classes on MIT Arrhythmia dataset	quantized CNN
Sakib et al. [40]	Jetson Nano (Quad-core ARM A57 @ 1.43GHz), Raspberry Pi 4 (Quad-core CortexA72 @ 1.5GHz), Raspberry Pi 3 (Quad-core Cortex-A53 @ 1.4GHz)	inference on edge	–	95.27% accuracy value for NSVF classes on MIT-BIH Supraventricular Arrhythmia Database	CNN
Yeh et al. [48]	Arduino UNO and Raspberry Pi 3B+	inference on edge	–	97% for accuracy value for: rhythm, QRS widening, ST depression, and ST elevation categories for detecting arrhythmia disorders on MIT Arrhythmia dataset	DNN
Hou et al. [52]	Raspberry Pi 3	training on cloud, inference on edge	–	98% for accuracy value for: normal(NOR), Left Bundle Brunch Block(LBB), Right Bundle Brunch Block (RBB), Paced beat(PAB), Premature Ventricular Contraction(PVC), Atrial Premature Contraction(APC), Ventricular Flutter Wave(VFW) and Ventricular Escape Beat(VEB) beats for detecting arrhythmia disorders on MIT Arrhythmia dataset	CNN
Our work	ST SensorTile	inference on edge	$9\,mW$ [II]	MIT-BIH dataset, NLRAV and NSVFQ classes: 97.42% and 96.98% for accuracy value, 98.26% and 98.22% for sensitivity value, 98.28% and 98.52% for precision value, respectively	quantized CNN

[I] When fully active.
[II] The device adapts itself to the workload and operating mode, the worst case is reported.

Table 2.1: Structure of some devices proposed in literature.

ing process [3]. A similar approach is considered in our work, the implemented system detects and identifies some simple sensorimotor exercises performed on the wobble board, giving a percentage of correct execution at the end of the exercise. As far as we know, our system is the only one that applies state-of-the-art deep learning-based techniques to recognize some specific movements on a wobble board, managing hardware and software dynamically in order to minimize power consumption.

2.4 Edge-computing paradigm & runtime adaptivity on nodes with low or high computational capabilities

On the one hand, the community is working on the design of low-power devices capable of supporting processing that exploits artificial intelligence techniques, these types of devices include accelerators, parallelization elements, and flexible power management. In the market or in the literature there are devices based on System on a Chip (SoC) or Systems on a Module (SoM) [57, 58, 59], embedded GPUs [60], or FPGA-based accelerators [61]. Among the different solutions regarding hardware accelerators, Adaptive Compute Acceleration Platform (ACAP) produced by Xilinx [62] combines the potential of three types of engines: Arm cores (scalar engines), the programmable logic (Adaptable Engines), and the new vector processor cores (AI engines). This tightly coupled hybrid architecture allows more dramatic customization and performance increase than any one implementation alone. In Przybył [63], the author presents a new type of architecture based on fixed-point arithmetic. The substantial difference compared to traditional solutions is the choice of the representation scale, which in this case occurs during compilation, while in the case of floating-point arithmetic it is done during execution. Other distinctive features of the proposed method are universal superscalar architecture and asymmetric structure of working registers.

On the other hand, there is a remarkable work of software development that tries to optimize firmware, middleware or libraries that optimize the mathematical tools used to implement solutions that exploit AI, some of them specifically created to run on specific platforms. Among these, we find: CUDNN [64] for GPUs ARM-NN for microcontrollers based on ARM Cortex processors [65]; Common Microcontroller Software Interface Standard (CMSIS). Other important techniques are pruning and quantization, widely discussed in the literature and which lead

to significant improvements in terms of energy efficiency. In Liang et al. [66], the authors discuss very thoroughly both the pruning and quantization compression techniques by testing them individually or simultaneously with different frameworks that support them. They analyze their strengths and weaknesses and provide practical guidance for compressing networks. Google Brain has proposed a new data format called Brain Float 16, it has gained wide adoption in AI accelerators from Google, Intel, Arm, and many others. The purpose is to minimize the prediction accuracy degradation due to a lowering of the data precision, with a consequent increase in throughput. The main difference comes from truncating the Floating Point 32 mantissa field from 23 bits to 7 bits [67]. In Dziwiński et al. [68], the authors propose a hardware implementation method of MRBF-TS systems.

2.4.1 Runtime adaptivity

An embedded system is defined as reconfigurable if it has the ability to change its software or hardware behaviour at runtime, this change may derive from an autonomous decision of the system or an external imposition. Software component reconfiguration corresponds to adding, removing or updating software tasks that implement the system. An example may be the addition of data processing task due to a user decision. Hardware reconfiguration, on the other hand, corresponds to adding, removing or updating physical components of the system. An example of this could be the variation of the system frequency imposed by an internal decision, due to the variation of the workload and therefore the real time constraints to be respected. In Boukhanoufa [69] the authors state that real-time systems can be distributed and operate in a dynamic environment. Therefore, to ensure its operation and maintainability, different operating modes and reliability techniques must be introduced. According to Wang et al. [70], new generations of embedded control systems are addressing new criteria such as flexibility and agility.

Introduced with International Business Machines (IBM)'s vision of autonomic computing, dynamic adaptation has been a very active area since the late 1990s and early 2000s [71]. Many of the existing techniques focus on adapting nodes with high computational capabilities. Adaptation of low-power, low-capacity devices has received less attention. With few resources available, the use of standard operating systems, middleware and frameworks is more complicated, making it difficult to design adaptive software. Low-level programming languages and ad-hoc manual solutions are often used. While this might be acceptable for building static and dedicated applications, this is not an ideal solution for IoT or data-dependent systems. One area where there is extensive study of runtime reconfiguration is certainly that of distributed systems. A distributed system is a collection of processes

working together to accomplish some task. Each process is a deterministic program unit able to execute separately from, and concurrently with, other processes [72]. In Fouquet et al. [73] the authors present a version of Kevoree called μ-Kevoree aimed at use on microcontroller-based devices. Kevoree is an open source project that aims at enabling the development of reconfigurable distributed systems [74]. Some points of Kevoree framework present in the authors implementation and of inspiration to our methodologies are:

- Parametric adaptation. Dynamic update of parameter values, e.g. change of sampling rate in a component that wraps a physical sensor (adaptation of instance properties).
- Architectural adaptation. Dynamic addition or removal of bindings or components, e.g. replication of software components and channels on different nodes to perform load balancing (adaptation of instances graph).

As far as we know, our proposed reconfiguration component called ADAM is the only one that deals with the above microcontroller reconfiguration aspects that can simultaneously interact with an Real-Time Operating System (RTOS) in terms of thread management and Dynamic Voltage and Frequency Scaling (DVFS). All the functionalities of this component will be described later in Section 3.5.

Regarding high-capacity devices, there are many works in the literature on *architectural adaptation* and specifically in the load balancing in multi-core devices. In Tuveri et al. [75], authors present a runtime approach to reconfigure core-to-task mapping and degree of parallelism of the application when the available resources or the application workload change, targeting shared-memory platforms. This work, however, focuses on fault tolerance and not on dynamically changing workload. Moreover, it's not tested on a dynamically manageable processing chip. In Jahn and Henkel [76], authors present an approach for workload self-organization on multi-cores. However, differently from our approach, computing tasks are seen as indivisible and only their mapping to the different cores is changed. In Choi et al. [77], tasks can be duplicated on multiple cores in order to have more freedom in pipeline organization, however, this is not tested on data-dependent workload and on cognitive computing.

Section 2.4.2 and 2.4.3 summarise the main approaches in literature about CNN workload partitioning and mapping and about dynamic voltage and frequency scaling in CNN-based designs. To the best of our knowledge, our work is complementary to both these kinds of strategies, being the first attempt to bring together dynamic remapping of CNN operators and consequent dynamic management of the hardware setup. Both these aspects are managed by the ADAM component.

2.4.2 CNN workload partitioning and mapping

There are different ways of distributing the workload (convolutional layers) on multiple cores, one of the possible strategies is called the "kernel-level", it involves the distribution of the kernels of a layer-*nth*, and therefore the output features calculation of the layer-*nth*, in a balanced way across all cores. Many state-of-the-art libraries adopt this solution, for example ARM-CL [78], tengine [79] or NCNN [80]. In Wang et al. [81], the authors indicate two approaches for workload partitioning in CNNs: layer-level and kernel-level strategy. The "layer-level" strategy involves assigning each layer to a group of cores, the number of cores per group is variable and, since not all cores are used for the layer-*ith*, pipelining techniques are used in order to maximize the throughput. The authors present *Pipe-it*, a framework capable of estimating the computational load of all layers and, using a layer-level strategy, very efficiently distributes the workload on the available cores in heterogeneous multi-core platforms. The layer-level strategy is equivalent to the method that we use in this work, however, In Wang et al. [81] it's not applied at runtime, but only during design time, design space exploration. In Wu et al. [82], the authors exploit the layer-level strategy obtaining good results as already demonstrated with the Pipe-it framework, in addition, they show the importance of taking into account the cache resources per core. Distribution of the workload that takes this fact into account minimizes the inter-core feature-map data movement overhead, finally demonstrating how, in the use case they considered, there is a 73% performance improvement. Also In Wu et al. [82], strategies are chosen only at design time. The reference platform that we used, Orlando board, lends itself well to the optimizations proposed In Wu et al. [82], due to the presence of intra-core memories that can reduce the features transfer, but the complexity of a generic neural network may require a quantity of memory for features that often exceed what is usually made available by data caches. In Table 2.2 the main features of the most important works in the literature concerning CNN workload partitioning and mapping are shown.

2.4.3 Dynamic Voltage and Frequency Scaling

Another aspect dealt with in our work and already widely discussed in the literature, concerns Dynamic Voltage and Frequency Scaling (DVFS). It has been discussed in the literature how to use the DVFS in order to better manage the temperatures of the processing units. In Huang et al. [83], they tackle the problem of overheating in different ways: by judiciously selecting tasks with different thermal characteristics as well as alternating the processor's active/sleep mode or

Work / Framework	***CNN workload partitioning and mapping***		***Runtime***
	Kernel-level	***Layer-level***	
ARM-CL, Arm [78]	✓		
tengine, OAID [79]	✓		
NCNN, Tencent [80]	✓		
Pipe-it, Wang et al. [81]	✓	✓	
Wu et al. [82]	✓	✓	
Our work	✓	✓	✓

Table 2.2: Qualitative comparison with CNN workload partitioning and mapping studies.

by exploiting the DVFS potential offered by the platform. In Yu et al. [84] and Yu et al. [85], the authors show how they dynamically choose the system output quality under temperature constraints, in the use cases they considered, the system output quality is highly dependent on the application of DVFS. In Ma et al. [86], on the other hand, the authors deepen the aspect linked to the reliability, the use of DVFS takes into account the minimization of the thermal cycles that stress the chip. In Weissel and Bellosa [87] and Vogeleer et al. [88], the authors exploit DVFS techniques with the sole purpose of maximizing energy efficiency. In particular, the first ones refer to general-purpose CPU systems while the second work focuses on the CPUs of commercial smartphones and how much the system frequency plays a fundamental role in them in terms of energy minimization.

Similarly to Weissel and Bellosa [87] and Vogeleer et al. [88], in our work DVFS techniques will be used to maximize energy efficiency, furthermore, techniques to respond to CNN-based workload variations by the use of task-mapping techniques at runtime on a multi-core platform will be discussed. There are other works that combine CNN applications with DVFS techniques on ASIC, CPU, or GPU based devices. In Nabavinejad et al. [89], the authors show how through the DVFS techniques and the choice of the precision of the Deep Neural Network (DNN) it is possible to reach a certain level of inference accuracy and power consumption under latency constraints. In Motamedi et al. [90], the authors develop a principled approach and a data-driven analytical model to optimize the granularity of threads during CNN software synthesis. In Bong et al. [91], the authors propose a CNN-based low-power facial recognition system, the DVFS mechanism is the basis of

Work	*DVFS constraints*		*DVFS on CNN*	*Dynamic partitioning and mapping*
	Temperature	*Energy*		
Huang et al. [83]	✓			
Yu et al. [84]	✓			
Yu et al. [85]	✓			
Ma et al. [86]	✓			
Weissel and Bellosa [87]		✓		
Vogeleer et al. [88]		✓		
Nabavinejad et al. [89]		✓	✓	
Motamedi et al. [90]		✓	✓	
Bong et al. [91]		✓	✓	
Santoro et al. [92]		✓	✓	
Our work		✓	✓	✓

Table 2.3: Qualitative comparison of some works in literature using DVFS techniques.

their method to increase energy efficiency. In Santoro et al. [92], authors use a performance-power analytical model fitted on a parametrized implementation of a DL accelerator in a 28-nm FDSOI technology to explore a large design space and to obtain the Pareto points that maximize the effectiveness of DVFS in the sub-space of throughput and energy efficiency. Finally, in Table 2.3 some of the main works in the literature dealing with the dynamic voltage and frequency scaling and our work are shown.

2.5 Current scenario summary and proposed novelties

Of the aforementioned devices on the market that are proposed as health monitoring solutions, the majority only provide data transmission to the cloud, thus not exploiting the potential of the edge-computing paradigm. However, edge-computing is not a simple application solution from a design point of view, especially when a

cognitive analysis is introduced. For example, the power consumption of resource-limited devices performing on-edge analysis may be high compared to the same system producing the data for cloud-side analysis. Or, moving the analysis on-edge may lead to an unacceptable drop in the quality and performance of the analysis. Therefore, it becomes important to exploit a model that better integrates with the resources used both from a software and hardware point of view. Cognitive analysis has been extensively validated in the literature by exploiting the computational potential of high-capacity devices based on floating-point architecture without often considering power consumption. In the previous sections, several examples are given of models based on artificial intelligence able to detect anomalies in the patient's ECG but evaluated on high-performance hardware. Today, there are few devices able to efficiently analyze patient's data by exploiting deep learning techniques on portable devices with limited resources. In this context, the term efficiency refers to a device capable of achieving high detection accuracy through deep learning techniques while maintaining low power consumption. To remedy some of these shortcomings, this thesis proposes:

- The definition of a hardware/software/firmware architectural template for the implementation of a remotely-controlled sensory node, allowing for near-sensor cognitive data processing for cardiac anomalies detection and physical exercises.
- Its validation on a state-of-the-art data analysis based on a Convolutional Neural Network.
- The evaluation of the effectiveness of in-place computing and operating mode dynamic optimization on ARM microcontroller platform and multi-core platform, as a method to reduce the power consumption of the node.

3

Methodologies, techniques and architectures

An overview of the system architecture as provided in this thesis is shown in Figure 3.1, it is divided into three levels. The lower level is composed of the sensor nodes, which acquire information from the environment. They are connected to the upper level using Bluetooth Low Energy (BLE) technology. The nodes are capable of reacting, reconfiguring their operating mode, to commands sent from higher levels, or to workload changes that can be detected near-sensor, thanks to the internal component called ADAM which will be described later. The intermediate level consists of several gateways, in charge of collecting the data from the sensor nodes and send them to the upper level. To test the approach presented in this work, the gateway was implemented with a Raspberry Pi 3 running a Linux operating system. For the same purpose, the cloud-based infrastructure, on top of the stack, has been implemented using Google App Engine. Data is stored securely on the cloud, and can be used for analysis or simply visualized by a healthcare professional. Such kind of user, accessing a web-based interface, can also send downstream commands to the nodes, to communicate a required change of the operating mode, e.g. changing the needed detail of acquisition of the patient's

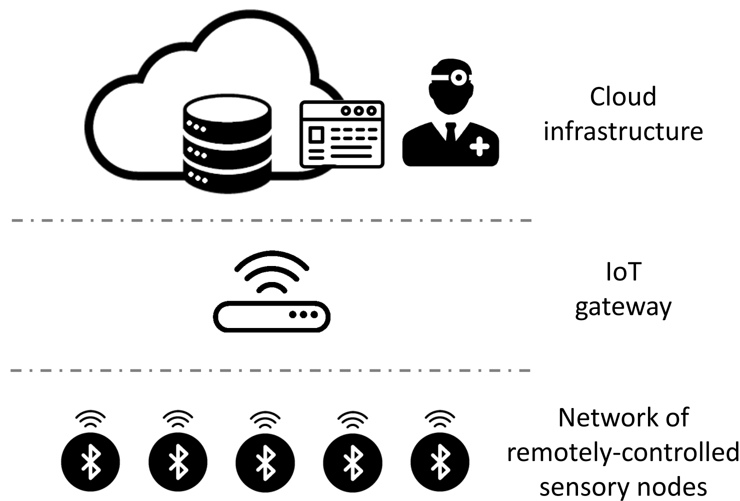


Figure 3.1: General overview of the proposed system.

parameters. In this thesis, attention will be focused only on the sensor node. The sensor node architecture itself can be seen as a layered structure, schematized in Figure 3.2. In the following sections, a description of each level and of the main instruments and methodologies used is provided.

The bottom layer is the hardware platform, which may be any kind of programmable microcontroller, that integrates sensors to take care of data acquisition, one or more processing elements, to manage housekeeping and pre-processing, and an adequate set of communication peripherals, implementing transmission to the gateway.

The hardware platform is managed at runtime by a firmware/middleware level, potentially including some Operating System (OS) support, to enable the management and scheduling of software threads. Moreover, this level must expose a set of low-level primitives to control hardware architecture details (e.g. access to peripherals, frequency, power operating mode, performance counting, etc.), and a set of monitoring Application Programming Interfaces (APIs) to continuously control the status of the hardware platform (e.g. energy and power status, remaining battery lifetime) and to characterize the performance of the different application tasks on it.

At the top of the node structure, there is the software application level, which executes tasks designed according to an adequate application model based on process networks, to be easily characterized and dynamically changed at runtime. To implement adaptivity, we add to the application an additional software agent, that

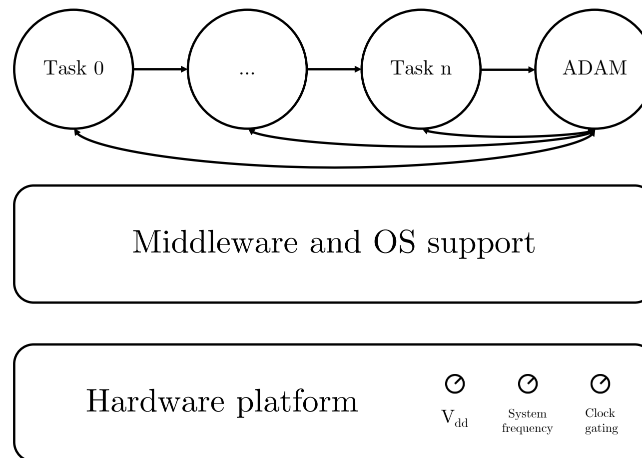


Figure 3.2: IoMT node architecture overview.

we call ADAM, which is in charge of monitoring all the events that may trigger operating mode changes (workload changes, battery status, commands from the cloud) and reconfigures the process network accordingly, to minimize power/energy consumption. Reconfiguration actions may involve changes in the processes network topology (activation/deactivation of tasks and restructuring of the inter-task connectivity) and play with the power-relevant knobs exposed by the architecture (e.g. clock frequency, power supply, supply voltage).

3.1 Hardware platform

As mentioned, to assess the feasibility of our approach based on dynamic reconfiguration, we have used a single-core microcontroller, namely an off-the-shelf platform designed by STMicroelectronics named SensorTile and a muli-core platform designed by STMicroelectronics named Orlando as node with higher computational capabilities. In the following, we will describe the main features of such platforms, exploited in this work.

3.1.1 SensorTile

The SensorTile measures $13.5 \times 13.5mm$ and is shown in Figure 3.3. It's equipped with an ARM Cortex-M4 32-bit low-power microcontroller. The small size and low power consumption allow the device to be powered also by the battery and to obtain good results in terms of autonomy without having to give up portability. The components included in the SensorTile are:

Figure 3.3: SensorTile.

- **MP34DT05-A:** Micro Electro-Mechanical Systems (MEMS) audio sensor omnidirectional stereo digital microphone;
- **LD39115J18R:** 150 mA low quiescent current low noise voltage regulator;
- **STM32L476:** Ultra-low-power with Floating-Point Unit (FPU) Arm Cortex-M4 MCU 80 MHz with 1 Mbyte of Flash memory, Liquid Crystal Display (LCD) controller, Universal Serial Bus (USB) On-The-Go (OTG), Digital Filters for external Sigma-Delta Modulators (DFSDM);
- **LSM6DSM:** iNEMO 6DoF Inertial Measurement Unit (IMU), for smart phones with Optical Image Stabilization (OIS)/Electronic Image Stabilization (EIS) and Augmented Reality (AR)/Virtual Reality (VR) systems. Ultra-low power, high accuracy and stability;
- **LSM303AGR:** Ultra-compact high-performance eCompass module: ultra-low power 3D accelerometer and 3D magnetometer;
- **LPS22HB:** MEMS nano pressure sensor: 260-1260 hPa absolute digital output barometer;
- **BlueNRG-MS:** BLE Network Processor supporting Bluetooth 4.2 core specification;
- **BALF-NRG-02D3:** 50 Ω / conjugate match balun to BlueNRG transceiver, with integrated harmonic filter.

Several architectural knobs can be used to adapt the platform to different conditions. SensorTile can work in two main modalities: *run mode* and *sleep mode*, in which different subsets of the hardware components are active. Moreover, in each mode, the chip can be set to a different system frequency (from 0.1 MHz to 80 MHz). Depending on the chosen system frequency and operating state, the device uses different voltage regulators to power the chip. In Table 3.1, we list some mode configurations, and associated current consumption, selectable using the mode-management APIs offered by the platform vendor. For our experiments,

RUN (Range 1) at 80 MHz	120 $\mu A/ MHz$
RUN (Range 2) at 26 MHz	100 $\mu A/ MHz$
LPRUN at 2 MHz	112 $\mu A/ MHz$
SLEEP at 26 MHz	35 $\mu A/ MHz$
LPSLEEP at 2 MHz	48 $\mu A/ MHz$

Table 3.1: SensorTile current consumption in different operating states.

we have chosen to use two approaches to dynamically reduce power consumption:

- to change system frequency (and consequently voltage regulator settings) over time according to the workload.
- to use the sleep mode of the microcontroller whenever possible. The operating system automatically sets a sleep state when there are no computational tasks queued to be performed and a timer-based awakening can be used to restart the run mode when needed.

The block diagram of the SensorTile board is shown in Figure 3.4. SensorTile is also equipped with programming/debugging circuitry. In fact, two of the microcontroller's General Purpose Input/Output (GPIO) pins can be used to implement the Serial Wire Debug (SWD) interface commonly adopted to program/debug ARM processors. The SWD interface is connected to a programmer, which in this case is STM32 Nucleo board and through the software System Workbench for STM32 it was possible to write the source code and program the board.

Together with the SensorTile, STMicroelectronics provides a solderable interface base to allocate the chip (Figure 3.5). The latter also provides:

- **STBC08PMR:** 800mA STANDALONE LINEAR Li-Ion Battery charger with thermal regulation;

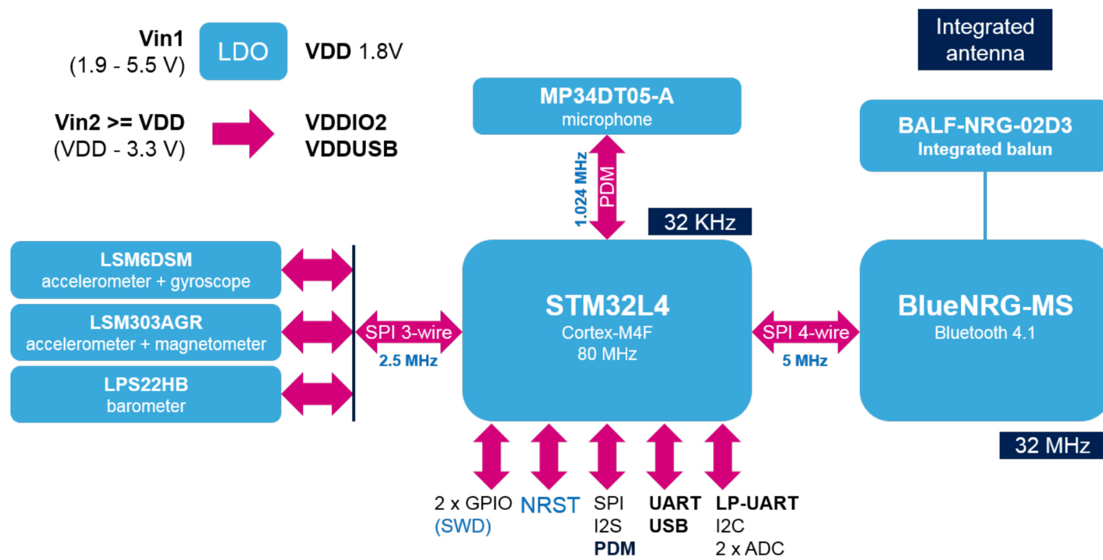


Figure 3.4: Block diagram of the SensorTile board.

- **HTS221:** Capacitive digital sensor for relative humidity and temperature;
- **LDK120M-R:** 200 mA low quiescent current very low noise Low-DropOut (LDO) regulator;
- **STC3115:** Gas gauge Inter Integrated Circuit (I^2C) with alarm output for handheld applications;
- **USBLC6-2P6:** ElectroStatic Discharge (ESD) Protection for USB 2.0 High Speed;
- Mini-B USB connector for power and communication;
- micro Secure Digital (microSD) card slot;
- SWD connectors for programming and debugging.

3.1.2 Orlando

The hardware architecture of the Orlando chip is represented in Figure 3.6. The chip is very flexible and it integrates:

- An on-chip reconfigurable data-transfer fabric to improve data reuse and reduce on-chip and off-chip memory traffic.

Figure 3.5: Additional board.

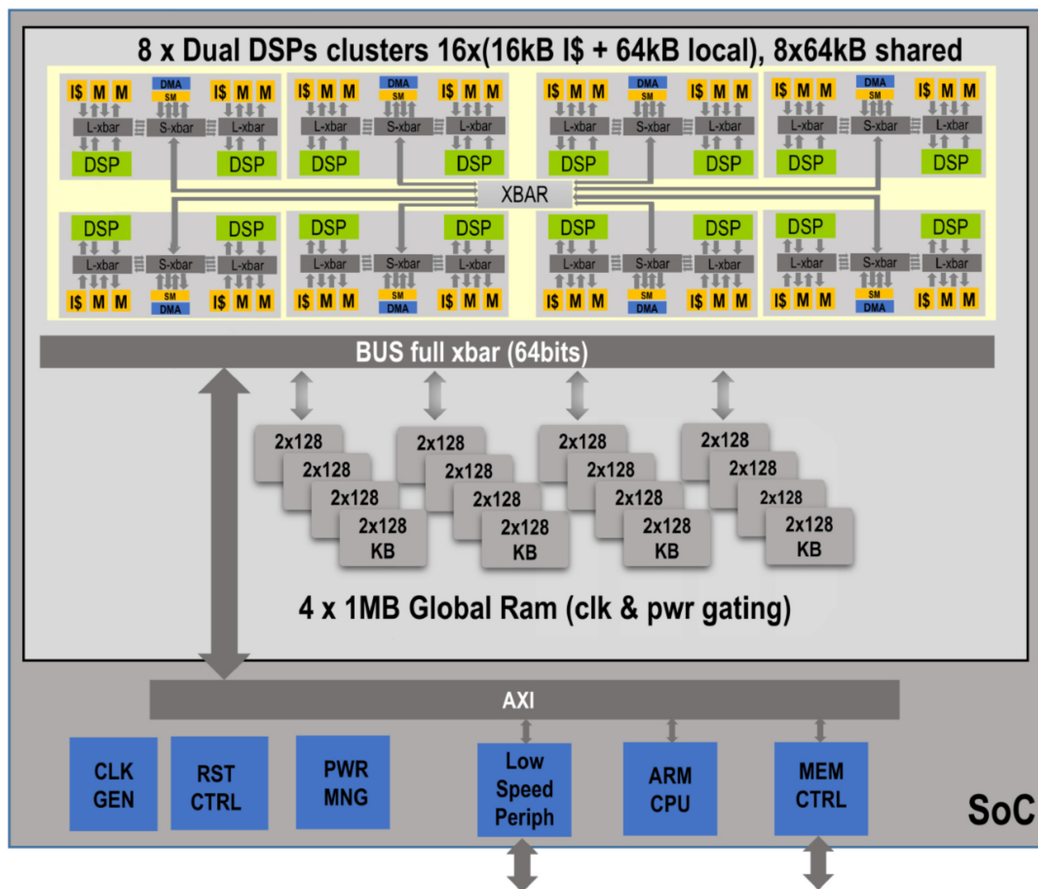


Figure 3.6: SoC top-level block diagram.

- An ARM-based host subsystem with peripherals.
- A range of high-speed Input/Output (IO) interfaces for imaging and other types of sensors.
- A chip-to-chip multi-link to pair multiple devices together.
- A power-efficient array of Digital Signal Processors (DSPs) to support complete real-world computer vision applications. Eight DSP clusters are present, each composed by 2 DSPs, 4-way 16 kB instruction caches, 64 kB local RAMs and a 64 kB shared DSP.
- 4×1 MB Static Random Access Memory (SRAM) banks.

As far as our work is concerned, attention will be focused on DSP cores. As architecture knobs available for dynamically changing the platform setup, we consider activation and de-activation of processing elements (DSP cores), and changes to system frequency and supply voltage. Being still in the prototyping phase, the Orlando board has not been subjected to normal post-production testing. Therefore, under the guidance of the manufacturer, we made an empirical investigation to characterize the device. Several experimental tests have been made to characterize the relationship between the supply voltage of the chip V_{dd} and the system frequency. Therefore, a lookup table has been obtained that can be consulted at any time by the ADAM, in order to correctly set the minimum power supply voltage necessary for the required system frequency. The characterisation process therefore involves finding the minimum voltage for each selected system frequency during an intense and prolonged workload. Figure 3.7 shows the output of the characterization process, for illustration purposes, the curve that approximates the trend of the supply voltage in relation to the system frequency is also shown.

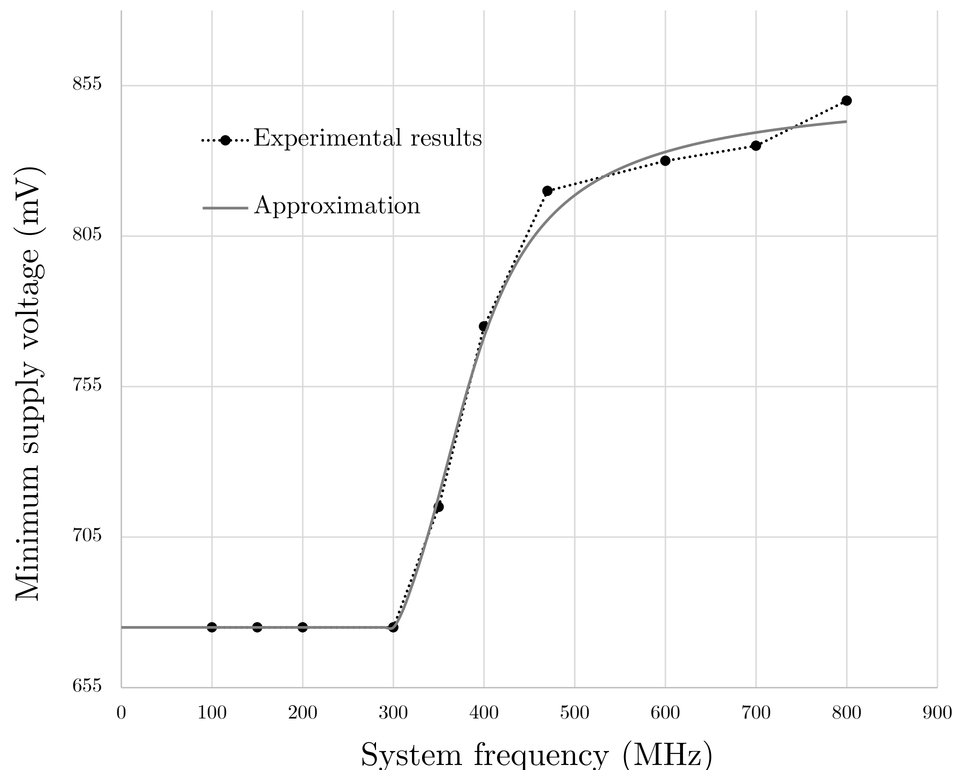


Figure 3.7: V_{dd} measurement and approximation for each system frequency.

3.2 Middleware & Operating System

In addition to the APIs offered by the manufacturer, we used other middleware components to manage multiple computation tasks at runtime and to execute CNN-based near-sensor processing with an adequate performance level.

3.2.1 Common Microcontroller Software Interface Standard

In order to be capable of executing in-place processing of the sensed data, we have exploited the CMSIS libreries, an optimized library specifically targeting Cortex-M processor cores[1]. It includes several modules having many libraries capable

[1] https://github.com/ARM-software/CMSIS_5

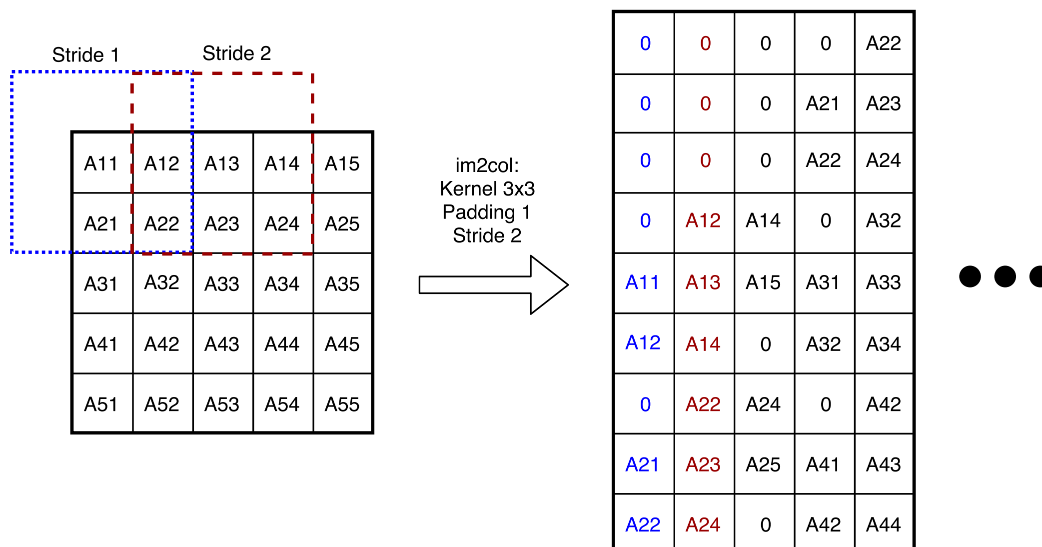


Figure 3.8: Example of im2col on a 2D image with a 3x3 kernel, padding size of 1 and stride size of 2 [1].

of optimizing mathematical functions based on the type of architecture used. Of particular interest is the CMSIS-NN module, inside there are various optimized functions that allow cognitive computational implementations.

NN models are traditionally trained using 32-bit floating point data representation. However, such precision is not usually required during inference. According to research, NNs perform well even with low-precision fixed-point representation [93, 94]. Quantization greatly reduces both network execution time and memory footprint, which is very important when the network is executed on resource-constrained devices.

A CPU-based convolution implementation is typically decomposed into input reordering and expanding (i.e. im2col, image-to-column) and matrix multiplication operations. im2col is a method for converting image-like input into columns that represent the data required by each convolution filter. Figure 3.8 shows an example. Since the pixels in the input image are repeated in the im2col output matrix, one of the main challenges with im2col is the increased memory footprint. CMSIS implement a partial im2col for convolution kernels to alleviate the memory footprint issue while retaining the performance benefits of im2col. The kernel will only expand a limited number of columns (e.g. 2), which is sufficient to obtain the maximum performance boost from matrix-multiplication kernels while minimizing memory overhead.

The image data format can also have an impact on convolution performance,

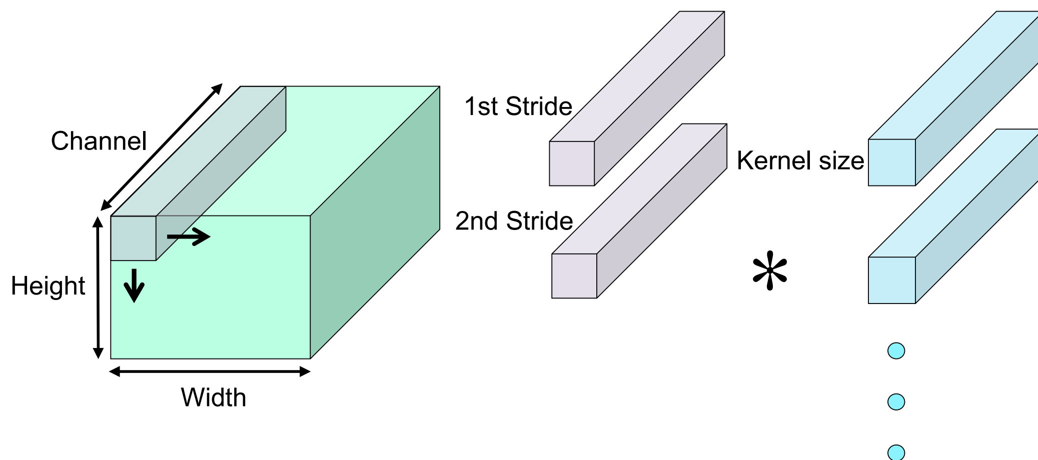


Figure 3.9: Convolution of three-dimensional (height, width, and channel) data [1].

particularly im2col efficiency [95]. The convolution operation is a 2-Dimensional (2D) convolution on 3-Dimensional (3D) data with a batch size of one, as shown in Figure 3.9. Channel-Width-Height (CHW), which means channel last, and Height-Width-Channel (HWC), which means channel first, are the two most common image data formats. The dimension ordering is the same as the data stride ordering. The data along the channel is stored with a stride of 1, the data along the width with a stride of the channel count, and the data along the height with a stride of (channel count $\times$ image width). As long as the dimension order of both weights and images is the same, the data layout has no effect on matrix-multiplication operations. Only the width and height dimensions are used for im2col operations. Because data for each pixel is stored contiguously and can be copied efficiently with Single Instruction Multiple Data (SIMD) support, the HWC-style layout allows for efficient data movement. In Lai et al. [1], the authors implement both CHW and HWC versions and compare the runtime on a Cortex-M7 processor-core to validate this. The results are shown in Figure 3.10, where the HWC input was fixed at $16 \times 16 \times 16$ and the number of output channels was swept. When the output channel value is zero, the software performs only im2col operations and no matrix-multiplication operations. In comparison to the CHW layout, HWC has a shorter im2col runtime while maintaining the same matrix-multiplication performance. With this result, the authors justify the choice of integrating the HWC-style into the CMSIS libraries.

In addition to optimizations concerning convolutional layers, In Lai et al. [1] the authors show software optimizations in CMSIS that are implemented for ma-

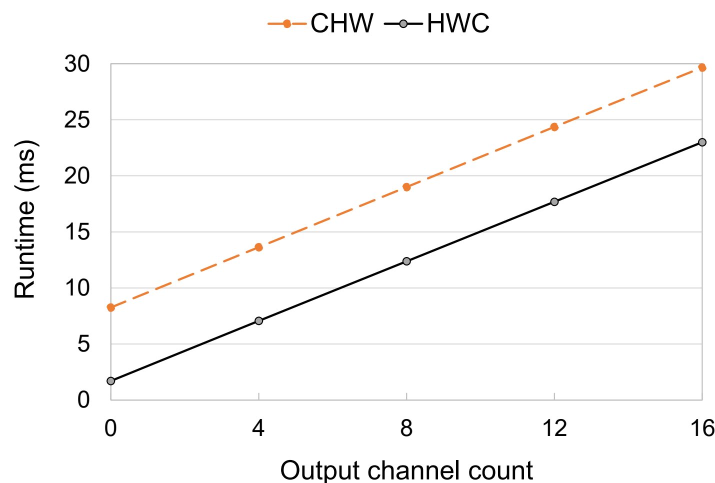


Figure 3.10: Experiment with different data layouts, CHW and HWC. The matrix-multiplication runtime is the same for both data layout styles. HWC has a shorter im2col runtime [1].

trix multiplication, layer pooling and the Rectified Linear Unit (ReLU) activation function. While CMSIS provides quite extensive support for neural network execution, we had to add some changes to support the use cases that are described in the following, namely to enable 1-Dimensional (1D) convolutions on 1D sensor data streams.

3.2.1.1 CMSIS modifications

This shows in detail the software modifications made to the CMSIS libraries in order to use them correctly with our system. In particular, four functions were modified, the function that multiplies inputs with kernels, the convolution layer, the fully connected layer and the max pooling layer. The first three functions have been modified, given the lack of support for scaling the output of the layers by any floating number. CMSIS in fact supports output calculation with a shift of n bits, in other words, the division of the output by 2^n. The max pooling layer has been modified as it is impossible to manage non-square kernels and therefore unusable in the case of one-dimensional input. Regarding the support of one-dimensional inputs, no modification was necessary for the matrix calculation, convolution and fully connected functions since one of the two input dimensions can be fixed at a value of 1.

arm_nn_mat_mult_kernel_q7_q15(). This function executes a matrix calculation between the inputs and the kernels. The original source code can be found at the link[2]. The parts of the code we have rewritten are shown in Listing 3.1.

```
51 q7_t      *arm_nn_mat_mult_kernel_q7_q15_div(const q7_t * pA,
52                                               const q15_t * pInBuffer,
53                                               const uint16_t ch_im_out,
54                                               const uint16_t numCol_A,
55                                               const uint16_t bias_shift,
56                                               const float out_div,
57                                               const q7_t * bias,
58                                               q7_t * pOut)
         ...
77         q31_t      sum =  (q31_t)(*pBias) << bias_shift;
78         q31_t      sum2 = (q31_t)(*pBias++) << bias_shift;
79         q31_t      sum3 = (q31_t)(*pBias) << bias_shift;
80         q31_t      sum4 = (q31_t)(*pBias++) << bias_shift;
         ...
122        *pOut++ = (q7_t) __SSAT((sum / out_div) + 0.5, 8);
123        *pOut++ = (q7_t) __SSAT((sum3 / out_div) + 0.5, 8);
124        *pOut2++ = (q7_t) __SSAT((sum2 / out_div) + 0.5, 8);
125        *pOut2++ = (q7_t) __SSAT((sum4 / out_div) + 0.5, 8);
         ...
140        q31_t      sum = ((q31_t)(*pBias) << bias_shift);
141        q31_t      sum2 = ((q31_t)(*pBias++) << bias_shift)
         ...
174        *pOut++ = (q7_t) __SSAT((sum / out_div) + 0.5, 8);
175        *pOut2++ = (q7_t) __SSAT((sum2 / out_div) + 0.5, 8);
```

Listing 3.1: arm_nn_mat_mult_kernel_q7_q15() source code with our modification.

In Listing 3.1, the changes described below have been made.

In Line 56 the input `out_div` of type `const float` has been added, replacing the old input `out_shift` of type `const uint16_t` (16-bit unsigned constant). This new input represents the scaling to be applied to the output.

Form Line 77 to 80 and from Line 140 to 141, the `NN_ROUND(out_shift)`

[2] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Drivers/CMSIS/NN/Source/ConvolutionFunctions/arm_nn_mat_mult_kernel_q7_q15.c#L51-L187

operand has been removed from the sum, its task is to compensate for the error due to the n-bit shifting of the output. The compensation value is equal to 2^{n-1} and is added to the `sum(i)` variable which is of type `q31_t` (32-bit signed). `sum(i)` accumulates the results of the multiplications between input and kernel that occur during convolution and, after each individual kernel overlap, represents a single output not yet scaled.

Form Line 122 to 125 and from Line 174 to 175, the shifting operation equal to the `out_shift`-bit was replaced and a division equal to the `out_div` value was introduced. The `__SSAT` function has the task of avoiding any overflow or underflow phenomena, possibly saturating the value at -128 in the case of underflow or 127 in the case of an overflow. The value of *pOut (8-bit signed) corresponds to the single scaled quantized output.

arm_status arm_convolve_HWC_q7_basic_nonsquare() It is an optimized two-dimensional convolutional layer, the input is HWC-style, as discussed in the previous paragraphs, is the best choice in Cortex-M processors in terms of execution time. The original source code can be found at the link[3]. The parts of the code we have rewritten are shown in Listing 3.2.

```
67  arm_status arm_convolve_HWC_q7_basic_nonsquare(const q7_t * Im_in,
                                                   ...
71                                                 const q7_t * wt,
    ...
79                                                 const q7_t * bias,
    ...
81                                                 const float out_div,
```

```
                                                   ...
86                                                 q7_t * bufferB)
                    ...
128                 pOut =
129                     arm_nn_mat_mult_kernel_q7_q15(wt, bufferA,
130                                                   ch_im_out,
131                                                   ch_im_in *
132                                                   dim_kernel_y *
      dim_kernel_x, bias_shift, out_div, bias, pOut);
```

[3] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Drivers/CMSIS/NN/Source/ConvolutionFunctions/arm_convolve_HWC_q7_basic_nonsquare.c#L67-L224

```
         ...
149      q31_t      sum = (q31_t)bias[i] << bias_shift;
         ...
179      *pOut++ = (q7_t) __SSAT((sum / out_div) + 0.5, 8);
```

Listing 3.2: arm_convolve_HWC_q7_basic_nonsquare() source code with our modification.

In Listing 3.2, the changes described below have been made.

In Line 132 the previously modified `arm_nn_mat_mult_kernel_q7_q15()` function is called up and the input `out_shift` is changed to `out_div` as well.

Modifications have been made in Lines 81, 149 and 179 similar to those seen for the `arm_nn_mat_mult_kernel_q7_q15()` function.

arm_fully_connected_q7_opt() It is a fully connected layer, the original source code can be found at the link[4]. The parts of the code we have rewritten are shown in Listing 3.3.

```
128 arm_status
129 arm_fully_connected_q7_opt(const q7_t * pV,
130                            const q7_t * pM,
                               ...
134                            const float out_div,
135                            const q7_t * bias,
         ...
154      q31_t      sum =  (q31_t)(*pBias++) << bias_shift;
155      q31_t      sum2 = (q31_t)(*pBias++) << bias_shift;
156      q31_t      sum3 = (q31_t)(*pBias++) << bias_shift;
157      q31_t      sum4 = (q31_t)(*pBias++) << bias_shift;
         ...
323      *pO++ = (q7_t) (__SSAT((sum / out_div) + 0.5, 8));
324      *pO++ = (q7_t) (__SSAT((sum2 / out_div) + 0.5, 8));
325      *pO++ = (q7_t) (__SSAT((sum3 / out_div) + 0.5, 8));
326      *pO++ = (q7_t) (__SSAT((sum4 / out_div) + 0.5, 8));
         ...
337      q31_t      sum = (q31_t)(*pBias++) << bias_shift;
         ...
```

[4] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Drivers/CMSIS/NN/Source/FullyConnectedFunctions/arm_fully_connected_q7_opt.c#L128-L480

```
367         *pO++ = (q7_t) (__SSAT((sum / out_div) + 0.5, 8));
```

Listing 3.3: arm_fully_connected_q7_opt() source code with our modification.

In Listing 3.3, the changes described below have been made.

Modifications have been made in Lines 134, 154–157, 323–326, 337 and 367 similar to those seen for the arm_nn_mat_mult_kernel_q7_q15() function.

arm_maxpool_q7_HWC() This is a max pooling function, the original source code can be found at the link[5]. For consistency, an HWC-style input has also been chosen here. The parts of the code we have rewritten are shown in Listing 3.4.

```
163 void
164 arm_maxpool_q7_HWC_1D(q7_t * Im_in,
165                     const uint16_t dim_im_in,
166                     const uint16_t ch_im_in,
167                     const uint16_t dim_kernel,
168                     const uint16_t padding,
169                     const uint16_t stride, const uint16_t dim_im_out)
```

```
170 {

172     /* Run the following code for Cortex-M4 and Cortex-M7 */

174     int16_t    i_x;

176     for (i_x = 0; i_x < dim_im_out; i_x++)
177     {
178         /* for each output pixel */
179         q7_t      *target = Im_in + i_x * ch_im_in;
180         q7_t      *win_start;
181         q7_t      *win_stop;

183         if (i_x * stride - padding < 0)
184         {
185             win_start = target;
186         } else
```

[5]https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Drivers/CMSIS/NN/Source/PoolingFunctions/arm_pool_q7_HWC.c#L163-L314

```
187         {
188             win_start = Im_in + (i_x * stride - padding) * ch_im_in;

189         }

191         if (i_x * stride - padding + dim_kernel >= dim_im_in)
192         {
193             win_stop = Im_in + dim_im_in * ch_im_in;
194         } else
195         {
196             win_stop = Im_in + (i_x * stride - padding + dim_kernel)
    * ch_im_in;

197         }

199         memmove(target, win_start, ch_im_in);

201         /* start the max operation from the second part */
202         win_start += ch_im_in;
203         for (; win_start < win_stop; win_start += ch_im_in)
204         {
205             compare_and_replace_if_larger_q7_fixed(target, win_start,
     ch_im_in);

206         }
207     }
208 }
```

Listing 3.4: arm_maxpool_q7_HWC_1D() source code with our modification.

The code in Listing 3.4 has been almost completely rewritten to support one-dimensional input.

In Line 205 the function `compare_and_replace_if_larger_q7_fixed()`, it is a modified version of function `compare_and_replace_if_larger_q7()`, the source code of the original function can be found at the following link[6]. For example, if an input vector has a size of 6 and the max pooling kernel size is 2, the original function will only parse the first four elements of the vector and not all 6 and the output will have a size of 2 elements instead of 3. The code in Listing 3.5 has

[6] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Drivers/CMSIS/NN/Source/PoolingFunctions/arm_pool_q7_HWC.c#L52-L82

been added to the end of the original function to solve the problem, in this way the whole input vector can be filtered.

```
83      cnt = length & 0b11;

85      while(cnt > 0u)
86      {
87        if (target[length - cnt] > base[length - cnt])
88          base[length - cnt] = target[length - cnt];
89        cnt--;
90      }
```

Listing 3.5: compare_and_replace_if_larger_q7() source code with our modification.

Given the optimisation on Cortex M processors, the implementation of CMSIS-NN functions leads to considerable benefits in terms of performance. These libraries exploit the SIMD capabilities of the microcontroller in the best possible way with a little impact on memory resources. However, a very important aspect to be taken into accountis the attention to the management of RAM or flash memory resources, that the target device makes, available according to the chosen convolutional neural network model.

3.2.2 FreeRTOS

SensorTile runs Free Real-Time Operating System (FreeRTOS) as Real-Time Operating System (RTOS). This firmware component is aimed at developers who intend to have a real-time operating system without too much impact on the memory footprint of the application. The size of the operating system is between $4\,kB$ and $9\,kB$. Some features offered by the operating system are real-time scheduling functionality, communication between processes, synchronization, time measurements. One of the most important aspects that led us to choose FreeRTOS is that of having the possibility to enable thread-level abstraction to represent processing tasks to be executed on the platform and to timely manage their scheduling at runtime. FreeRTOS creates a system task called *idle* task, which is set with the lowest possible execution priority. When this task is executed, the system tick counter is deactivated and the microcontroller is put in a sleep state. Due to the priority setting, the idle task is only executed if there are no other tasks waiting to be called by the scheduler.

3.2.2.1 FreeRTOS modifications

FreeRTOS does not support a change of system frequency at runtime. Changing the operating frequency at runtime results in the incorrect operation of timers used, for example, to wake up the system from a sleep period or to execute a previously scheduled task. For example, if the frequency is doubled at runtime, the duration time of a timer is halved. Before sleeping, an RTOS task will specify a time after which it requires "waking". The FreeRTOS kernel measures time using a tick count variable. A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy, allowing the real time kernel to measure time to a resolution that derives from the chosen timer interrupt frequency. The tick frequency of the FreeRTOS system is set at operating system start-up through the define in Listing 3.6, the original code can be seen at the following link[7].

```
62 #define configTICK_RATE_HZ                    ((TickType_t)1000)
```

Listing 3.6: FreeRTOSConfig.h file. FreeRTOS original source code.

The tick duration depends on the operating frequency of the system and is set at the start of the operating system.

We had to modify part of the OS in order to enable system frequency changes without impact on the rest of the functionality. To support dynamic frequency scaling, we added a dynamic update of the variable which expresses the duration of a single tick. When frequency changes dynamically, this parameter must be adapted according to any new value of clock cycle time. The variable in question is called `ulTimerCountsForOneTick`, the source code that manages the timings based on this variable can be found at this link[8]. Listing 3.7 shows the original source code, only the lines relating to the operation of variable `ulTimerCountsForOneTick` will be shown and commented.

```
151 #if( configUSE_TICKLESS_IDLE == 1 )
152   static uint32_t ulTimerCountsForOneTick = 0;
153 #endif /* configUSE_TICKLESS_IDLE */
        ...
530       ulReloadValue = portNVIC_SYSTICK_CURRENT_VALUE_REG + (
    ulTimerCountsForOneTick * ( xExpectedIdleTime - 1UL ) );
```

[7] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Middlewares/Third_Party/FreeRTOS/Source/include/FreeRTOSConfig_template.h#L62

[8] https://github.com/STMicroelectronics/STM32CubeL4/blob/f6877af03b5c8b666b3443f5295cf95ec9c0b15d/Middlewares/Third_Party/FreeRTOS/Source/portable/GCC/ARM_CM4F/port.c

```
    ...
684 ulTimerCountsForOneTick = ( configSYSTICK_CLOCK_HZ /
    configTICK_RATE_HZ ) ;
```

Listing 3.7: port.c file. FreeRTOS original source code.

The following comments will refer to the lines of code in Listing 3.7. In Line 152 the variable `ulTimerCountsForOneTick` is declared. In Line 530 there is an example of the use of the variable `ulTimerCountsForOneTick`. `ulReloadValue` is calculated, which corresponds to the time a task or the system goes into sleep before being woken up by an interrupt. `xExpectedIdleTime` corresponds to the number of idle ticks. In Line 684 shows how the `ulTimerCountsForOneTick` variable is set when the operating system starts up.

3.2.3 Orlando middleware

With the selected hardware platform it's not currently possible to exploit the support of a RTOS. We need to consider tasks in the same chain to be prospectively executed by independent hardware cores communicating through a set/hierarchy of shared memories. The overall management is implemented using platform-specific low-level primitives provided by STMicroelectronics, named Remote Procedure Call (RPC) APIs. We have used them to manage communication and synchronization between the DSPs in the platform and to manage the operating state of the processing elements, setting to sleep mode those that are stalled on input (no input data from FIFO) or output channels (the output FIFO is full), or that are not assigned with a task. We present the RPC APIs that we have used in Table 3.2, concerning functions to turn on and off DSPs, and Table 3.3, concerning functions adopted for synchronizing DSPs while accessing FIFOs in a mutually exclusive way.

When idle, a DSP executes a `rpc_serve()` function. In this way, the core waits from an activation message from other cores, and it is set into a sleep state (through `sleep()` function) until a request is received, activating it again (through `wakeup()` function). The requests are stored in a queue, one per DSPs, stored in the main shared memory (4×1 MB SRAM banks) and served with a round-robin priority scheme.

In Table 3.4, we show the functions usable to send activation messages and to check the execution of the assigned tasks on a remote DSPs.

Function name	***Description***
`sleep()`	It's invoked by the DSP that intends to go to sleep, once invoked the DSP is placed in a low-power state and will remain in this state until it receives a wake-up signal.
`wakeup(...)`	Once this function is invoked, a wake-up signal is sent to the specified core.

Table 3.2: Core activation and deactivation functions.

Function name	***Description***
`mutex_init(...)`	Initialization of the mutual exclusion.
`mutex_lock(...)`	Request mutual exclusion.
`mutex_unlock(...)`	Release mutual exclusion.

Table 3.3: Synchronization functions, which are mainly used to read/write on the same FIFO in a mutually exclusive way.

Function name	***Description***
`rpc_call(...)`	Execute a function passed as an input on a remote processor. From the inputs, it's possible to choose whether `rpc_call` is blocking or non-blocking.
`rpc_check(...)`	Check the execution status of a certain function call on a specific core (non-blocking).
`rpc_wait(...)`	Wait for the conclusion of a function on a specific core (blocking).

Table 3.4: Call functions, used by the ADAM system to manage the execution of tasks on the cores.

3.3 PyTorch framework & quantization

PyTorch is an open source machine learning library based on the Torch library, primarily developed by Facebook's Artificial Intelligence Research (FAIR) labs for applications such as computer vision and natural language processing. It is open-source software distributed under the Modified Berkeley Software Distribution (BSD) license. PyTorch offers various quantization tools, which, as we have seen in Section 2.2.1, 2.4 and 3.2.1, lead to advantages in terms of execution time and memory footprint.

Compared to typical Floating-Point 32 models, PyTorch supports Integer 8 quantization, which allows for a 4x reduction in model size and a 4x reduction in memory bandwidth requirements. When compared to FP32 computations, hardware support for INT8 computations is typically 2 to 4 times faster. PyTorch supports a variety of methods for quantizing a deep learning model. In most cases, the model is trained in Floating-Point 32 before being converted to Integer 8 (quantization post-training). Furthermore, PyTorch supports quantization aware-training, which uses fake-quantization modules to model quantization errors in both forward and backward passes[9].

The method we have chosen is a post-training quantization called static quantization[10]. Post-training static quantization entails not only converting the weights from float to int, as in dynamic quantization, but also running batches of data through the network and computing the resulting distributions of the various activations (specifically, this is done by inserting observer modules at different points that record this data). These distributions are then used to determine how the various activations should be quantized at inference time. We have chosen to insert a *MinMax* observer[11]. This observer uses the min/max statistics of the tensor for each layer to calculate *scale* and *zero-point* values, used for weights and bias conversion from float to int precision. the quantization parameters, minimum/-maximum $x_{\mathrm{min/max}}$ is computed in Equation 3.1 where X is the observed tensor.

[9] https://pytorch.org/docs/stable/quantization.html
[10] https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html
[11] https://github.com/pytorch/pytorch/blob/master/torch/ao/quantization/observer.py#334

$$
\begin{aligned}
x_{\min} &= \begin{cases} \min(X) & \text{if } x_{\min} = \text{None} \\ \min\left(x_{\min}, \min(X)\right) & \text{otherwise} \end{cases} \\
x_{\max} &= \begin{cases} \max(X) & \text{if } x_{\max} = \text{None} \\ \max\left(x_{\max}, \max(X)\right) & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.1}
$$

Once the X value is calculated, *scale* and *zero point* values are computed in Equation 3.2 where $Q_{\min}$ and $Q_{\max}$ are the minimum and maximum of the quantized data type. The symmetrical case has been chosen, so the *zero point* value is forced to zero.

$$
\begin{aligned}
\mathit{scale} &= 2\max(|x_{\min}|, x_{\max}) / \left(Q_{\max} - Q_{\min}\right) \\
\mathit{zero\ point} &= \begin{cases} 0 & \text{if dtype is qint8} \\ 128 & \text{if dtype is quint8} \end{cases}
\end{aligned}
\tag{3.2}
$$

In addition, activations were also forced to zero during the procedure.

Referring to the Listing 3.2 shown in Section 3.2.1.1, in Line 71 `wt` are the quantized weights, in Line 79 `bias` is equivalent to a vector of zeros since the activations are all null and in Line 81 `out_div` is equivalent to a *scale* value.

Referring to the Listing 3.3 shown in Section 3.2.1.1, in Line 130 `*pM` is the pointer to quantized weights, in Line 134 `out_div` is equivalent to a *scale* value and in Line 135 `bias` is equivalent to a vector of zeros since the activations are all null.

3.4 Electrocardiogram configuration

The heart is actually two separate pumps: a right heart that circulates blood through the lungs and a left heart that circulates blood through the peripheral organs. Each of these hearts, in turn, is a pulsatile two-chamber pump made up of an atrium and a ventricle. Each atrium functions as a weak primer pump for the ventricle, assisting in the movement of blood into the ventricle. The ventricles then provide the primary pumping force that propels blood either through the pulmonary circulation (right ventricle) or through the peripheral circulation (left ventricle). Special mechanisms in the heart cause a continuous succession of heart contractions known as cardiac rhythmicity, which is caused by action potentials transmitted throughout the cardiac muscle and results in the heart's rhythmical beat. When the cardiac impulse passes through the heart, an electrical current also spreads from the heart into the adjacent tissues surrounding the heart. A small

portion of the current spreads all the way to the surface of the body. If electrodes are placed on the skin on opposite sides of the heart, electrical potentials generated by the current can be recorded; the recording is known as an ECG [96].

The electrocardiogram is a special graph that represents the electrical activity of the heart from one instant to the next. Thus, the ECG provides a time-voltage chart of the heartbeat. For many patients, this test is a key component of clinical diagnosis and management in both inpatient and outpatient settings. The device used to obtain and display the conventional ECG is called the electrocardiograph, or ECG machine. It records cardiac electrical currents (voltages or potentials) by means of conductive electrodes selectively positioned on the surface of the body. For the standard ECG recording, electrodes are placed on the arms, legs, and chest wall (precordium). In certain settings (emergency departments, cardiac and intensive care units, and ambulatory monitoring), only one or two "rhythm strip" leads may be recorded, usually by means of a few chest electrodes. The 12 standard ECG leads (connections or derivations) are commonly used to record these voltages from the heart. The leads actually show the differences in voltage (potential) between electrodes placed on the body's surface [97].

The leads are divided into two groups: limb (extremity) leads and chest (precordial) leads. The six limb leads I, II, III, aVR, aVL, and aVF use electrodes on the extremities to record voltage differences. Based on their evolutionary history, they can be divided into two subgroups: three standard bipolar limb leads (I, II, and III) and three augmented unipolar limb leads (aVR, aVL, and aVF). The six chest leads V1, V2, V3, V4, V5, and V6 use electrodes placed at various locations on the chest wall to record voltage differences [97].

Named after the theorist and father of modern electrocardiography, the Einthoven Triangle (shown in Figure 3.11[12]) is based on the imaginary arrangement of an inverted equilateral triangle on the patient's chest, the center of which coincides with the heart. Each corner of the geometric figure is electrically coincident with a point on a specific limb which is given a name: Right Arm (RA), Left Arm (LA) and Left Leg (LL). The calculation of leads I, II and III is shown in Equations 3.3, 3.4 and 3.5 respectively.

$$Lead\ I = LA - RA\,, \tag{3.3}$$

$$Lead\ II = LL - RA\,, \tag{3.4}$$

$$Lead\ III = LL - LA\,. \tag{3.5}$$

There are different types of electrode placement, the method we have chosen

[12] https://www.nottingham.ac.uk/nursing/practice/resources/cardiology/function/bipolar_leads.php

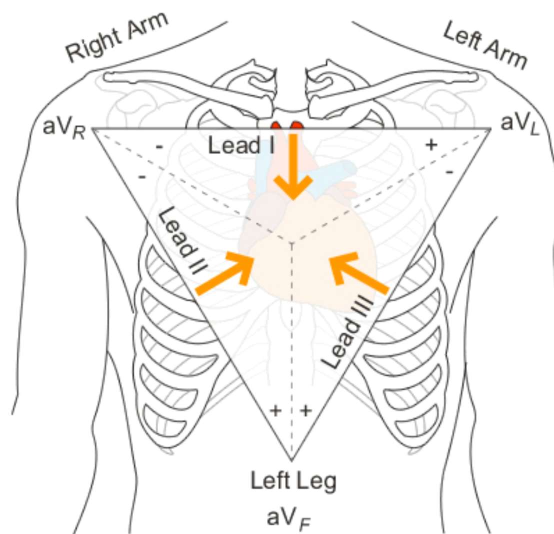


Figure 3.11: Einthoven Triangle.

is called Modified Limb Lead (MLL), Figure 3.12 shows the consistent placement of electrodes according to this method. As we will see later, only lead II will be taken into consideration in the thesis work and we will call it Modified Limb Lead II (MLLII). This lead is the same one used to generate the signals contained in the dataset we have chosen for our studies, Section 3.4.1 will describe the dataset in more detail.

3.4.1 MIT−BIH Arrhythmia Database

Since 1975, research on arrhythmia analysis and related subjects has been conducted in the laboratories at Beth Israel Hospital (BIH) in Boston (now Beth Israel Deaconess Medical Center) and Massachusetts Institute of Technology (MIT). The MIT−BIH Arrhythmia Database [98], which was completed and distributed in 1980, was one of the first major products of this effort. The database was the first widely available set of standardized test material for evaluating arrhythmia detectors, and it has been used for this purpose, as well as basic cardiac dynamics research, at over 500 sites around the world. The ECGs in the MIT−BIH Arrhythmia Database were obtained from a collection of over 4000 long-term Holter recordings obtained by the Beth Israel Hospital Arrhythmia Laboratory between 1975 and 1979. Around 60% of these recordings were obtained from inpatients. The database includes 23 records (numbered 100 to 124 inclusive, with some numbers missing) chosen at random from this set, as well as 25 records (numbered

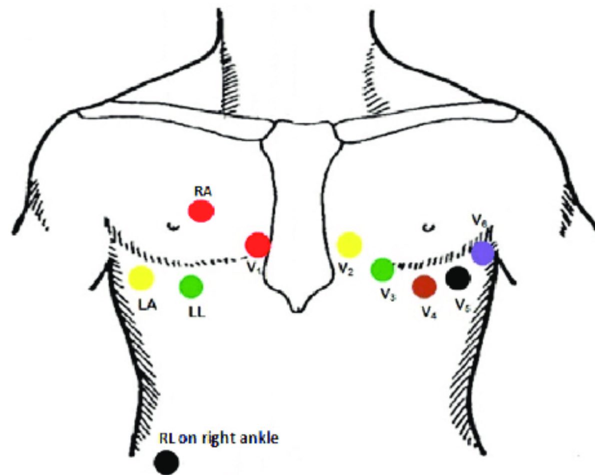


Figure 3.12: Electrodes placement according to the modified limb lead method [2].

200 to 234 inclusive, with some numbers missing) chosen from the same set to include a variety of rare but clinically important phenomena that would not be well-represented by a small random sample of Holter recordings. Figure 3.13 shows an example of an ECG trace taken from the dataset. Each of the 48 records lasts slightly more than 30 minutes. The full database has been available since February 2005 on PhysioNet [99].

The subjects were 25 men aged 32 to 89 years, and 22 women aged 23 to 89 years. In most records, the upper signal is a MLLII, obtained by placing the electrodes on the chest. The lower signal is usually a modified lead V1 (occasionally V2 or V5, and in one instance V4); as for the upper signal, the electrodes are also placed on the chest. This configuration is routinely used by the BIH Arrhythmia Laboratory.

3.5 Application model & ADAptive runtime Manager

We selected an application structure based on process networks. Tasks are represented as independent processes, communicating with each other via FIFO structures, using blocking *read* and *write* communication primitives to avoid data loss in case of busy pipeline stages. Processes may be potentially executed in parallel, in case of available processing resources, potentially improving performance using

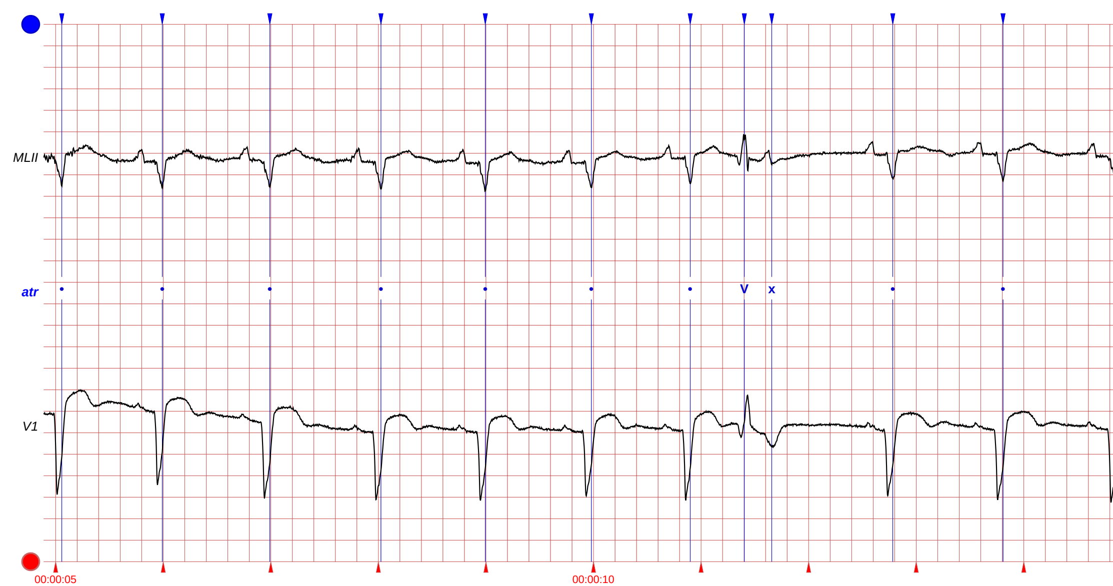


Figure 3.13: 10-second ECG trace, recording number 108 in the MIT−BIH Arrhythmia dataset. In the upper part is present the MLLII trace, in the middle part the notes and in the lower part the modified lead V1 trace.

a software pipeline.

In particular, for each sensed variable to be monitored, we build a chain of tasks that operate on the sensed data (Figure 3.14).

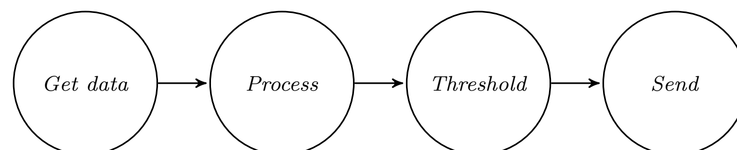


Figure 3.14: Simple task chain.

A chain of processes is generated for each sensor node, so that, if required by changes in the operating mode, it's possible to dynamically turn on and off the useful and non-useful components.

For each sensor, we envision four types of general tasks:

- *Get data task:* takes care of taking data from the sensing hardware integrated into the node.
- *Process task:* it's possible to have multiple tasks of this type, representing multiple stages of in-place data analysis algorithm. Having more than one

task of this type allows a prospective user to select, for example, a certain depth of analysis, which determines an impact on the required communication bandwidth, detail of the extracted information, and power/energy consumption.

- *Threshold task:* this task allows to filter data depending on the results of the in-place analysis. For example, a threshold task may be used to send data to the cloud only when specific events or alert conditions are detected. Its purpose is to limit data transfers from the node.
- *Send task:* is the task in charge of outwards communication to the gateway.

Considering the selected process network model, activation/deactivation of tasks or entire chains corresponding to sensors can be implemented by:

- enabling/stopping the periodic execution of the involved task;
- reconfiguring the FIFOs to reshape the process chain accordingly.

In this way, it's possible to select multiple application configurations, corresponding to operating modes characterized by different levels of in-place computing effort, bandwidth requirements, monitoring precision.

Within the process network, a task was exclusively dedicated to the management of dynamic hardware and software reconfiguration of the platform. We have implemented such reconfiguration in a software agent called ADAptive runtime Manager. ADAM can be activated periodically by means of an internal timer. It evaluates the status of the system, monitoring:

- reconfiguration commands from the gateway;
- changes in the workload, e.g. rate of events to be processed. For example, a task may have to be executed periodically, with a rate that depends on the frequency of certain events in the sensor data. This poses real-time constraints that may be varying over time in a data-dependent manner.
- other relevant variables (e.g. battery status).

Depending on such input, ADAM can react to change the platform settings, performing different operations:

- Enable or disable the individual tasks of the sensor task chain or the entire chain;

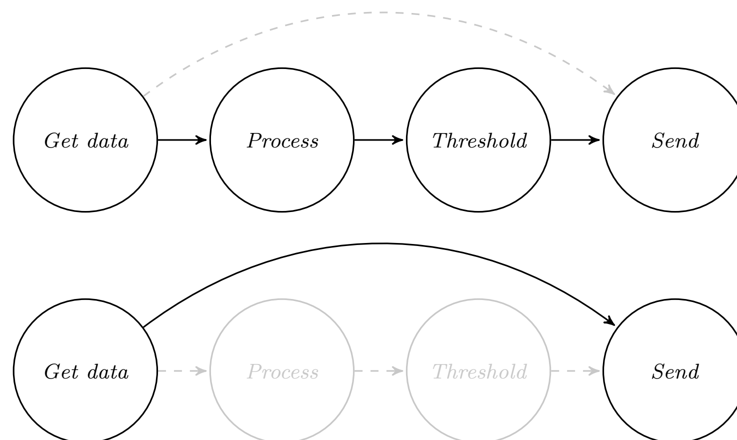


Figure 3.15: Two possible configurations of a generic system.

- Choose whether to set the microcontroller in a sleep mode or not;
- Set the operating frequency of the microcontroller to increase/reduce performance level;
- Reroute the data-flow managed by the FIFOs according to the active tasks.

Figure 3.15 shows an example of the reconfiguration of the system that may be applied by ADAM, deactivating a *process* task, to switch from an operating mode that sends pre-processed information to the cloud to another sending raw data.

This proposed application model can be easily declined in different use cases and application domains involving sensor monitoring and near-sensor processing. Adapting to new scenarios could require for example to add different or additional *process data* tasks and/or connecting them in a different order. Nevertheless, the possibility of switching between different configurations by reconfiguring the process network would be preserved.

3.5.1 ADAptive runtime Manager algorithm

This section will describe in detail what the ADAptive runtime Manager algorithm does during its execution. Algorithm 3.1 is easily adaptable to single-core devices such as the one we use. In Chapter 6 we will explain how ADAM features have been integrated in order to better manage workloads based on neural network processing on multi-core platforms.

initialise & suspend all threads, Line 1. All the threads associated with the different tasks at the operating system level are defined. The main parameters to be set for each thread during this phase are: assigning a job, giving a higher or lower execution priority and allocating a certain amount of stack memory. Once all the threads have been initialised, ADAM preventively blocks the execution of all the threads while waiting for a configuration message. It is executed only once at system start-up.

send message (parameters setup), Line 2. As described above, all threads have a dedicated input FIFO and are in a phase of waiting for data to be processed. With this command, ADAM sends a message to itself regarding the configuration of generic components. This configuration is carried out only once at system start-up and is predefined by the user.

send message (task), Line 3. Similar to Line 2, ADAM sends a message to itself with regard to the configuration of the threads. The configuration is strictly dependent on the operating mode decided at run time by the user, either directly or, as in this case, indirectly. In fact, the user establishes an initial configuration determining which operating mode is to be executed at start-up. Obviously, a possible configuration is equivalent to a quiescent system, which does not perform any work but waits for an external reconfiguration message.

This configuration is carried out only once at system start-up and is predefined by the user.

wait internal or external message, Line 5. At this point, the system has entered an infinite loop in order to handle internal or external messages. The first action that ADAM takes once it has entered the loop is to remain dormant, waiting for a reconfiguration message to be input to its FIFO.

The first time this command is reached there are already two incoming messages, namely those described in Lines 2 and 3.

switch *message*, Line 6. The system sorts incoming messages and makes related decisions. Messages can have three different labels: *parameters setup*, *task setup* and *periodic check*.

A "parameters setup" message involves updating the tabulated configurations, for all operating mode/sensor pair, of SoC peripherals, sensor configuration, general-purpose timers timing and thread wake-up timers timing if present.

A "task setup" message involves reading the tabulated configurations, for the operating mode (specified in the reconfiguration message) of the ith sensor. The reconfiguration concerns the SoC peripherals, the sensor configuration, the timing of the general purpose timers and those that wake up the threads if present. On the basis of the tabulated configuration read, the system is correctly reconfigured and the operating mode is changed.

A "periodic check" message is a message that the ADAM system receives periodically through the use of one of the general purpose timers set at Line 9. This message allows the ADAM system to monitor the status of the system and possibly make changes to it.

SoC peripherals setup, Line 8 and 20. Reading/writing parameters of a generic nature regarding the devices integrated in the SoC, e.g. changing specifications of the BLE stack.

timers setup, Line 9, 21, 14 and 26. Reading/writing the timing parameters of general purpose timers (Line 9, 21), e.g. the timer that manages the periodic wake-up of ADAM or the timer that wakes up the BLE routine that manages the service. Or, reading/writing the timing parameters of threads timers (Line 14, 26), e.g. the timer that periodically runs the thread dealing with the sampling of ith sensor data.

sensors setup, Line 11 and 24. Reading/writing of sensor parameters. This function initiates user-specified routines, such as an Serial Peripheral Interface (SPI) read/write to the configuration registers of the nth sensor according to the user or operating mode requirements.

check operating mode, Line 22. Check whether a change of operating mode has been requested. If not, do not proceed with the `case` statement. However, if the operating mode change is requested, an "operating mode index" is specified for each sensor involved.

threads setup, Line 27. Consistent with the initialization in Line 1, this function enables or disables threads as required by the operating mode of the ith sensor.

FIFOs setup, Line 28. Once the topology of the process network has been changed, the data exchanged between the various threads must be routed. That

is, each thread has to route the produced data to the right FIFO of the next thread, according to the selected operating mode of the ith sensor.

custom action, Line 34. If defined, custom actions are executed under custom conditions.

check workload, Line 40. The workload is calculated as a percentage. When the system starts up, two timers are triggered, the first timer advances unconditionally, the second timer advances only if the system is not in sleep mode. Based on a user-defined time window, the workload percentage is calculated as the ratio of the second timer to the first one.

frequency & voltage regulator, Line 41. The system frequency and power supply voltage change depends on the workload and real time constraints. The real time constraints are set so as to avoid overlapping tasks during a succession of events that trigger the process chain, even though FIFOs would be able to handle these cases without data loss. The possible values of frequency and voltage are tabulated and their variation is sensitive to ranges of workload and not to any minimum variation of it. This is done for two reasons: firstly, to avoid overloads due to numerous system demands related to changing system frequency/supply voltage; secondly, not all devices support fine-grained operating frequency tuning. In addition to the workload ranges, the user must also set a parameter called patience, which determines how long the system will not accept any further requests for frequency and voltage changes after it has just received one. It will again be at the user's discretion to choose intervals that are first of all usable and do not create constant, short-term changes in the system configuration.

```
1  initialise & suspend all threads;
2  send message (parameters setup);
3  send message (task setup);
4  while true do
5      wait internal or external message;
6      switch message do
7          case parameters setup
8              SoC peripherals setup;
9              timers setup;
10             for s ← 1 to number of sensors do
11                 sensors setup [ s ];
12                 for o ← 1 to number of operating mode do
13                     for t ← 1 to number of threads do
14                         timers setup [ s ][ o ][ t ];
15                     end for
16                 end for
17             end for
18         end case
19         case task setup
20             SoC peripherals setup;
21             timers setup;
22             check operating mode;
23             for s ← 1 to number of sensors do
24                 sensors setup [ s ][ operating mode index ];
25                 for t ← 1 to number of threads do
26                     timers setup [ s ][ operating mode index ][ t ];
27                     threads setup [ s ][ operating mode index ][ t ];
28                     FIFOs setup [ s ][ operating mode index ][ t ];
29                 end for
30             end for
31         end case
32         case periodic check
33             if custom condition then
34                 custom action;
35             end if
36             check battery level;
37             if battery level < threshold then
38                 send message (task setup)
39             end if
40             check workload;
41             frequency & voltage regulator [ workload range (workload) ];
42         end case
43     end switch
44 end while
```

Algorithm 3.1: ADAM algorithm.

4 Cardiovascular disease detection on electrocardiogram

To implement detection of cardiac abnormalities on ECGs, we have applied the previously described application model to deploy an adequate waveform analysis application on SensorTile. A single-lead configuration was chosen due to the practical convenience of not having too many electrodes attached to the skin. Lead II is often read individually and therefore adequate in case that as little data as possible needs to be processed. In particular, the MLLII configuration was chosen, the same configuration present in the records of the MIT−BIH arrhythmia database, these recordings will be taken into consideration during the training. We built a prototype using an AD8232[1] sensor module from Analog Devices, connected to the Analog to Digital Converter (ADC) integrated into the reference platform. The device supports a single-lead configuration, with two or three electrodes. The official documentation proposes different solutions to manage the signal noise, the compromise is between signal distortion and rejection of motion artifacts. The chosen configuration assumes that the patient remains relatively still during the measurement, and therefore, motion artifacts are less of an issue. The AD8232 is configured with a 0.5 Hz two-pole high-pass filter fol-

[1] https://www.analog.com/media/en/technical-documentation/data-sheets/ad8232.pdf

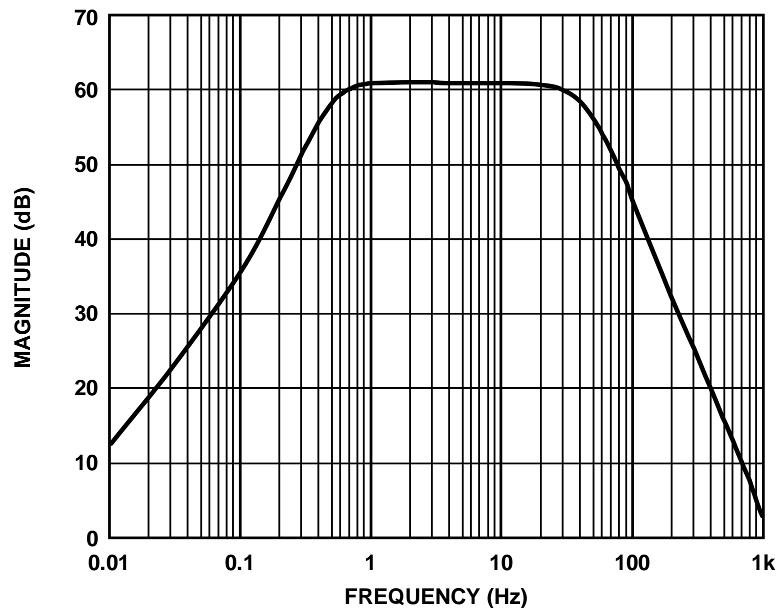


Figure 4.1: Frequency response of the circuit configuration chosen for the AD8232.

lowed by a two-pole, 40 Hz, low-pass filter. A third electrode is driven for optimum common-mode rejection.

In this section, we describe the supported operating modes, that can be selected at runtime, and the processing tasks coexisting in the different operating modes. It is also possible to find the original project in the repository[2].

4.1 Operating modes

We have enabled three different operating modes to be selectable by the user, by sending adequate commands from the cloud. Operating modes are shown in Figure 4.2.

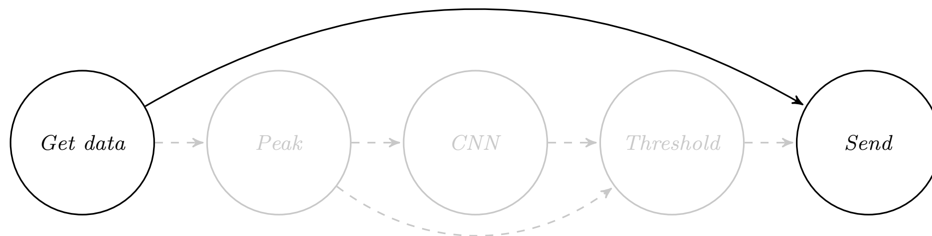
4.1.1 Operating mode: Raw data

The first operating mode envisions sending the entire data stream acquired by the sensor node to the gateway. There is therefore no near-sensor data analysis enabled, and it poses fairly high requirements in terms of bandwidth. In this operating mode are:

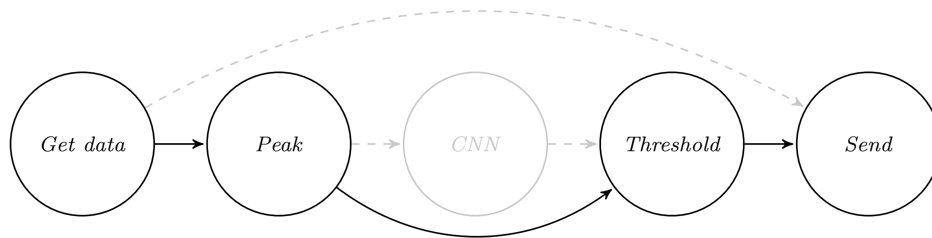
- Multiple samples are been grouped and inserted into a packet of 20 Bytes (8 ECG data 16 bit, 1 timestamp 32 bit).

[2] https://github.com/matteoscrugli/adam-iot-node-on-stm32l4

Operating mode: Raw data.



Operating mode: Peak detection.



Operating mode: CNN processing.

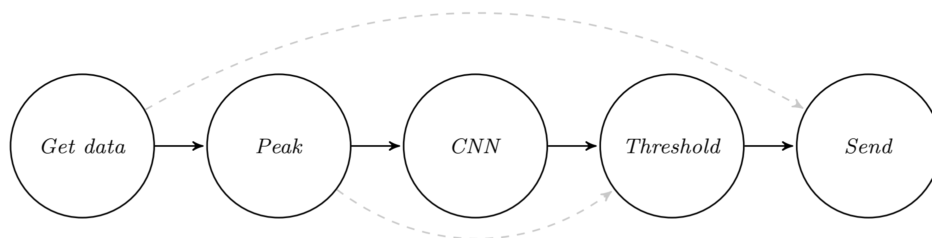


Figure 4.2: ECG application model.

- The sample rate of the ADC is set to 330 Hz, considering sending multiple samples at a time, one Bluetooth packet is sent every 24 ms.

4.1.2 Operating mode: Peak detection

This operating mode does not provide visual access to the whole ECG waveform. A healthcare practitioner, when selecting this mode when accessing the data, can select to monitor only heartbeat rate, requiring a lower level of detail in the information sent to the cloud. As a common technique to reduce the power consumption related to communication when sampled values are not interesting, a healthcare practitioner could also set thresholds and receive a notification only when thresholds are exceeded. In this operating mode, four tasks are active:

- Get data task
- Process data (peak detection)
- Threshold task (alert heartbeat rate evaluation)
- Send task

This operating mode processes samples to search for signal peaks and consequently computes the heartbeat rate. The first task (Figure 4.2) collects data from the sensor (as in raw data operating mode), the second analyzes the signal analysis and calculates the heart rate, and the fourth allows data transmission. The threshold task is used to determine if data must be sent to the cloud. For example, no data is sent if the heartbeat rate is controlled between two high and low alert values. The peak detection algorithm it's not very critical in terms of time and power consumption, it will be better discussed in Section 4.2. The size of the package sent is 5 Bytes packet (1 heartbeat rate value, represented on 8 bit, 1 time-stamp 32 bit). The transmission rate is given dependent, in the worst case a package is sent for each peak detected. Thanks to the threshold task, the communication-related power consumption is heartbeat-dependent, since the execution of the send task is triggered only when the heartbeat exceeds the preset threshold defined by the medical staff.

4.1.3 Operating mode: CNN processing

In the latter operating mode, a further level of analysis is introduced. An additional task implements a convolutional neural network, classifying the ECG waveform to recognize physically relevant conditions. Using such classification technique, the

practitioner can monitor the morphology of the signal without the need of sending the entire data stream to the cloud, saving transmission-related power/energy consumption. The neural network implemented recognizes anomalous occurrences in the ECG tracing, in this case, communications with the gateway occur only in case of anomaly detection. The enabled tasks are:

- Get data task
- Process data 1 task (peak detection)
- Process data 2 task (CNN)
- Threshold task (anomalous shapes in the ECG waveform)
- Send task

The required communication bandwidth is more similar to *peak detection OM* than *raw data OM*, however, with respect to *peak detection OM*, computing effort is higher. The node executes the 1D convolution neural network similar to the one described in [100]. We have designed the system to be capable of classifying ECG peaks according to alternative sets of categories, each composed by 5 classes, named *NLRAV* and *NSVFQ* (see Figure 4.5). The design process used to select, train and deploy the specific neural network topology is explained in Section 4.3.

The size of the data transferred to the cloud is 6 Bytes (1 heartbeat data 8 bit, 1 label data 8 bit, 1 timestamp 32 bit). The CNN, threshold and send tasks are executed only if a peak is detected, the activation frequency of these tasks, therefore, depends directly on the heart rate value.

4.2 The peak detection algorithm

The processing of the ECG signal is activated in *peak detection* and *CNN OM*, in both operating modes it's necessary to identify the R peaks in the signal, therefore a simplified version of the Pan Tompkins algorithm was used in order to obtain the position of the R peaks during data acquisition from the sensor. The reference study to implement the R peak recognition algorithm is [101].

An exploration was made with different combinations of filters and mathematical functions blocks in order to reduce the computational load as much as possible and at the same time obtain a good level of peak detection accuracy. The accuracy on the MIT−BIH Arrhythmia dataset was validated for each step of the exploration. The Figure 4.3 shows the block diagram representing the signal processing

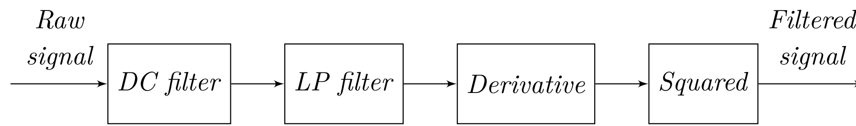


Figure 4.3: Filtering block diagram.

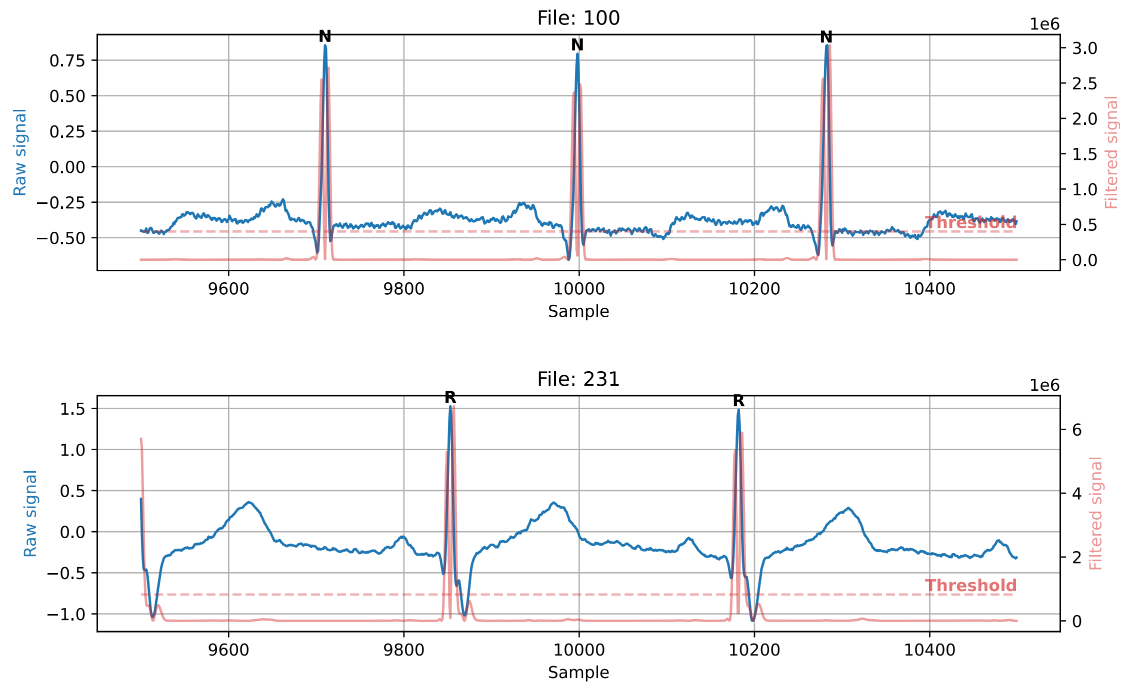


Figure 4.4: Raw signal and filtered one from two different recordings.

algorithm chosen after the exploration, composed by: Direct Current (DC) filter, Low Pass (LP) filter (f_c = 11Hz), derivative and squared block. If CNN OM is enabled, the signal is segmented, a number of samples equal to the input size of the neural network are considered to generate a frame and the detected peak is centered in it.

Figure 4.4 shows the raw signal in blue color and the filtered one in red from two different recordings. A peak is detected when a filtered signal exceeds a predefined threshold, then returns to a local minimum point and the delay introduced by the filter is taken into account, the threshold value may be set differently for each recording.

A detected peak is considered a true positive when it is associated with a dataset peak in a neighborhood of 50 samples within the track under analysis.

Equation 4.1 and 4.2 shows the sensitivity/True Positive Rate (TPR) and precision/Positive Predictive Value (PPV) data of the peak detection algorithm on the MIT−BIH arrhythmia database. In Equation 4.1, Equation 4.2 and in successive equations, the following operands are used: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN)

$$TPR = \frac{TP}{TP + FN} = 0.99674\,, \tag{4.1}$$

$$PPV = \frac{TP}{TP + FP} = 0.99421\,. \tag{4.2}$$

Table 4.1 shows true positive, false positive, false negative of the peak detection algorithm with a tolerance of 50 samples.

False positives are the peaks detected by the algorithm that cannot be matched to any in the dataset. False negatives are the peaks in the dataset that are not detected by the detection algorithm. We report in Table 4.2 the distribution of the type of peaks present in the dataset and in the false positives subset, the NSVFQ labels are reported since they cover all useful types of classes present in the dataset. The extreme case corresponds to analyze an ECG trace containing only V peaks, in this case, 0.97% of peaks are not detected. In literature, there are more advanced real-time algorithms that obtain sensitivity and precision values similar or higher than those obtained with our algorithm, such as in Gupta et al. [102] and Laitala et al. [103]. In Laitala et al. [103], more offline algorithms are shown with higher sensitivity and precision values, the authors state that it is easy to address the same results in the real-time case for their method. Our aim is to find an algorithm that achieves high sensitivity and accuracy values and, at the same time, provided a low computational load and easy integration on the microcontroller.

4.3 Designing the CNN: training and optimization

As seen in Chapter 2, introducing deep learning into this type of device brings numerous benefits. Beyond that, we chose the neural network described in this section for its high accuracy, low computational load, latency, and power consumption. These results will be better described in the dedicated Section 4.4.

We have exploited a training procedure using and comprising a static quan-

File	***TP***	***FP***	***FN***	***File***	***TP***	***FP***	***FN***
100	2273	0	0	201	1957	5	6
101	1865	7	0	202	2135	3	1
102	2187	0	0	203	2929	75	51
103	2084	0	0	205	2653	0	3
104	2219	27	11	207	1832	192	28
105	2559	61	14	208	2936	47	19
106	2025	6	2	209	3003	9	2
107	2134	1	3	210	2639	24	11
108	1639	30	118	212	2746	0	2
109	2531	5	1	213	3248	1	3
111	2123	3	1	214	2256	6	6
112	2538	4	1	215	3363	2	0
113	1794	0	1	217	2202	2	6
114	1877	2	2	219	2153	0	1
115	1953	0	0	220	2048	0	0
116	2391	7	21	221	2426	4	1
117	1535	6	0	222	2482	3	1
118	2278	6	0	223	2605	6	0
119	1987	1	0	228	2033	53	21
121	1861	2	2	230	2256	3	0
122	2476	1	0	231	1571	0	0
123	1518	3	0	232	1778	4	2
124	1619	0	0	233	3072	1	7
200	2599	10	2	234	2753	0	0

Table 4.1: True positives, false positives, and false negatives of our peak detection algorithm with a tolerance of 50 samples for each ECG recording on MIT−BIH arrhythmia database.

Classes	***Classes distribution***		
	Dataset	***FN subset***	***FN subset / Dataset***
N	90369	255	0,28 %
S	2781	8	0,29 %
V	7230	70	0,97 %
F	803	4	0,5 %
Q	3895	7	0,18 %

Table 4.2: Classes distribution over the dataset or the false negative subset.

tization[3] step, the source code is available at our public repository[4]. This process enables the conversion of weights and activations from floating-point to integers and allows to implementation of the CNN using the CMSIS-NN optimized function library, which expects inputs represented with 8-bit precision. In static quantization[2], which takes place right after quantization, *float* values are converted to *qint8* format. We set the procedure to force bias values to be null, while, to quantize weights, *MinMax* observers[5] are inserted inside the network to detect the output values dynamics in each layer. On the basis of the reported distribution, *scale* and *zero-point* values are selected and used to convert effectively and prevent data saturation.

The functions implementing convolution and fully connected layers in the CMSIS-NN library provide for output shifting operations to apply the *scale* factor on the outputs, allowing for scaling values ranging from -128 to 127. The quantization procedure in PyTorch, on the other hand, requires a *scale* value that is not necessarily a power of 2. For this reason, we slightly modified the CMSIS functions to support arbitrary *scale* values. Such modifications have led to a limited increase in the inference execution time. As an example of such performance degradation, we report here the execution time increase for two examples CNN topologies, named *20_20_100* and *4_4_100* networks (network name indicates the main topology parameters as *conv1OutputFeatures_ conv2OutputFeatures_ fc1Outputs*), corresponding to respectively 2,87% and 10.52%.

[3] https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html
[4] https://github.com/matteoscrugli/ecg-classification-quantized-cnn
[5] https://pytorch.org/docs/stable/_modules/torch/quantization/observer.html

Hyperparameter	***Value***	***Hyperparameter***	***Value***
Epochs	200	Optimizer	SGD
Batch size	32	Learning rate	0.01
Loss criterion	Cross Entropy	Momentum	0.9
ES patience	5	ES evaluation	Every epoch

Table 4.3: Hyperparameters used during the training phase.

4.3.1 Model exploration

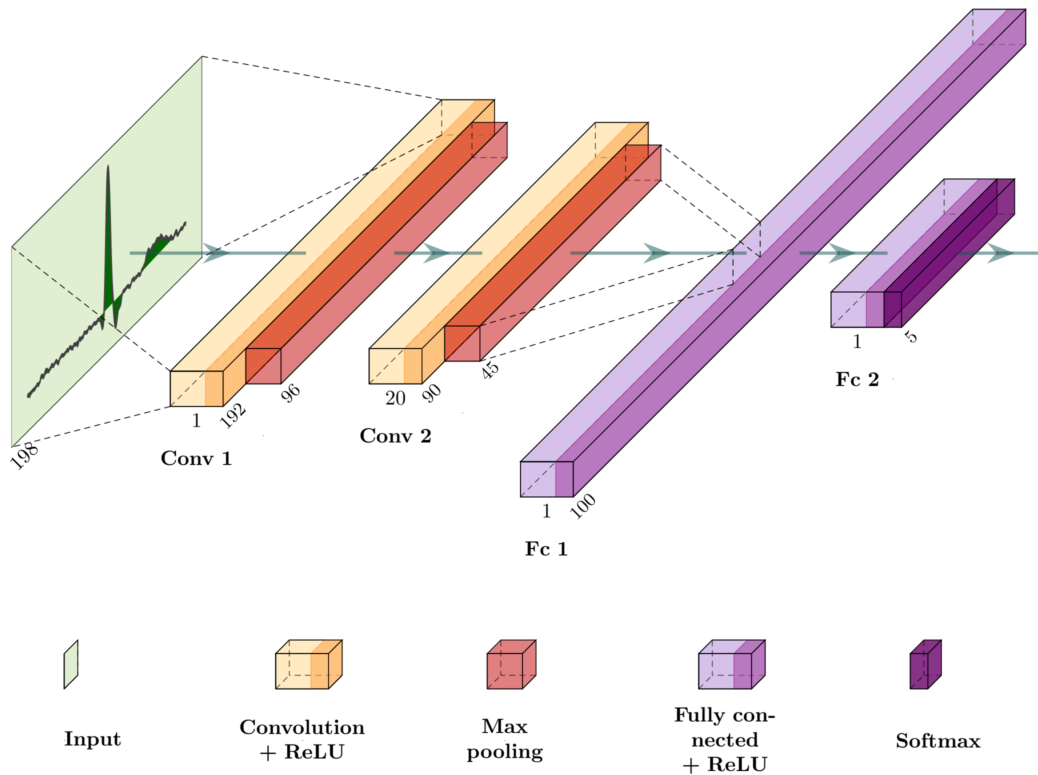
In order to select an optimized CNN topology implementing the classification task required for the system, we have carried out a design space exploration process, comparing tens of neural network topologies in terms of accuracy reached after training and in terms of computing workload associated with executing the inference task on SensorTile. We have explored multiple topologies composed by two convolution layers, two down-sampling layers, and two fully connected layers, as represented in Figure 4.5, the size of the input sample frame is equal to 198.

Explored topologies feature different numbers of output channels from each layer. The results are reported in Figure 4.6, showing the most interesting results for both the *NLRAV* and *NSVFQ* classes. Models *NLRAV_20_20_100* and *NSVFQ_20_20_100* achieve the highest Accuracy (ACC) value as shown in Equation 4.3 and 4.4. The training set is composed by 70% of the elements of the entire dataset and they are chosen randomly. Figure 4.7 shows the trend of the accuracy value during the training stage. As shown in Table 4.3, a maximum number of epochs has been set equal to 200 and, to avoid overfitting effects, the Early Stopping (ES) algorithm was chosen. This algorithm stops the training phase if it detects an increase in the loss value [104], the loss is evaluated every epoch and a *patience* value of 5 is chosen, i.e. the training stops only if a loss increment is detected for 5 consecutive epochs. The loss values for each epoch during the training phase are reported in Figure 4.7.

$$ACC_{NLRAV_20_20_100} = \frac{TP + TN}{TP + TN + FP + FN} = 0.9922, \tag{4.3}$$

$$ACC_{NSVFQ_20_20_100} = \frac{TP + TN}{TP + TN + FP + FN} = 0.9889. \tag{4.4}$$

Figure 4.8 shows a Pareto plot representing accuracy and energy consumption for the most accurate topologies identified by the exploration.



(N) Normal beats	**(N)** Normal beats
(L) Left bundle branch block	**(S)** Superventricular ectopic beats
(R) Right bundle branch block	**(V)** Ventricular premature contraction
(A) Atrial premature contraction	**(F)** Fusion beats
(V) Ventricular premature contraction	**(Q)** Unclassificable beat

Figure 4.5: CNN structure and two possible classes of labels.

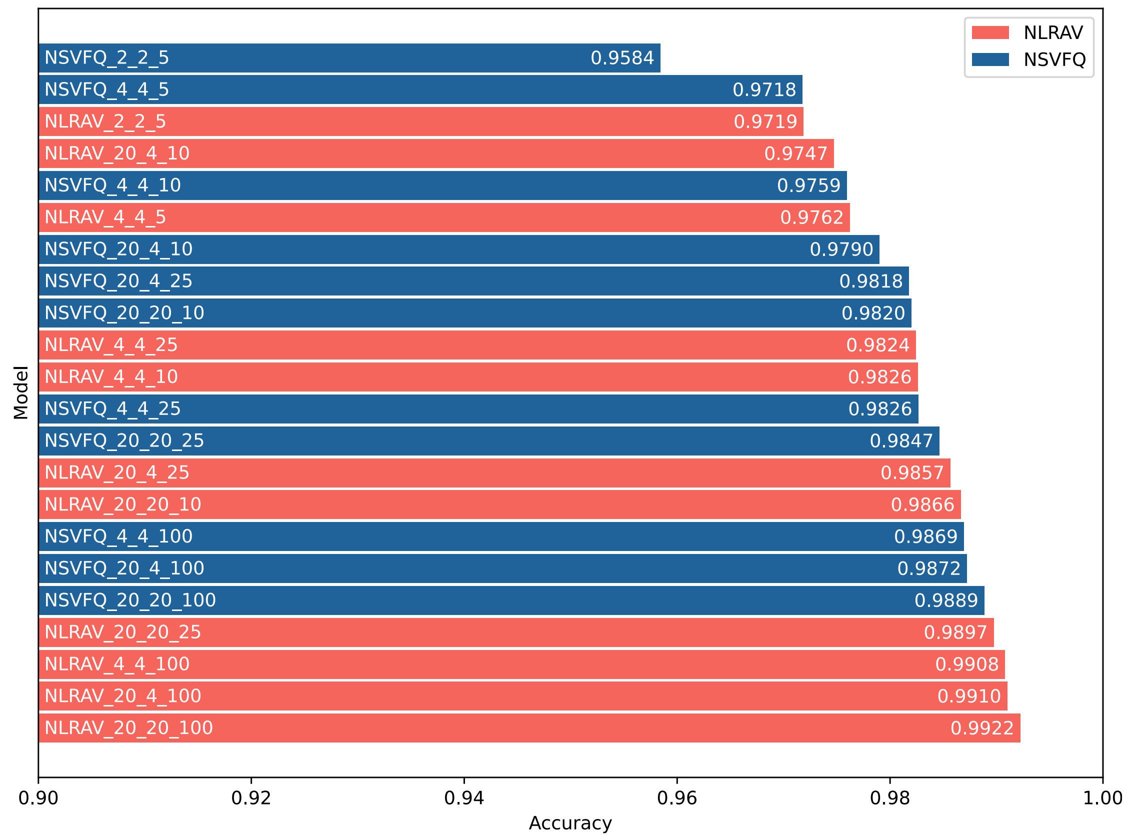
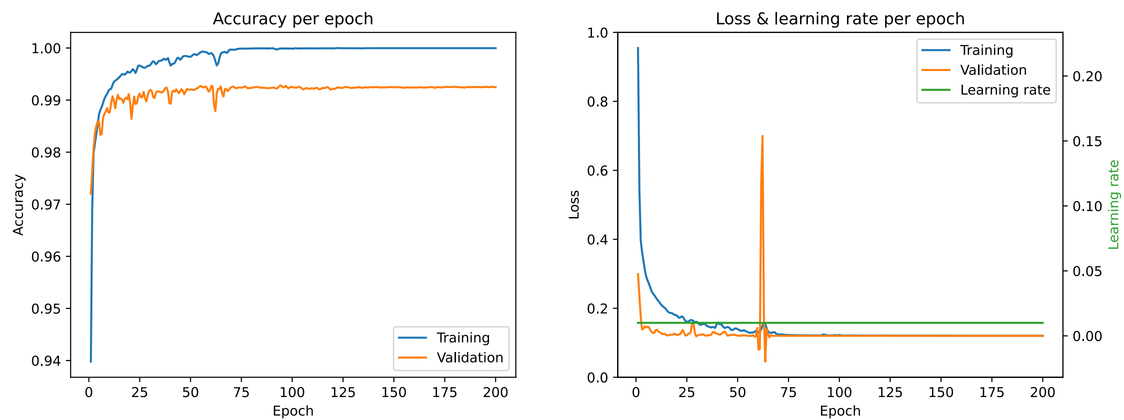
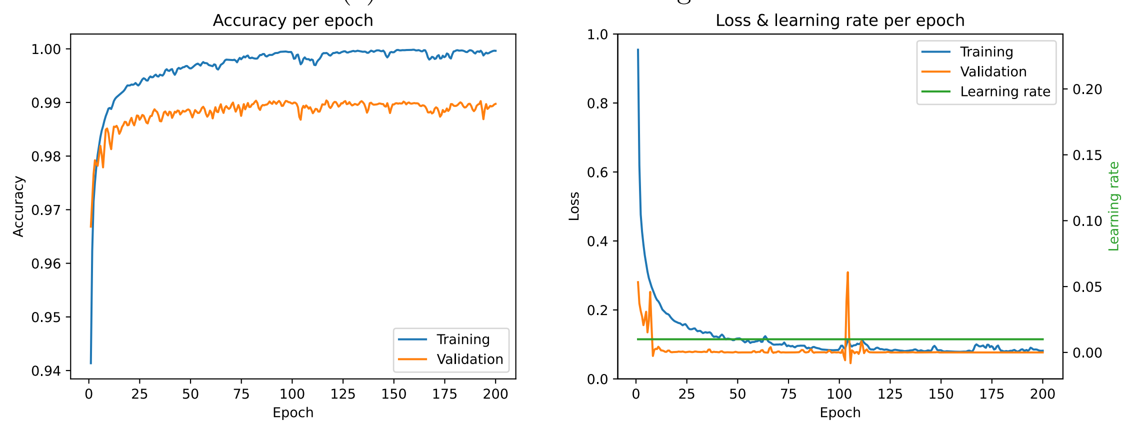


Figure 4.6: Exploration of the chosen neural network model, the name comes from *labels_ conv1OutputFeatures_ conv2OutputFeatures_ fc1Outputs*.



(a) NLRAV classes training results.



(b) NSVFQ classes training results.

Figure 4.7: Results obtained from the training of the model having: 20 output features for *Conv1*, 20 output features for *Conv2* and 100 output for *Fc1*.

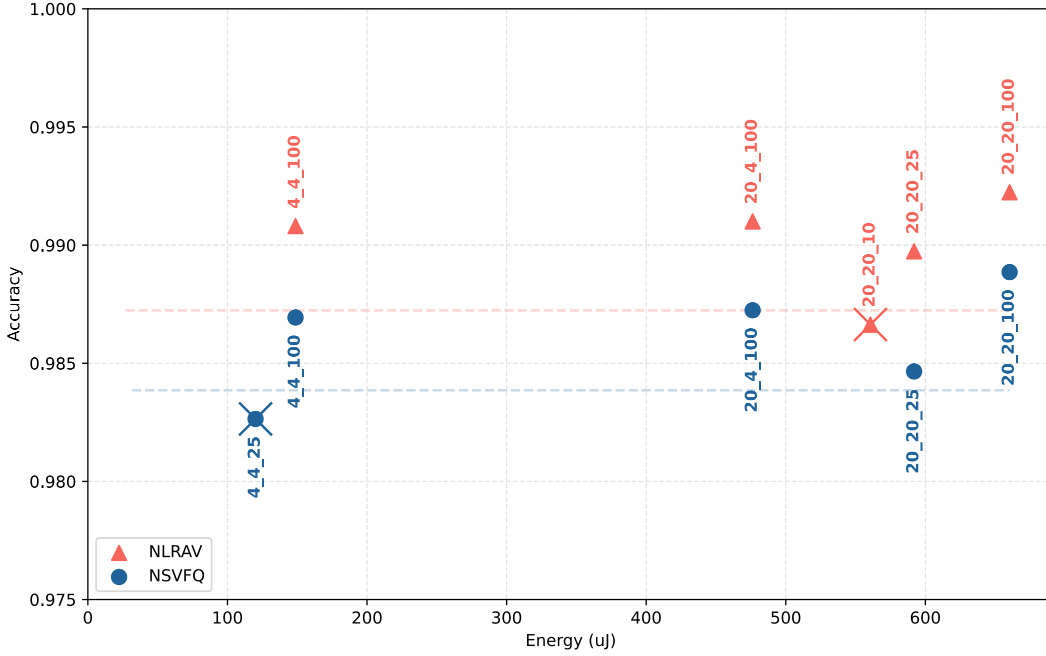


Figure 4.8: For the most accurate models, the energy consumption for a single CNN task call is shown. The dotted line represents the maximum allowable drop in accuracy (0.5% with respect to the most accurate model) for NLRAV (red line) and NSVFQ (blue line) classes. The models marked with an "×" do not respect the constraints imposed on the minimum necessary accuracy value.

For both classes NLRAV and NSVFQ, only one neural network model must be selected which allows to reduce power consumption as much as possible but, at the same time, does not lead to an excessive drop in accuracy. A maximum accuracy drop equal to 0.5% with respect to the most accurate model (represented in Figure 4.8 by the dotted lines) was chosen. The reported energy consumption is associated with a single CNN inference task execution on SensorTile. Models that are above the 0.5% threshold are considered to be valid, and, for each set of labels, the valid model that consumes less energy is chosen to be refined in the next steps and deployed on the board. Eventually, we have selected *NLRAV_4_4_100* and *NSVFQ_4_4_100*. The accuracy values are reported in Equation 4.5 and 4.6.

$$ACC_{NLRAV} = 0.9908\,, \tag{4.5}$$

$$ACC_{NSVFQ} = 0.9869\,. \tag{4.6}$$

4.3.2 Post-deployment degradation and refinement with Augmentation

The ECG peaks in the reference dataset are perfectly centered in the frame of samples that is received in input by the CNN during the training stage. As a consequence, the network is trained to recognize the chosen classes as long as the peak is centered in the signal frame. The peak detection algorithm on the SensorTile, on the other hand, operates online on the incoming signals and would not always detect the peak in the same position specified in the dataset.

To assess the accuracy degradation after the deployment, we calculated post-deployment accuracy values by:

- considering false positive and false negative peaks produced by the peak detection algorithm, which need to be accounted for in Equation 4.5 and 4.6.
- using a post-deployment validation dataset, composed by the same samples in the original one, but modified to be centered as dictated by the peak detection algorithm during online analysis.

In these conditions, there is a degradation in accuracy, the results will be shown and discussed in Section 4.4. Pre-deployment accuracy, therefore, does not consider the aforementioned non-idealities.

To overcome the deriving inaccuracy, the chosen networks have been retrained for refining their precision in case of imperfectly centered input frames. We have used a data augmentation technique to create a larger dataset that contains not only perfectly centered peaks in the frame but also off-center ones. Operations like translating the training signal for a few samples in each direction can often greatly improve generalization [104]. We have chosen to create a dataset that contained peaks translated (with respect to those contained in the starting dataset) to the left or to the right by a number of samples multiple of 3, with a maximum translation of 48 samples. Figure 4.9 shows qualitatively how the peak translation technique was applied as augmentation technique. Once the dataset augmentation has been generated (larger than the one initially used), during the training phase, a number of elements equal to the size of the dataset initially used are taken into consideration, these elements are chosen randomly for each epoch.

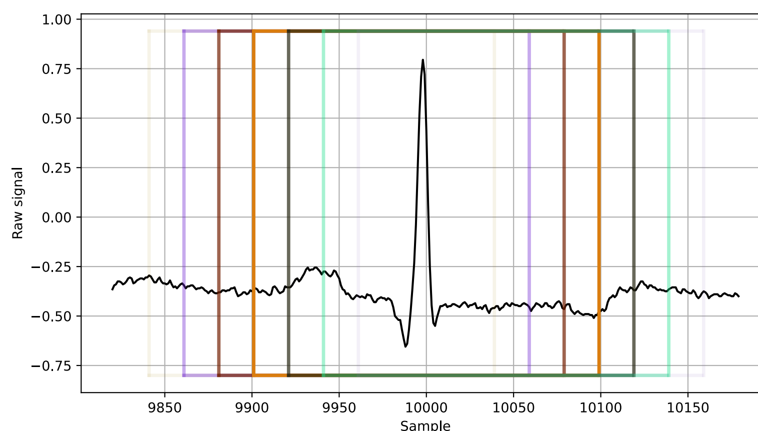


Figure 4.9: Qualitative example of augmentation.

4.4 Experimental results

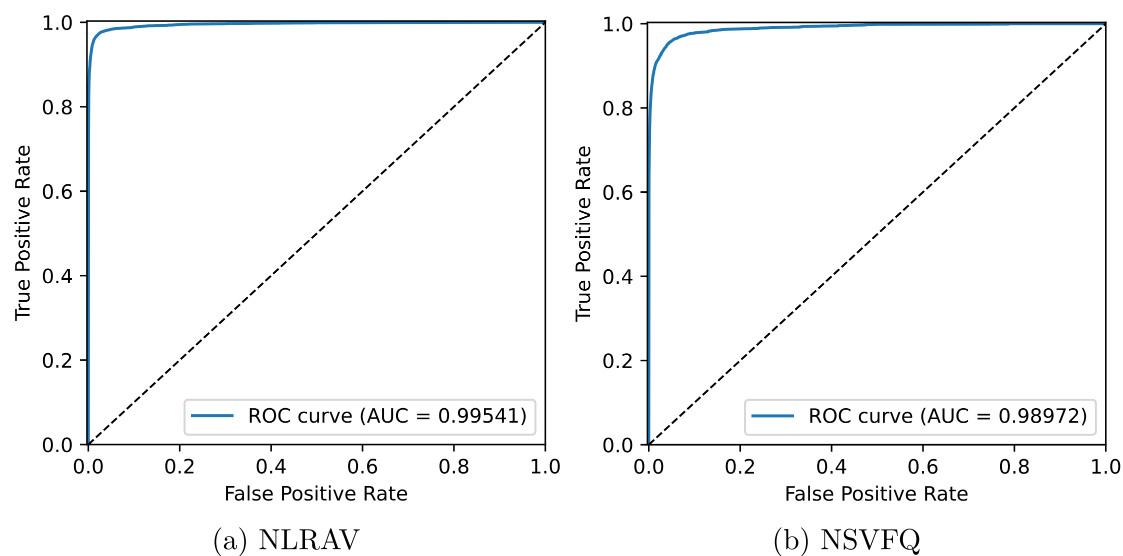
In this section, we show our main experimental results. We first show a detailed accuracy evaluation to show the effectiveness of the data augmentation procedure and the class-level classification capabilities of the designed CNNs. Moreover, we present measures of the energy consumption of the entire system and we highlight the energy contributions of each task. To estimate power consumption in each operating mode, we have performed a thorough set of experiments measuring energy consumption in different setup conditions. The results were used to create a model highlighting the contribution of each task to the energy consumption of the node.

4.4.1 Pre-deployment CNN accuracy

As anticipated in Section 4.3.1, after the topological exploration on neural networks, we selected the NLRAV_4_4_100 and *NSVFQ_4_4_100* models. A pre-deployment accuracy of 99.08% and 98.69% was obtained for classes *NLRAV* and *NSVFQ* respectively. Table 4.4 summarizes the layer parameters used during the training phase. Figure 4.10a and 4.10b, show the Receiver Operating Characteristic (ROC) curve and Area Under Curve (AUC) value for *NLRAV_4_4_100* and *NSVFQ_4_4_100* with augmentation. Eventually, Figure 4.11 shows the confusion matrix for the two selected networks.

Layer	***Input dimension***	***Output dimension***	***Input features***	***Output features***	***Kernel size***
Convolutional	198	192	1	4	7
Max pooling	192	96	4	4	2
Convolutional	96	90	4	4	7
Max pooling	90	45	4	4	2
Fully connected	180	100	–	–	–
Fully connected	100	5	–	–	–

Table 4.4: Parameters used for *NLRAV_4_4_100* and *NSVFQ_4_4_100* training phase.



(a) NLRAV (b) NSVFQ

Figure 4.10: ROC curve and AUC value for *NLRAV_4_4_100* and *NSVFQ_4_4_100* models with augmentation support.

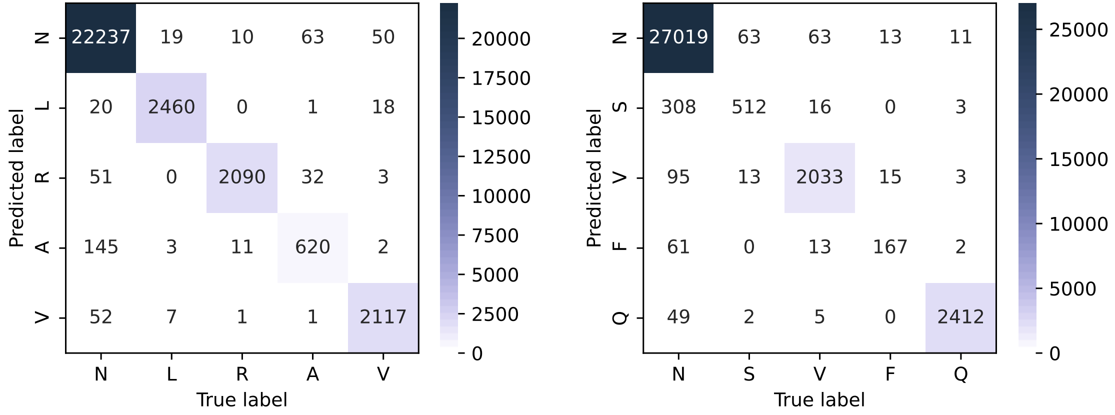


Figure 4.11: Confusion matrix for *NLRAV_4_4_100* and *NSVFQ_4_4_100*.

4.4.2 Post-deployment CNN accuracy

As mentioned above, when considering the ideally centered samples in the dataset, the selected CNNs are very accurate. The precision of the classification, however, decreases significantly when peaks are detected online and imperfectly centered. In fact, for the selected neural network models, a post-deployment accuracy equals to 94.52% and 94.09% for *NLRAV_4_4_100* and *NSVFQ_4_4_100* respectively is obtained. As a solution to such accuracy degradation, we have enriched the training set with samples derived from the original ones by applying some artificial shifting as described in Section 4.3.2. Data augmentation techniques reduce the specialization of the CNN on the perfectly centered validation set, sightly dropping the accuracy to a value of 98.37% and 97.76% for *NLRAV* and *NSVFQ* respectively. On the other hand, ECG recordings with anomalous peaks, that are difficult to be perfectly centered by the peak detection algorithm, are expected to be classified much more accurately. To prove the obtained improvements, we report a detailed classification analysis. In Figure 4.12, we report the number of false positives and false negatives cases resulting from the peak detection algorithm, and we classify the remaining cases, true positives, with the neural network selected for *NLRAV* and *NSVFQ* classes. Such classification is executed on the post-deployment validation set mentioned in Section 4.3.
The improvement in post-deployment accuracy after data augmentation is shown in Equations 4.7 and 4.8.

$$ACC_{NLRAV_post} = 0.9742\,, \tag{4.7}$$
$$ACC_{NSVFQ_post} = 0.9698\,. \tag{4.8}$$

Data augmentation techniques allow recovering most (around 2.9%) of the

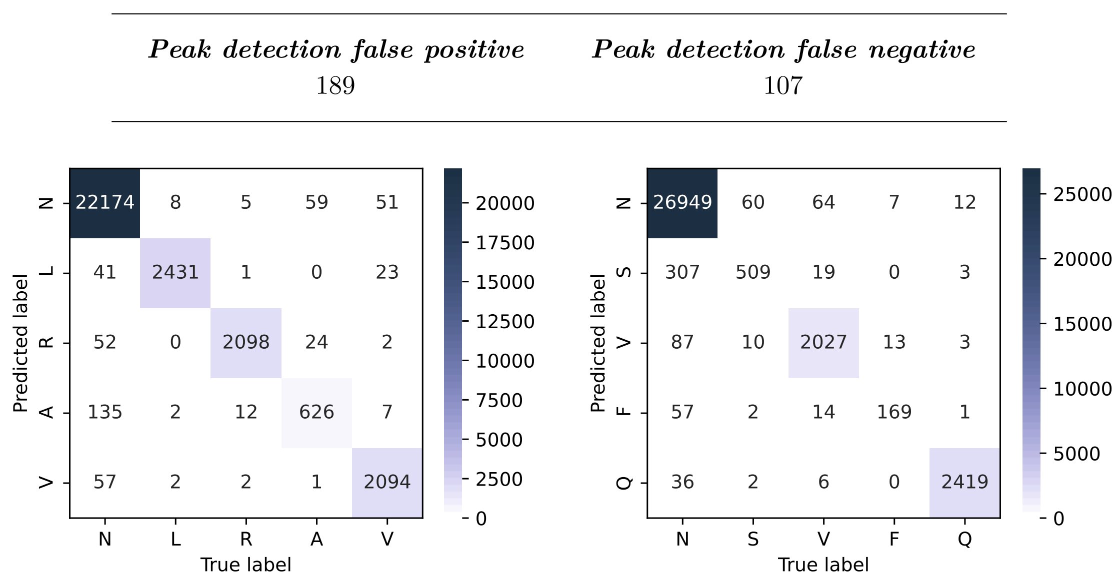


Figure 4.12: False positives and false negatives cases resulting from the peak detection algorithm and classification with remaining true positive cases for NLRAV and NSVFQ classes using CNNs trained with augmentation techniques.

drop due to imperfect centering of the input ECG peaks. Data augmentation has obviously no effect on the drop due to misdetections, which still determine 1.7% degradation with respect to the pre-deployment phase.

Figure 4.13 shows a more detailed view of the effects of the quantization procedure and of the augmentation on the accuracy, focusing on the classification of the peaks detected online. The two leftmost plots represent the accuracy levels when no augmentation is exploited. The accuracy, as can be noticed in the leftmost bar of each plot is very high, with small variability over the different tracks, and is only slightly decreased when quantization is applied to obtain a fixed-point implementation. However, when considering the positioning of the peak as identified by the online detection, as shown in the two rightmost bars of each plot, precision degrades on some of the tracks, as can be noticed by the presence of multiple outlier tracks with very bad classification accuracy. This happens independently on the data representation format since the behavior is similar for both the fixed- and floating-point implementations. The two graphs on the right show the impact of data augmentation. As may be noticed by the rightmost bars in these two plots, general accuracy is significantly improved: classification works correctly for all the tracks and even the outliers show an accuracy higher than 90%.

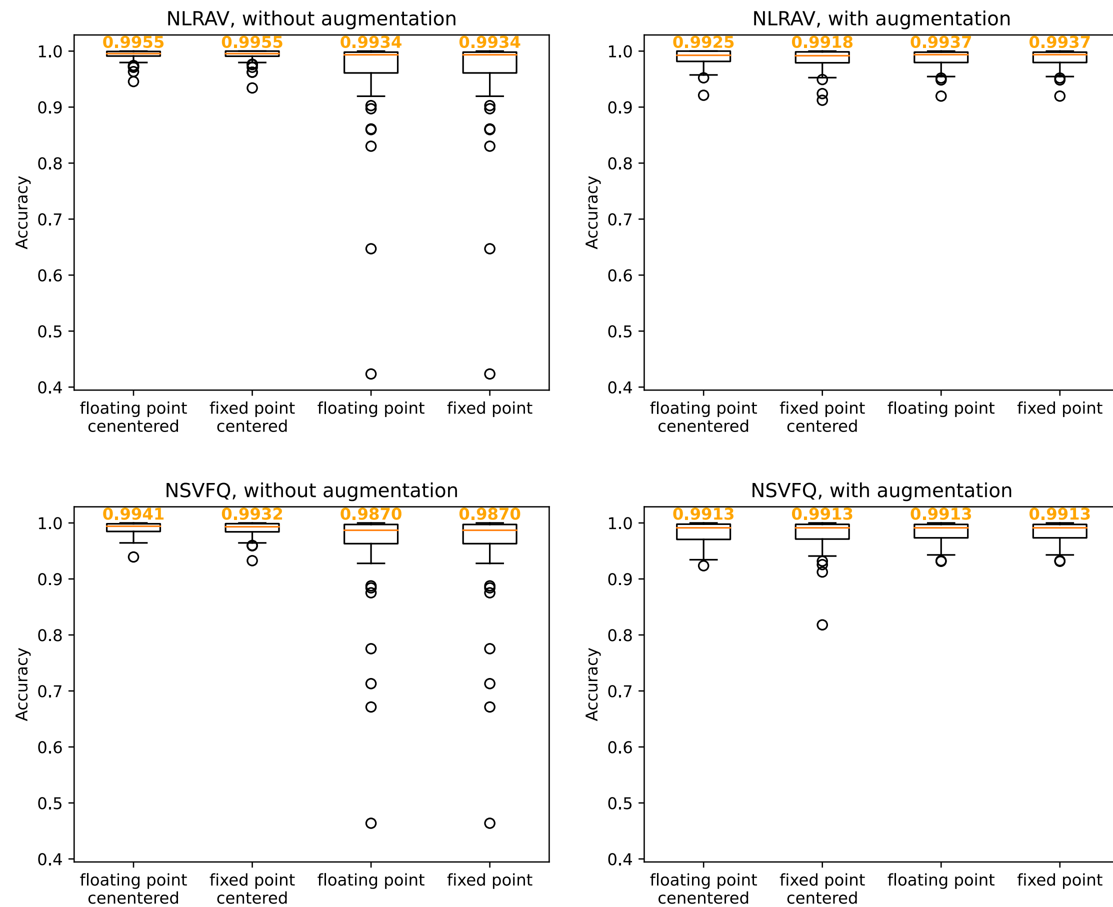


Figure 4.13: Taking into consideration the true positive peaks obtained with a tolerance equal to 50 samples, the statistical distribution of the accuracy values for each ECG recording, obtained from the classification on the validation set, is represented. The floating-point and fixed point models are tested, inference with centered and non-centered peaks is also tested. In orange, the median value.

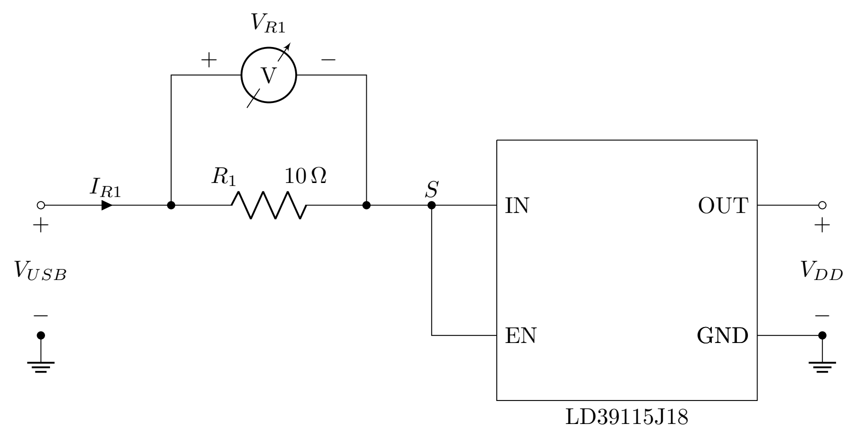


Figure 4.14: Circuit used to measure power consumption.

4.4.3 Power consumption measures

With the purpose of measuring the power consumption of our reference platform, the SensorTile board, we monitored the current absorption through an oscilloscope and a Shunt resistor. We used ANALOG Discovery 2 to measure the voltage on the shunt resistor, Figure 4.14 shows the circuit schematic used to measure the power consumption, in particular: V_{USB} is equal to $5V$, R_1 is the shunt resistor, LD39115J18[6] (SensorTile component) is a voltage regulator, V_{DD} is the power supply voltage of the entire chip. The voltage regulator holds V_{DD} at a stable voltage of $1.8V$. As input, it accepts a voltage between 1.5 and $5.5V$, then voltage drops on R_1 are expected. The Figure 4.15 shows some data on power consumption experimental results for different operating modes at different heart beats per minute (bpm), the individual cases will then be taken and discussed.

4.4.3.1 Case: 50 *bpm*

With low heart rates values, considerable energy savings are obtained even without adapting the system frequency to the workload. In fact, *peak detection OM* and *CNN processing OM* are workload-dependent, which in this case is low. For the latter reason, they consume less than the *raw data OM*, which constantly sends data to the cloud. There is a further energy saving given by the reduction of the system frequency according to the workload, in this case, the *peak detection OM* is set to $2\,MHz$ and the *CNN processing OM* is set to $4\,MHz$. The *raw data OM*, the worst case, works always at $8\,MHz$. The Figure 4.15 also shows the power consumption values if data transmission to the cloud is present or not (Tx, No tx),

[6] https://www.st.com/resource/en/datasheet/ld39115j.pdf

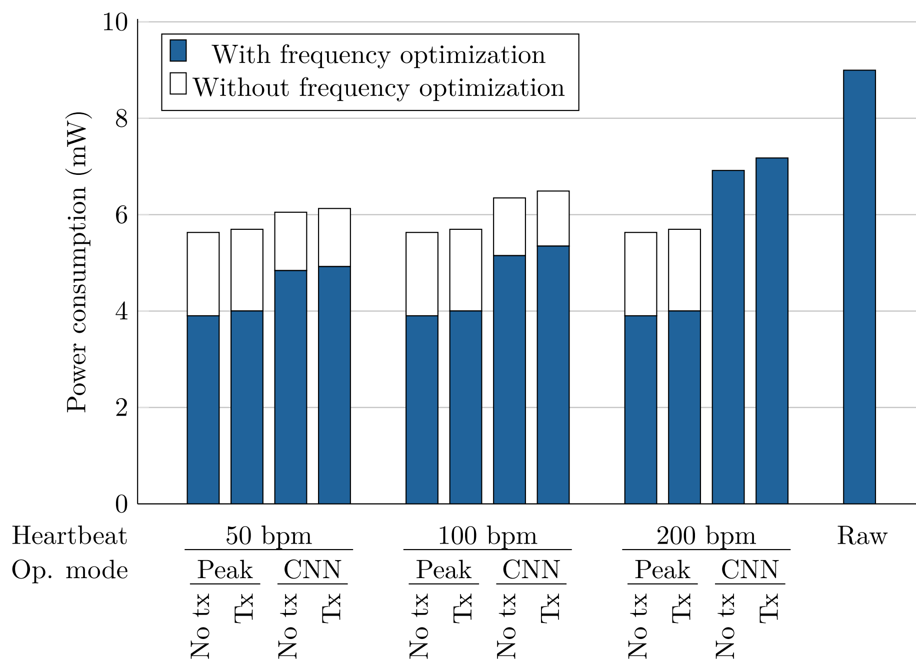


Figure 4.15: The graph summarizes the energy consumption for different heartbeat rates, when data sending is enabled (Tx) or not (No Tx). *Raw OM* does not depend on heartbeat nor on the threshold settings and the threshold task is disabled, so only one value is shown.

as already said, the decision is up to the threshold task.

4.4.3.2 Case: 100 *bpm*

Peak detection OM and *CNN processing OM* keep the same operating frequency of the previous case. Thus there is a slight increase in power consumption for such modes, only due to the more intense data-dependent workload. Obviously, no change in *raw data OM* in terms of power consumption.

4.4.3.3 Case: 200 *bpm*

Compared to the previous cases, again, there are no changes in the *raw data OM*. An increase of the working frequency to 8 *MHz* is required to sustain the *CNN processing OM*. The role of the threshold task, that implies the difference between the *Tx* and the *No Tx* bar for the *CNN processing OM*, is more important. Even with this very high rate, the CNN-based monitoring is still convenient with respect to the *raw data OM*, confirming the usefulness of near-sensor processing.

4.4.4 Power model & operating mode power consumption estimation

We have performed a thorough set of experiments measuring energy consumption in different setup conditions. The results were used to create a model highlighting the contribution of each task to the energy consumption of the node. By interpolating the experimental results on power consumption in the different use cases and knowing the duration of each task, we were able to build a model capable of estimating the energy consumption of the device under each possible use case. Table 4.5 shows the energy values for each task in the process network. Table 4.6 instead shows the power consumption of the platform in idle state and ECG sensor.

At this point it's possible to easily estimate the power consumption relative to each operating mode, the Equations 4.9, 4.10 and 4.11 the power consumption for each operating mode are calculated.

$$P_{raw\,data\,OM} = (E_g + \alpha E_s) \cdot f_s + P_{idle} + P_{sensor}\,, \tag{4.9}$$

$$P_{peak\,detection\,OM} = E_{gp} \cdot f_s + (E_t + E_s) \cdot f_p + P_{idle} + P_{sensor}\,, \tag{4.10}$$

$$P_{cnn\,processing\,OM} = E_{gp} \cdot f_s + (E_c + E_t + E_s) \cdot f_{hr} + P_{idle} + P_{sensor}\,. \tag{4.11}$$

In Equations 4.9, 4.10 and 4.11, the following operators are used:

Task type	***Number of cycle***	***Execution time (8 MHz)***	***Energy contribution***
Get data	841	$105\,\mu s$	$E_g = 2.96\,\mu J$
Get data + peak	$1\,550 + 841$	$300\,\mu s$	$E_{gp} = 3.76\,\mu J$
CNN 4_4_100	361 360	$45\,ms$	$E_c = 148.78\,\mu J$
CNN 20_20_100	1 719 582	$215\,ms$	$E_c = 660.37\,\mu J$
Threshold	910	$114\,\mu s$	$E_t = 2.73\,\mu J$
Send data	$\sim 25\,000$	$\sim 3\,ms$	$E_s = 83.96\,\mu J$

Table 4.5: Summary of consumption and execution time for each task.

Device	***Power consumption***		
	2 *MHz*	**4 *MHz***	**8 *MHz***
Platform in idle state	$2.609\,mW$	$3.101\,mW$	$4.546\,mW$
ECG sensor	$237\,uW$	$237\,uW$	$237\,uW$

Table 4.6: Summary of consumption of peripherals.

- f_s is the sampling frequency,
- f_{hr} is the heart rate,
- f_p is the peak data sanding frequency,
- α^{-1} it's the number of samples inserted in a BLE package,
- P_{idle} power consumption of the platform in idle state, depends on the system frequency,
- P_{sensor} energy consumption of the ECG sensor.

Figure 4.16 shows the estimate of the power consumption of the device and the contributions of each task in case the heart rate is around 60 *bpm*. The purpose of Figure 4.16 is to graphically show the power consumption contributions of the tasks for each operating mode.

The following list shows the estimated battery life ($600\,mAh$, $3.7\,V$ Li-Ion) for each operating mode:

- *Raw data OM*: 10.29 days

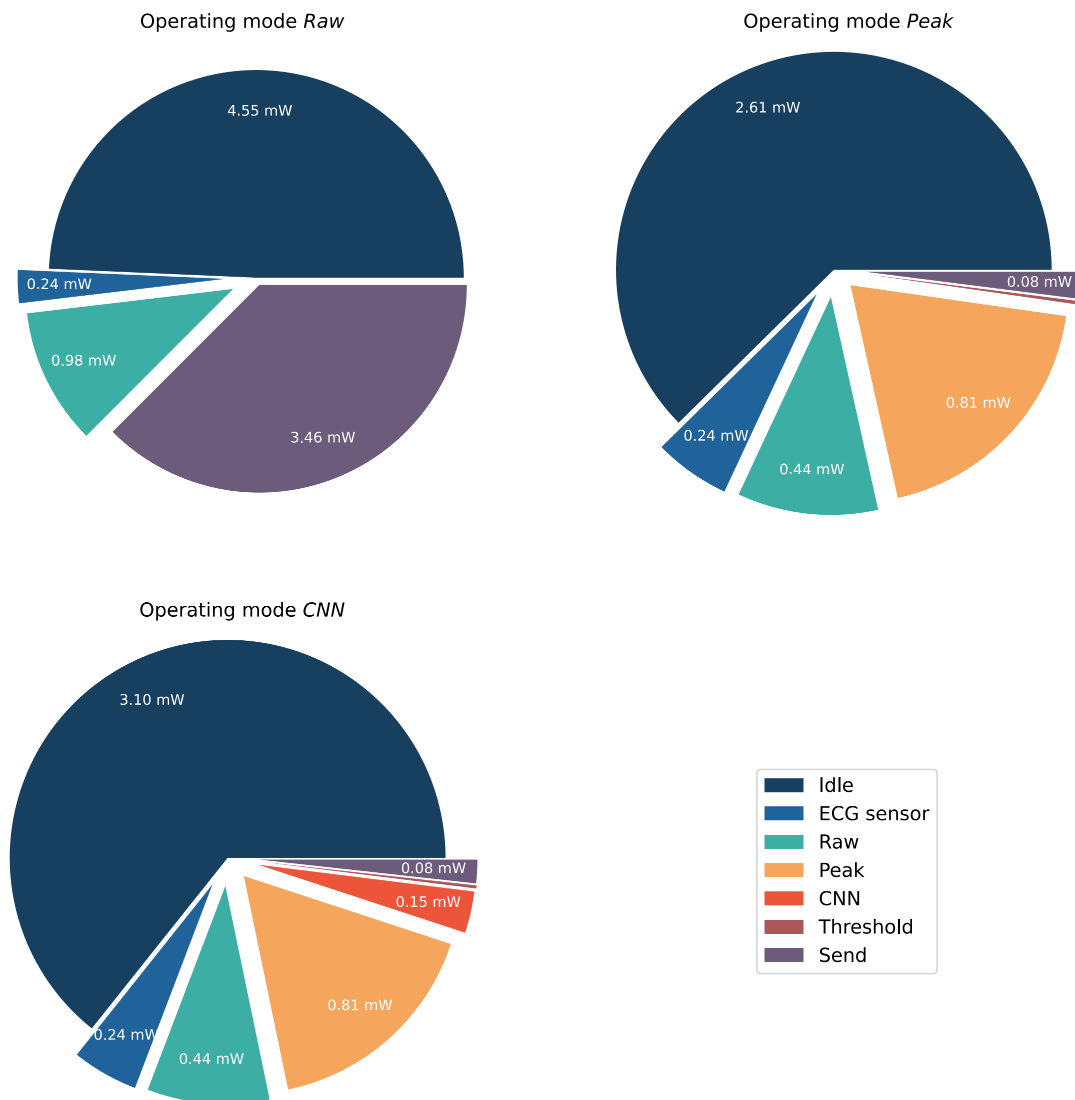


Figure 4.16: Estimation of energy consumption for each task of each operating modes at 60 bpm.

- *Peak detection OM*: 23.49 days
- *CNN processing OM*: 20.20 days

Considering the results in terms of battery life just reported, it is possible to assert that the different optimizations have made possible higher energy efficiency for enabling on-edge processing OMs compared to raw OM. To summarize, the increase in efficiency is mainly due to the introduction of the ADAM component and various optimizations in the design and development phase. ADAM, to this aim, sets the microcontroller in sleep/active mode and selects, at runtime, optimal clock frequency, and power supply to respect the real-time constraints. These hardware-related settings are also optimized at runtime to exploit the data-dependent nature of the workload (based on the heart rate). Finally, we have considered power optimization as the main objective when taking all the design and development choices. We have used a simplified peak detection algorithm. Moreover, we have explored a large number of neural network models to obtain good results in terms of accuracy values with smaller computational resources, with respect to those used in literature. Thanks to quantization and CMSIS APIs, the SIMD microcontroller's capabilities have been exploited as much as possible, significantly increasing the performance of neural network inference and reducing the memory footprint.

4.5 Work comparison

In this section we compare to the works discussed in Section **??** that deals with inference at-the-edge. Table 4.7 summarizes the results in terms of neural network accuracy on MIT-BIH dataset. As may be noticed, our system gets results higher or very close compared to the alternatives, despite being, to the best of our knowledge, the only work actually evaluating post-deployment accuracy, and considering all the contributions to errors deriving by all the steps in the online processing system. Only for the precision metric, there are works that report higher values, but exploiting platforms with much higher computational capabilities than those of a microcontroller and more complex neural networks.

In Sakib et al. [40] a good results in terms of accuracy and precision is obtained, the F-score value is not reported and was therefore calculated from the reported confusion matrix. Power consumption is not provided but a higher value is expected compared to our as their methodology is tested on platforms such as Jetson Nano and RaspberryPi.

In Azimi et al. [46] the inference does not occur in the Cloud side but from an intermediate device placed in the same WBAN network as the sensory node, the latter will have the task of transmitting all the data acquired in raw format thus leading to a possible excessive power consumption of the node. Also in this case the F-score parameter was calculated from the results proposed within the paper.

In Burger et al. [47], a good result in terms of accuracy has been obtained, however, they report far higher power consumption than our methodology. The higher consumption is due to the fact that they used an FPGA-based platform and their system is capable of classifying 335 beats per second.

In Hou et al. [52] they obtain good results in terms of accuracy, although in the work there are not many references to how they were calculated and there are no supporting confusing matrix, making the calculation of the remaining parameters not possible. Here too, a Raspberry is used as a reference platform, which leads to a significantly increasing of power consumption if compared to those obtained in our work.

Work	***Accuracy***	***Sensitivity***	***Precision***	***F-score***	***Diseases***
Sakib et al. [40]	95.98%	–	95.9%	93.5%	NSVF
Azimi et al. [46]	96%	76.18%	44,51%	61,6%	NSVFQ
Burger et al. [47]	97%	–	–	–	NLRAV
Hou et al. [52]	98%	–	–	–	[1]
Our work	99.08%	98%	96,31%	98,12%	NLRAV
Our work	98.69%	95,52%	92,6%	96,16%	NSVFQ
Post-deployment results					
Our work	97.42%	96,92%	91,50%	94,89%	NLRAV
Our work	96.98%	95,35%	85,17%	91,12%	NSVFQ

[1] Normal (NOR), Left Bundle Brunch Block (LBB), Right Bundle Brunch Block (RBB), Paced beat (PAB), Premature Ventricular Contraction(PVC), Atrial Premature Contraction (APC), Ventricular Flutter Wave (VFW) and Ventricular Escape Beat (VEB).

Table 4.7: Results in terms of accuracy value on MIT-BIH dataset (see classes names in Figure 5.4).

5 Sensorimotor exercise detection using a wobble board

We used a wobble board capable of 360° rotation (Figure 5.1), the sensory node is fixed in the upper-middle part. Since the device is battery powered and communication is via a BLE module, no cable is used to interface with the sensor node. We chose STMicroelectronics SensorTile microcontroller-based platform, which is equipped with an ARM Cortex-M4 32-bit low-power microcontroller. It takes advantage of the LSM303AGR[1] accelerometer sensor integrated into the Sensortile, only the two axes X and Y parallel to the floor are taken into account. It was chosen to run FreeRTOS on the node, to have more control over the running tasks due to its ability to create a thread-level abstraction.

This chapter describes the implementation of a system able to recognize typical movements in exercises that involve the use of a conventional wobble board or, more simply, the wireless transmission of raw data acquired from the sensor.

[1] https://www.st.com/resource/en/datasheet/lsm303agr.pdf

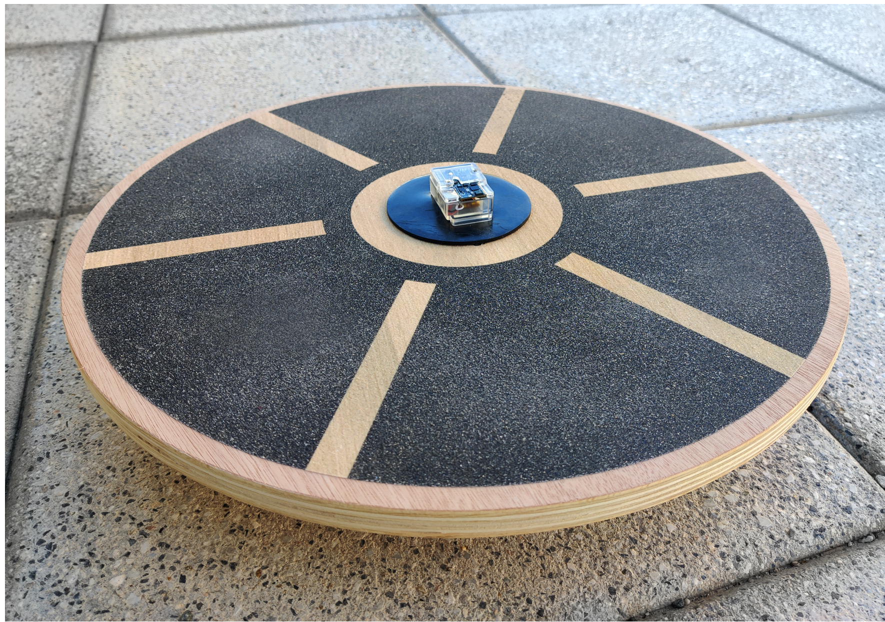


Figure 5.1: Wobble board used to validate our approach.

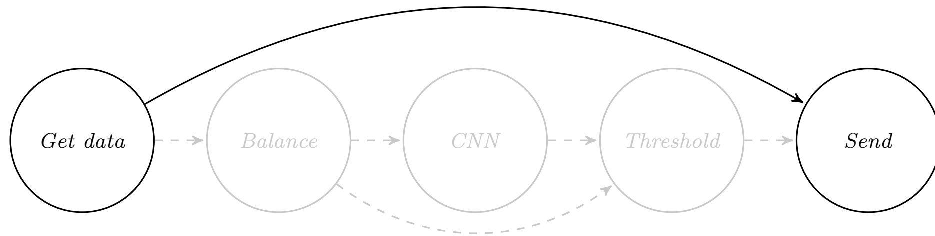
5.1 Operating modes

The application model chosen for this use case provides two possible levels of processing able to evaluate the nature of the movement. The OMs chosen for the selected use case are shown in Figure 5.2 and described below.

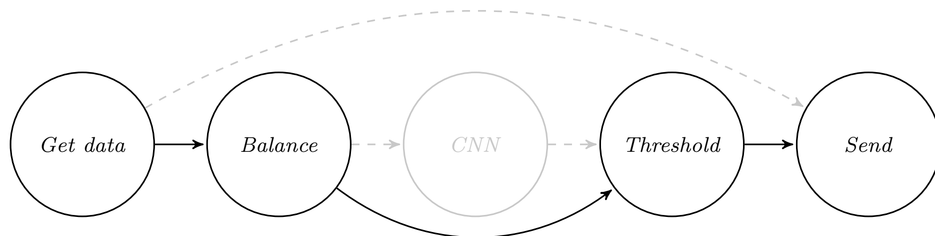
5.1.1 Operating mode: raw data

This is the simplest OM, using only two tasks. It is possible to acquire data from the sensor and send it via Bluetooth, with a sampling frequency of 100 Hz. In order to reduce the power consumption related to the transmission, it was decided to encapsulate four samples taken from the sensor in a single low energy Bluetooth packet. The Bluetooth packet has a size of 20 bytes, 4 bytes the timestamp, and four pairs of data taken from the sensor at different instants of time, the data pair is formed by the values relative to the accelerometer's X and Y axis, each with a size of 2 bytes.

Operating mode: Raw data.



Operating mode: basic balance.



Operating mode: CNN.

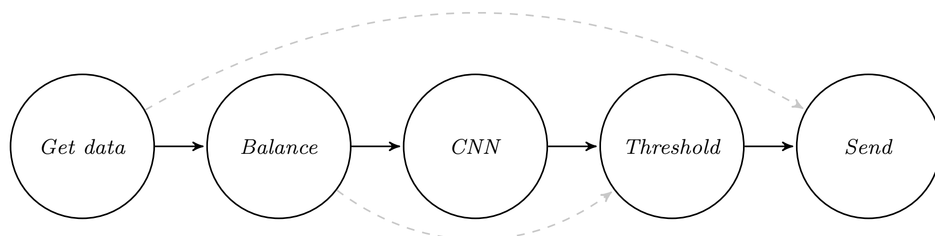


Figure 5.2: Application model. Top *raw OM*, middle *balance OM*, bottom *CNN OM*.

5.1.2 Operating mode: basic balance

This OM enables the first level of processing. The sampling rate is lowered to 100/7 Hz, which is more than sufficient to perform the analysis in this OM. A simple algorithm calculates how much, in percentage, the wobble board is in a balanced position. The extreme cases, the analysis returns a value of 100% if the board remains horizontal within a certain tolerance and 0% when the board remains in constant contact with the ground. The result of the analysis is transmitted every second, this leads to a significant energy saving due to the decrease of information that has to be sent via Bluetooth, which is no longer used to transmit raw data.

5.1.3 Operating mode: CNN

Some exercises were selected which were not too complicated to be recognized by deep learning techniques. Also in this case, a frequency of 100/7 Hz is ideal to obtain good results with the neural network and have a not excessive workload. The raw data related to the two X and Y axes of the accelerometer are used as two different input features as input to the neural network. The *balance* task remains active so that if a total stop of the table is detected, no CNN is executed. Again, the result of the analysis is transmitted every second. Some typical exercises recommended by Anders Heckmann [3] are those shown in Figure 5.3.

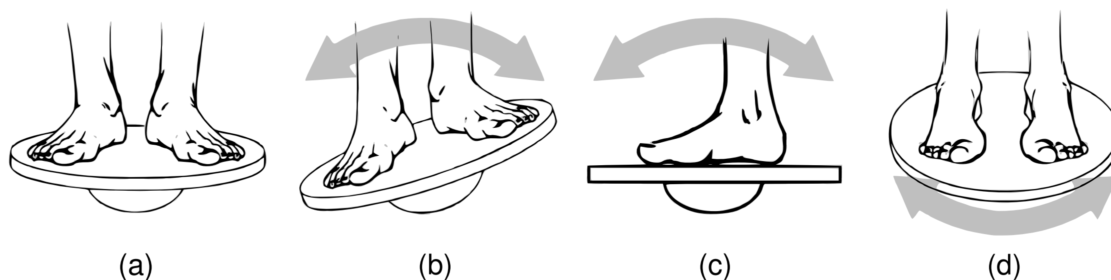


Figure 5.3: The four common wobble board exercises recommended by physiotherapist Anders Heckmann [3]. (a) Balance while keeping as steady as possible. (b) Move the board back and forth. (c) Move the board from sideto side. (d) Clockwise and counterclockwise circular movement. The Figure was extracted from Nilsson et al. [4].

The correct execution of the exercise involves:

- **Basic stance balance (Figure 5.3.a):** Stand on the board with the edges of your feet on the outer edges of the board. Maintain a neutral spine and keep your torso upright. Balance on the board by shifting your weight to

prevent any of the board's edges from touching the floor. The goal is to maintain the balance for 60 seconds.

- **Forward/backward tilt (Figure 5.3.b):** Stand on the wobble board with your feet on the outer edges. Stand upright in a neutral spine position. Tilt the board to the front to touch the floor. Tilt it back onto the heels to touch the floor behind you. Continue tilting forward and back in a slow, steady, controlled motion for 60 seconds.

- **Side tilt (Figure 5.3.c):** Stand on the wobble board with your feet on the outer edges. Stand upright in a neutral spine position. Tilt the board from left to right by transferring your weight from your left leg to your right leg. Moving in a slow and controlled manner, keeping an upright torso and tight core. The duration of the exercise is 60 seconds.

- **Two leg tilts (Figure 5.3.d):** Stand on the wobble board with your feet on the outer edges. Stand upright in a neutral spine position. In a combination of the two previous exercises, you will roll the board in a 360-degree motion. Begin by tilting the board to the left. When the board touches the ground on the left, transfer your weight to the front to touch the floor. Now transfer your weight to touch the floor to the right side. Complete the revolution by tilting the board to the floor behind you. Keep your body centralized throughout. You may need to balance with your arms as you get used to the movement. Reverse the motion to move in the other direction. Continue for 60 seconds.

- **Other:** there is a fifth class that represents everything that is not foreseen by the previous exercises, for example, the fall from the table or the absolute absence of movement.

5.2 Neural network design

We used a training procedure that included a static quantization[2] step, the source code is available in our public repository[3]. This process converts floating-point weights and activations to integers, allowing the CNN to be implemented using the CMSIS-NN optimized function library, which expects inputs with 8-bit precision. We have chosen to force the value of the bias to zero, while for the conversion

[2] https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html
[3] https://github.com/matteoscrugli/deepwobbleboard

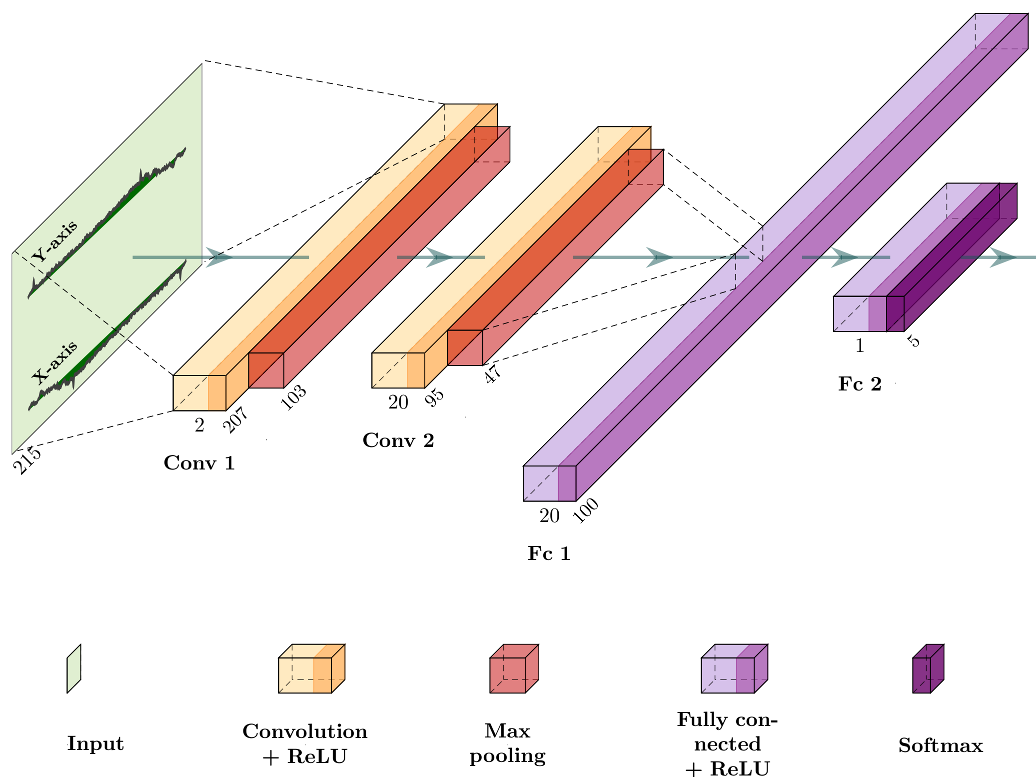
Hyperparameter	***Value***	***Hyperparameter***	***Value***
Epochs	30	Optimizer	Adadelta
Batch size	32	Learning rate	1.0
Loss criterion	Cross Entropy	Rho	0.9
ES patience	5	ES evaluation	Every epoch

Table 5.1: Hyperparameters used during the training phase.

of the weights we have inserted *MinMax* observers[4], who have the task of studying the outputs of each layer. Evaluating the distribution of the output values of each layer allows the observer to establish a value of *scale* and *zero-point* in order not to saturate these values using a quantized network. The CMSIS-NN library's functions for implementing convolution and fully connected layers include output shifting operations for applying the Scale factor to the outputs, with scaling values ranging from -128 to 127. In PyTorch, however, the quantization procedure requires a Scale value that is not always a power of two. As a result, we modified the CMSIS functions slightly to support arbitrary *scale* values. This change resulted in a minor increase in inference execution time. After testing inference with and without this modification, we calculated an increase in execution time of 2.87%.

We used a design space exploration process to compare tens of neural network topologies in terms of accuracy achieved after training and computing workload associated with executing the inference task on SensorTile. Figure 5.4 shows the selected convolutional network and the five selected output classes.
All exercises are one minute in length, each movement performed during the four different exercises has a different duration. Generally, the longest exercise is the two leg tilts. CNN does not analyze the exercise for its entire duration (60 seconds) at once, the signal is divided into windows of 15 seconds duration, a good compromise between temporal precision and distinction between the movements to be evaluated. The maximum number of epochs was set to 30 and the ES algorithm was chosen to avoid overfitting effects. This algorithm terminates the training phase if it detects an increase in the loss value [104]; the loss is evaluated every epoch, and a Patience value of 5 is selected, implying that the training terminates only if a loss increment is detected for 5 consecutive epochs. Table 5.1 summarizes the parameters chosen for training, while Table 5.2, shows the dimensions of the various layers chosen.

[4] https://pytorch.org/docs/stable/_modules/torch/quantization/observer.html



(B) Basic stance balance
(S) Side tilt
(G) Other
(FB) Forward/backward tilt
(R) Two leg tilts

Figure 5.4: CNN structure and classes description.

Layer	***Input dimension***	***Output dimension***	***Input features***	***Output features***	***Kernel size***
Convolutional	215	207	2	20	9
Max pooling	207	103	20	20	2
Convolutional	103	95	20	20	9
Max pooling	95	47	20	20	2
Fully connected	940	100	–	–	–
Fully connected	100	5	–	–	–

Table 5.2: Model parameters.

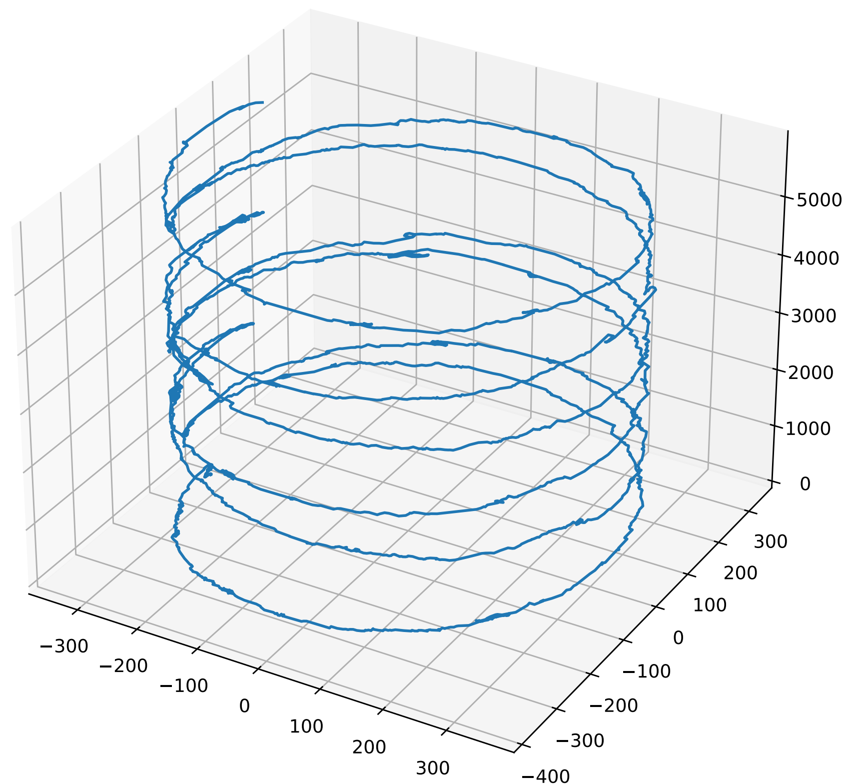


Figure 5.5: Example graph of single acquisition of the "two leg tilts" exercise. The two axes in the lower plane represent the raw data acquired by the sensor. The vertical axis represents time in milliseconds.

5.2.1 Data augmentation & generalization

Unfortunately, no datasets already used in the literature containing the selected exercises were found. Therefore, a dataset was created with 12 one-minute recordings for each type of exercise (including class "other"), Figure 5.5 shows an example graph of single acquisition of the "two leg tilts" exercise. A random split of the dataset was chosen in order to use 0.8% of the data for the training set and 0.2% for the validation set. Operations such as translation, rotation and time dilation of the signal in each direction can often greatly improve generalization [104].

In more detail, augmentation techniques are (also summarized in Table 5.3):

- **Translation:** During training, the window to the entire signal is shifted by 0.25 s per frame. For example, a 60-second recording with a translation of 0.25 generates a number of frames of $(60 - 15)/0.25 + 1 = 181$.
- **Rotation:** A rotation transformation was applied to the X and Y axes of

Parameter	***Value***
Traslation, temporal distance between frames	0.25 s
Rotation, X and Y axis rotation	$\angle -4$, $\angle 0$, $\angle 4$
Time dilation, downsampling	6, 7, 8

Table 5.3: Augmentation parameters.

the sensor data, in our case we chose two rotations of $\angle -4$ and $\angle 4$ degrees. For each record, two more are then generated.

- **Dilation:** The sampling frequency of the signals in the dataset is 100 Hz, but the neural network is trained with 100/7 Hz signals. The size of the input signal is therefore equal to $\lfloor (15 \times 100 + 7 - 1)/7 \rfloor = 215$. Time dilation can be obtained by increasing or decreasing the downsampling while keeping the input size to the neural network constant, in this case, two additional downsampling values of 6 and 8 were chosen. For each recording, two more are generated with different time dilations.

5.3 Experimental results

In this section, we will show the results obtained after the neural network training and we will make a detailed analysis of the power consumption for each OMs.

5.3.1 Neural network accuracy

After the training phase, an accuracy of 97.652% was measured on the validation set. Figure 5.6 shows the results of the training, showing how the windows extracted from the validation set are classified. It is possible to notice that the major difficulty for the network is to recognize in a correct way when the wobble board is used with movements that do not match the four proposed ones.

5.3.2 Power consumption measures

We measured the power consumption for each OMs, for this purpose the digital oscilloscope ANALOG Discovery 2 was used to measure the voltage on the shunt

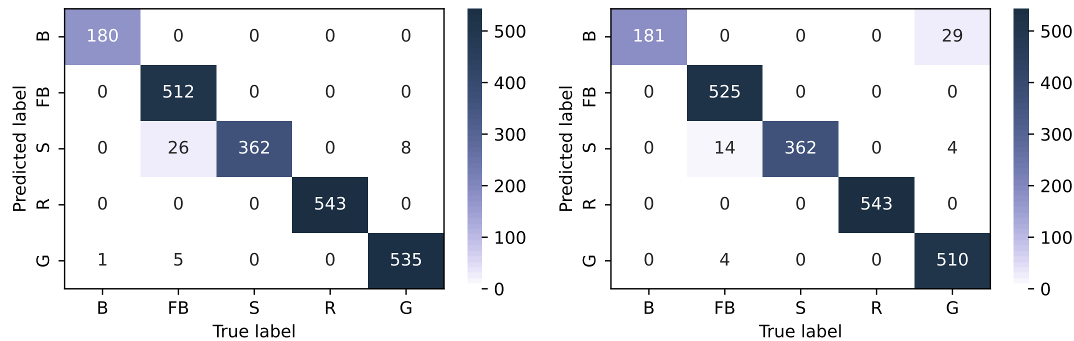


Figure 5.6: Validation set confusion matrix, to the left the model with floating point weights and to the right fixed point weights.

resistor placed in series to the power cable of the SensorTile node. Figure 5.7 shows the result of the measurement.

5.3.2.1 Operating mode *raw*

This is the OM with the highest amount of data to be sent via Bluetooth, the minimum system frequency to handle data traffic with the Bluetooth module present in the SensorTile module is 8 MHz. In order to optimize data sending via Bluetooth, four sensor acquisitions are merged for each packet, reducing the data sending frequency from 100 Hz to 25 Hz.

5.3.2.2 Operating mode *balance*

In contrast to the previous one, this is the OM where there is less data transmission, in fact, the evaluation of the exercise is done every one second, invoking Bluetooth transmission at the same frequency. It has been tested that a system frequency of 2 MHz is sufficient to meet the real-time constraints, Figure 5.7 shows the savings due to dynamic optimization of the system frequency.

5.3.2.3 Operating mode *CNN*

It was chosen to send the information about the classification result of the exercise every time the neural network inference is performed. In order to correctly execute the neural network and at the same time respect the real-time constraints, a system frequency of 4 MHz has been set. The length of the input frame is obviously the same as that used during training, while the distance between frames in this evaluation phase, as for operating mode *Balance*, is one second. For this reason, the

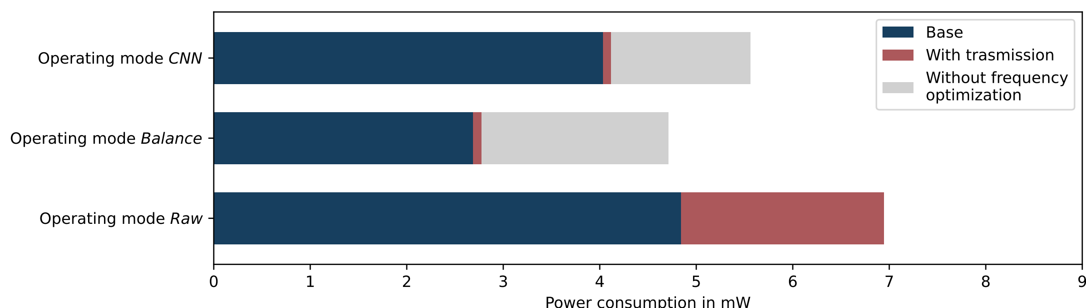


Figure 5.7: Power consumption for each OM.

power consumption is data-dependent and the worst case will thus be taken into account for the calculation of power consumption. The maximum number of times a single data item is sent via the Bluetooth module is equal to $(60-15)/1+1 = 46$.

5.3.3 Power model & operating mode power consumption estimation

We conducted a comprehensive set of experiments measuring energy consumption in various setup conditions. The results were used to create a model that highlighted the contribution of each task to the node's energy consumption. The energy values for each task in the process network are shown in Table 5.4, Table 5.5 instead shows the power consumption of the platform as a function of the chosen system frequency

Task type	***Number of cycles***	***Execution time (8 MHz)***	***Energy contribution***
Get data	841	$105\,\mu s$	$E_g = 2.96\,\mu J$
Get data + balance	$1\,550 + 841$	$300\,\mu s$	$E_{gb} = 3.76\,\mu J$
CNN	2 219 582	$277\,ms$	$E_c = 852.38\,\mu J$
Threshold	910	$114\,\mu s$	$E_t = 2.73\,\mu J$
Send data	$\sim 25\,000$	$\sim 3\,ms$	$E_s = 83.96\,\mu J$

Table 5.4: Summary of consumption and execution time for each task.

At this point, Equations 5.1, 5.2 and 5.3 representing the estimated power

Device	*Power consumption*		
	2 *MHz*	**4 *MHz***	**8 *MHz***
Platform in idle state	$2.609\,mW$	$3.101\,mW$	$4.546\,mW$

Table 5.5: Summary of consumption of peripherals.

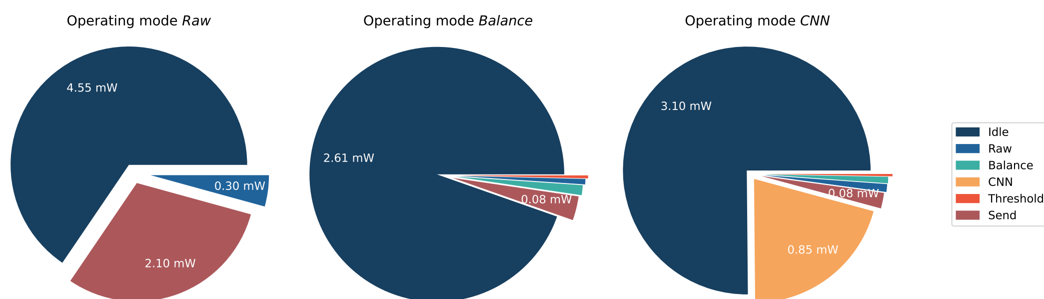


Figure 5.8: Estimation of energy consumption for each task of each OM.

consumption for each OMs have been obtained:

$$P_{raw\,data\,OM} = (E_g + \alpha E_s) \cdot f_s + P_{idle}\,, \tag{5.1}$$

$$P_{basic\,balance\,OM} = E_{gb} \cdot f_s + (E_t + E_s) \cdot f_b + P_{idle}\,, \tag{5.2}$$

$$P_{cnn\,processing\,OM} = E_{gb} \cdot f_s + (E_c + E_t + E_s) \cdot f_c + P_{idle}\,. \tag{5.3}$$

In Equations 5.1, 5.2 and 5.3, the following operators are used:

- f_s is the sampling frequency,
- f_c frequency of convolutional neural network activation,
- f_b is the basic balance data sanding frequency,
- α^{-1} is the number of samples inserted in a BLE package,
- P_{idle} power consumption of the platform in idle state, depends on the system frequency.

Figure 5.8 shows graphically the contribution of each task to the power consumption of each OMs.

6

Runtime reconfiguration on multi-core platforms

In this chapter we extend ADAptive runtime Manager to target multi-core advanced IoT platforms capable of executing more complex applications, thus enhancing near-sensor processing possibilities. This exposes additional challenges when it comes to the dynamic runtime management of the platform. First aspect, modern multi-processor IoT nodes, especially the plethora of prototype solutions currently designed by the community to support AI-related workloads and optimized for low-power, have limited OS support. To try our approach on Orlando, we had to implement adaptivity on a bare-metal system, exploiting the platform-specific set of APIs to manage the application model, the process network, and the related operating modes. As a second aspect, the availability of more cores requires, when switching operating mode, the adaptation of the parallelism level exploitable within the application structure. The workload imposed by a given mode must be optimally partitioned between the available processing elements, using splitting/merging and pipeline methods.

6.1 Adaptivity in Advanced Multi-core Hardware Platforms

Modern data analysis algorithms, such as those relying on neural networks and deep learning, are characterized by critical demands in terms of computing power. They are composed of multiple layers and each layer is usually processing *tensors*. Thus, such algorithms intrinsically expose additional parallelism to be exploited when more processing elements are available.

To comply with this complexity, we extended the application model described in Section 3.5, enabling representation of lower-level building components of the *Process* tasks. For example, individual layers composing a *Process* task representing a CNN, can be represented themselves as single tasks and communicate with each other through FIFOs. In this way, tasks can be independently mapped to physical cores and throughput can be improved.

Figure 6.1 shows an example where each layer in a CNN is mapped to an independent core. The lower part of the figure shows a legend explaining how layers, cores, and FIFOs are represented. In this case, the throughput is obviously determined by the layer with a longer execution time, which limits the pipeline rate.

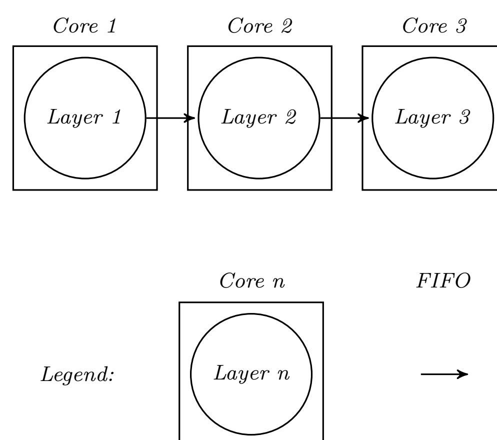


Figure 6.1: One core for each convolutional layer.

6.1.1 ADAM for Multi-Cores

When multi-core platforms are enabled, ADAM can act to change the mapping and partitioning of the software tasks, to create a software pipeline configuration

that optimally fits with the required workload. The objective is to balance pipeline stages and to set up an optimal frequency-voltage operating point for the platform.

We have defined a workload-partitioning mechanism, called *splitting*, that enables to divide a single task to be executed in parallel on several cores, as depicted in Figure 6.2, to reduce the duration of a limiting pipeline stage and to improve the overall throughput. In this case, ADAM, depending on a selected optimization policy, is in charge to activate, when needed, a set of supporting cores sharing the initial workload, called *helpers*. ADAM activates and deactivates the core helpers in total autonomy, so it knows at all times how many cores are available and how many are occupied. The lower part of the figure shows a legend explaining how blocks, cores, and FIFOs are represented.

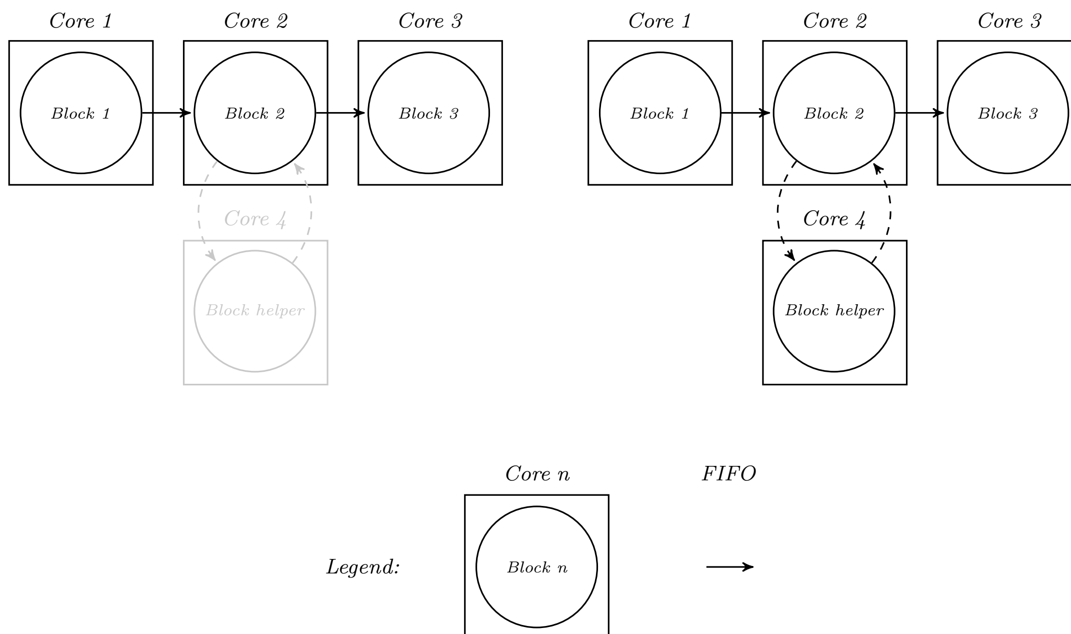


Figure 6.2: Subdivision of the workload given by the *blocks* into several cores.

At the moment, we have tested the splitting technique on Orlando, referring to the typical structure of neural network layers as a specific application use case. The splitting of a layer with one or more *helpers* is operated asking the *helpers* to compute part (half in case of one single *helper*) of the layer's output features.

6.2 Splitting policies

ADAM can implement different policies that may be used to combine splitting and frequency/voltage scaling to dynamically adapt to changing workloads.

In this work, we have implemented two policies:

- a first policy, that we call ADAM Frequency First (FF), which tries to minimize the working system frequency as the main objective,
- a second policy, that we call ADAM Idle First (IF), which is more indicated for systems that have less reactive frequency management, which tries to set as many processing elements as possible in sleep mode.

Both policies envision the system to be set, at the start-up, in a mapping configuration, called hereafter *baseline* setup, that balances pipeline stages as much as possible. To this aim, we merge tasks (layers in our CNN-related experiments) in *blocks*, until we obtain groups that are as similar as possible to each other in terms of execution time. The merging of the operators in one single block is done at design time. The merged block is represented as a single process network, thus there is no performance degradation due to the scheduling of multiple nodes on the same processing element. Figure 6.3 shows an example of a balanced pipeline: the first and second layers are processed by core 1, the third layer is processed by core 2, and core 3 instead processes layers 4 and 5, merged together in one single *block*. As can be seen from Figure 6.3 and for all subsequent figures, the first layer, belonging to the first block, is not clearly visible on core 1, due to the computational complexity, the execution time is extremely reduced compared to the other layers. Again, Figure 6.3 and the figures representing the timing of the pipeline, highlight the first stage of the pipeline of each core. What each core does after the first pipeline stage is nothing more than the same work repeated for all the other stages. In Figure 6.3 (and similar ones) the pipeline stages following the first one are shown in grey, in order to make the graph more readable, while the empty spaces indicate the sleep state of the cores.

We envision the definition of the baseline setup to be identified offline, at design time, by manual profiling or using adequate existing system-level design tools, such as those described in Pimentel [105] and Meloni et al. [106].

At this point, thanks to the splitting mechanism, it's possible to divide the workload of each task (being an independently mapped layer or a *block*) between several cores, activating one or more *helpers*. Figure 6.4 shows an example of how this is used to reduce the execution time of the three stages of the pipeline (the first stage is shared by three cores, the second and third stages by two cores each).

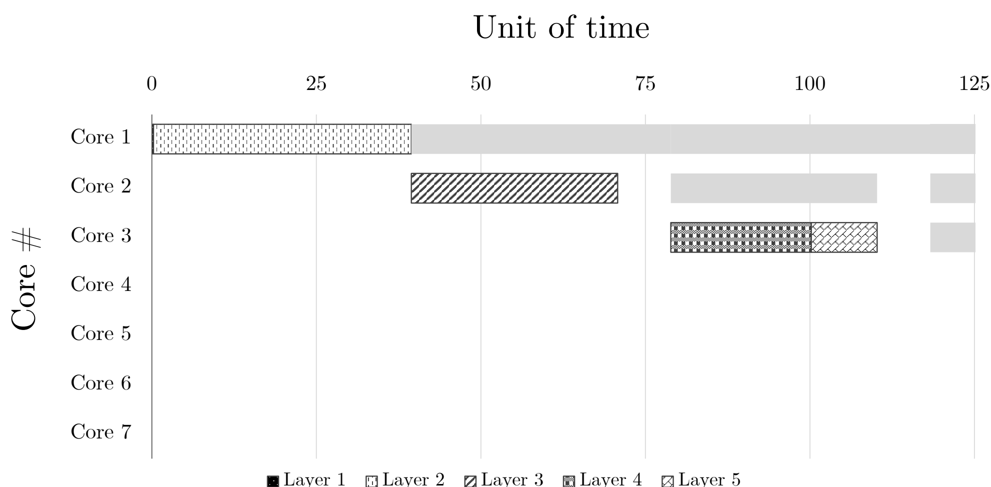


Figure 6.3: Example of balanced pipeline (grey shadows show the potential execution of successive pipelined computations of the same application).

Reducing the limiting length of the stages the pipeline can switch at a higher rate, thus the throughput is improved.

ADAM-FF policy is outlined in Algorithm 6.1. In this policy, the system starts from the baseline setup and, independently from the actual workload to be supported, applies splitting iteratively to throughput-limiting tasks until all available processing elements are used as *helpers*. After this phase ADAM enters in a routine, that may be triggered by a timer (as in the pseudo-code) or by other external events. It monitors the workload and increases the system frequency (adapting the voltage accordingly) when a higher performance level is required to support real-time constraints or reduces it when constraints are more relaxed.

ADAM-IF is outlined in Algorithm 6.2. Again, the system starts from the baseline. It sets the system frequency to be capable of respecting worst-case real-time constraints, corresponding to the highest workload, when the maximum splitting of the tasks is applied. At this point, ADAM keeps monitoring the workload and adds or removes *helpers* to meet the needs posed by real-time at any monitoring step. When constraints are relaxed, the system uses the minimum number of cores, leaving the others to wait when to be activated as *helpers*, moving them from idle to active state only when more performance is needed.

6.2.1 Splitting model on Orlando

As already described in the Section 3.1.2 and as visible in Figure 3.6, Orlando chip mounts 8 clusters containing: 2 DSPs, 2 instruction cache memories (each

```
1  setup baseline;
2  while there are cores available do
3      locate the bottleneck;
4      apply the splitting mechanism on the bottleneck;
5  end while
6  while true do
7      wait trigger from periodic timer;
8      check workload;
9      if real-time constraints are not respected then
10         Increase system frequency;
11         Increase supply voltage;
12     else
13         Decrease system frequency;
14         Decrease supply voltage;
15     end if
16 end while
```

Algorithm 6.1: ADAM-FF policy algorithm.

```
1  setup baseline;
2  set system frequency and supply voltage in worst-case;
3  while true do
4      wait trigger from periodic timer;
5      check workload;
6      if real-time constraints are not respected then
7          enable helper cores;
8      else
9          disable helper cores;
10     end if
11 end while
```

Algorithm 6.2: ADAM-IF policy algorithm

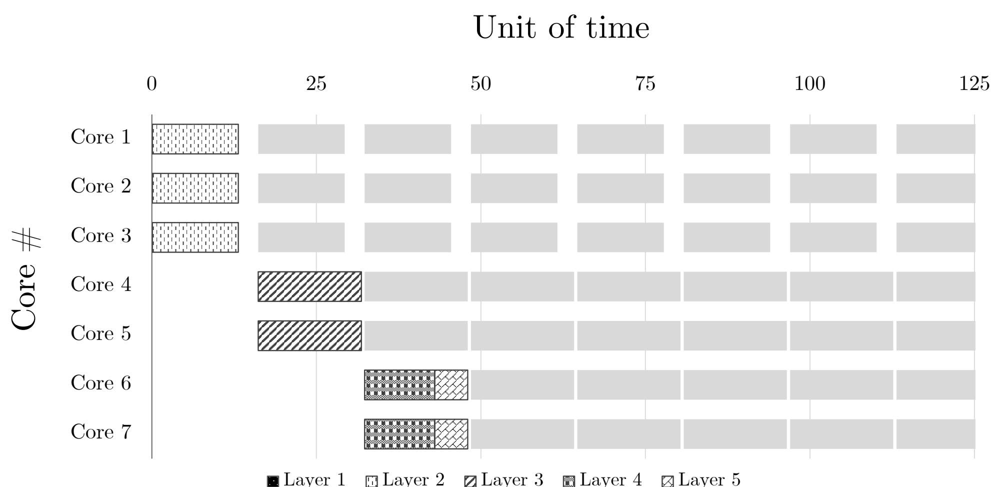


Figure 6.4: Example of redistribution of the workload over multiple cores (grey shadows show the potential execution of successive pipelined computations of the same application).

of 16 *kB*) 2 local 64 *kB* memories, and a memory of 64 *kB* shared between the two DSPs. In general, it's possible to exploit these local cluster memories in order to optimize access to memory using the layer-level strategy mentioned in Wang et al. [81]. This optimization has not been possible in this case due to the size of data (CNN weights) which forces the adoption of the 4×1 MB SRAM banks.

In order to better explain the splitting method adopted on Orlando, the function `rpc_call()` seen in the Table 3.4 will be better described. The Table 6.1 describes the input parameters associated with the aforementioned function.

Each *block* is associated with one or more `rpc_call()` functions, generally one for each layer of the neural network. For example, in `*func` a convolutional function pointer is specified. In `varargs` the structure containing all the data pointers useful for the convolution is provided. If the n-th core is in a sleep state, once an RPC function is executed, by assigning the value n to the `core_helper` variable, the n-th core is awakened from the sleep state and performs the function specified in `*func`.

The steps to enable core *helper* activation and workload splitting will be described in the following list:

1. The ADAM system constantly calculates how many *helper* cores must be enabled for each i-th *block*.

2. The core dedicated to the i-th *block* is constantly informed by the ADAM

Input parameter	***Description***
`int flags`	The first parameter specifies how the function is executed. Between the two mainpossibilities we find: - RPC_SYNC, request will be blocking until completion. - RPC_ASYNC, request will be executing asynchronously.
`void *func`	It's the pointer to the function to be executed on the specified core.
`int core_caller`	Indication of the core that executed the `rpc_call` function.
`int core_helper`	Core on which the pointed function will be executed.
`int n_parameters`	Number of parameters that the pointed function takes as input.
`int *ret`	Pointer to the return variable of the specified function.
`varargs`	Input parameters to the previously specified function.

Table 6.1: `rpc_call(...)` function arguments.

system on how many *helper* cores are assigned to its *block*. Within this core, as many `rpc_call()` functions are performed as there are *helper* cores specified by ADAM on the *i*-th *block*.

3. Furthermore, the core dedicated to the *i*-th *block* will take care of passing data in a coherent way to the *helper* core. For example, if two *helper* cores are assigned for the *i*-th *block*, the convolutional kernel pointer of each `rpc_call()` function that awakens a core *helper* will be changed, in particular, a third of the kernels will be associated with each core *helper* (so that each core *helper* calculates one-third of the output features) and the remaining third is used within the calling core.

6.3 Operating mode

In order to test the effectiveness of our system, we propose an architecture that allows constant monitoring of a patient's ECG signal, capable of detecting cardiac anomalies, by exploiting artificial intelligence techniques. Connected to the ADC of the reference platform, Orlando, there is the AD8232[1] sensor module developed by Analog Devices. For the considered application model, several operating modes

[1] https://www.analog.com/en/products/ad8232.html

have been originally envisioned [5]. In this work, besides targeting an advanced multi-core hardware platform, for which ADAM has been extended, we focus on the CNN processing operating mode.

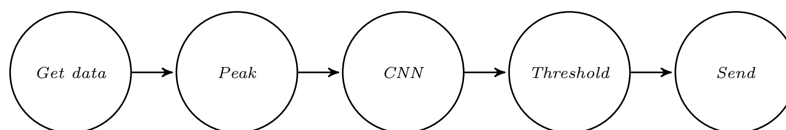


Figure 6.5: ECG application model for the CNN processing operating mode [5]

We identified five different task types with respect to the selected use case:

- *Get data*: takes care of acquiring the signal from the AD8232 module.
- *Peak*: analyzes the ECG signal to detect peaks and calculate the heart rate, the amount of information sent to the server is greatly reduced.
- *CNN*: using cognitive analysis based on concurrent neural networks, cardiac abnormalities are detected in the ECG tracing. Signal frames around the peaks detected by the previous processing task are considered. Also in this case, the amount of information sent to the server is greatly reduced.
- *Threshold*: decides whether or not the results from the enabled processing levels should be sent to the cloud, for example, if the heart rate is within a normal range there is no need to transmit the data.
- *Send*: packages and sends the data to the server.

The *Process data* tasks, according to the application model presented in Section 3.5, are two: *Peak* and *CNN*. The peak detection algorithm on the ECG signal is based on a derivative filter, it was chosen for its simplicity of implementation and the low computational capacity it requires. The adopted CNN recognizes anomalies on the ECG signal with an accuracy of 88%, and the inference process leads to 3 different output classes: Normal Sinus Rhythm, Atrial Fibrillation, or Other Rhythm [107]. The neural network is shown in Figure 6.6 and consists of 13 layers, each involving a one-dimensional convolution, a batch normalization, a ReLU, and a dropout stage. Only 3 layers have also a max pooling stage with a pooling size of 2 between ReLU and dropout. The overall size of the data transferred to the cloud is 6 Bytes (1 heartbeat data represented with 8 bit, 1 classification label data represented with 8 bit, 1 timestamp represented with 32 bit). This network

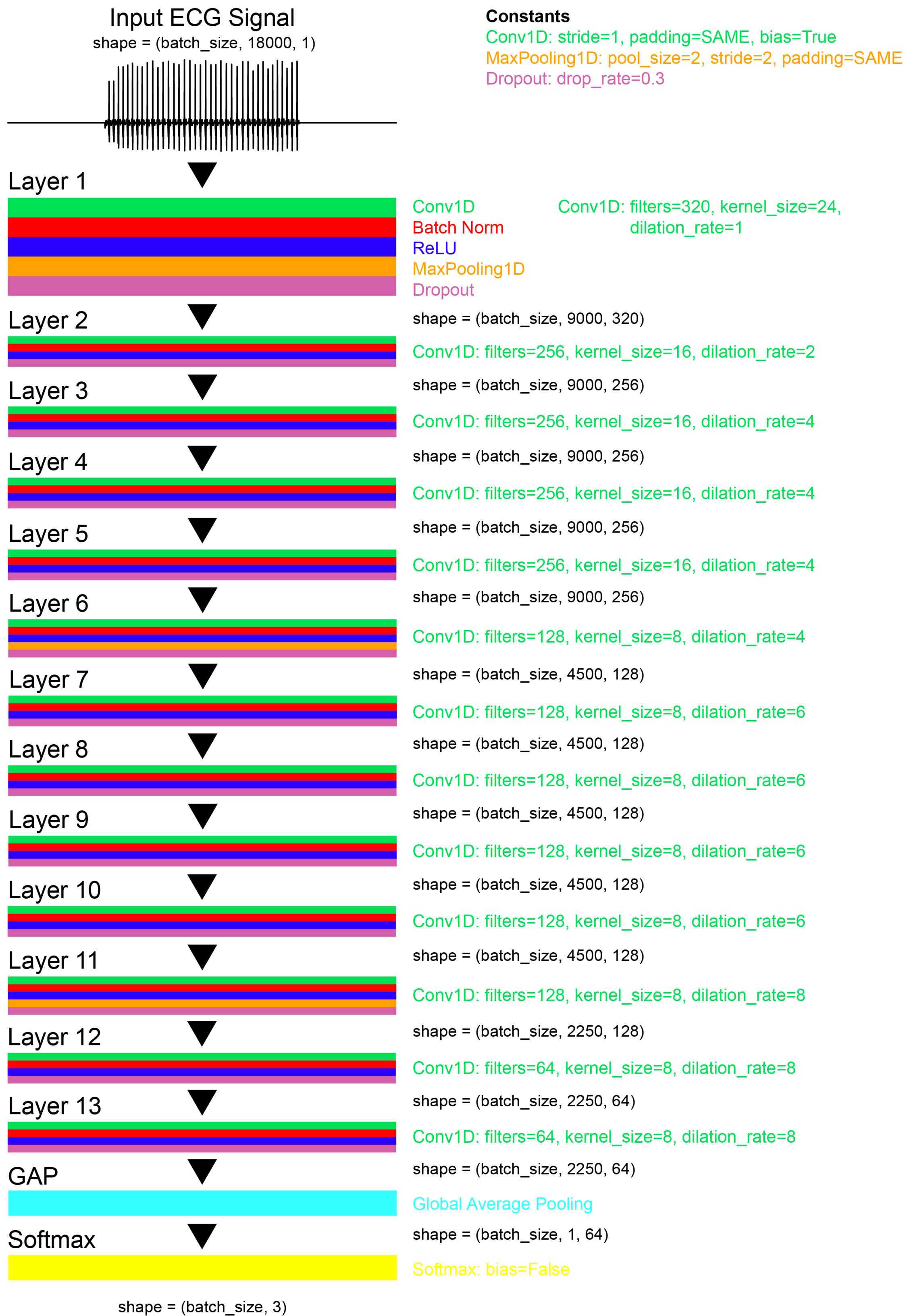


Figure 6.6: Neural network structure.

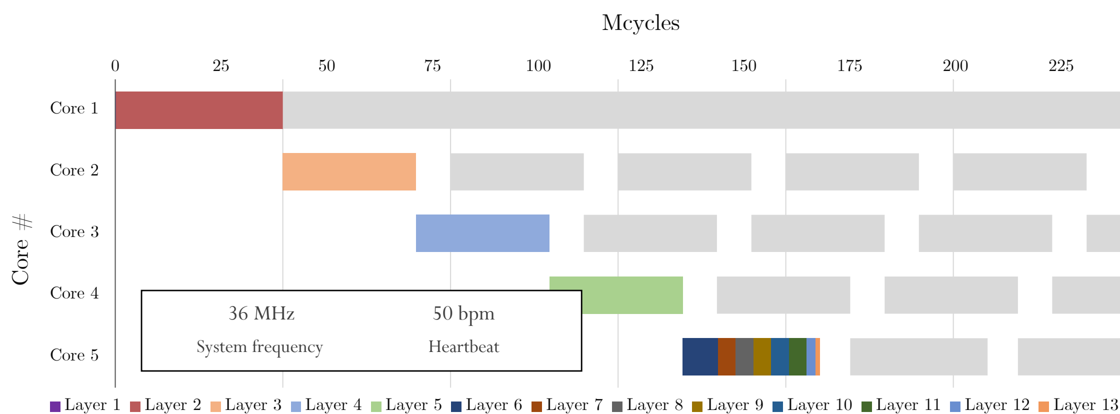


Figure 6.7: The baseline setup for the selected use case on Orlando.

requires a huge computational power, thus deeply stressing the capabilities of the adopted hardware platform. It constitutes a proper test bench for the proposed approach, which adapts the exploited level of parallelism to changing workload and operating conditions.

6.4 Experimental results

In this section, we are going to show the results obtained with the proposed extension of ADAM for multi-core platforms considering the Orlando board from STMicroelectronics [58]. To show the potential of the proposed approach, we evaluated on the target hardware platform two different conditions: single channel, considering one sensor connected to the sensing node and whose results are shown in Section 6.4.2, and multi-channel, where multiple sensors are connected to the sensing node and whose results are shown in Section 6.4.3.

6.4.1 Experimental setup

We have executed the CNN described in Section 6.3 on the Orlando multi-core platform, adopting ADAM for the dynamic adaptation of the processing, as discussed in Section 6.1.1. The baseline has been chosen manually at design time and is shown in Figure 6.7. Two layers are mapped on core 1, core 2, 3 and 4 execute one layer each, while in core 5 eight layers have been merged and mapped together. The Orlando prototyping board provides pins to measure all the device current supply, was therefore used a digital oscilloscope and a hall effect probe to evaluate

power consumption corresponding to different workload conditions. We forced the system to sustain three different workloads in order to show how the system reacts and dynamically adapts itself to this variation. For this purpose, we have adopted three workload conditions by fixing the average heart rate respectively to 50, 100, 200 *bpm*. Such values are chosen to represent the low, moderate, and high cardiac activity of a healthy individual. We have considered 200 *bpm* as the overall maximum workload to be supported by the system. For testing purposes, dummy data has been adopted in order to activate the CNN, evaluate the execution times and measure the related power consumption.

To assess the benefits of the proposed splitting technique and of its combination with dynamic voltage and frequency scaling, we have compared ADAM solutions with two more static policies, namely Fixed Topology (FT) and Static System Frequency (SSF). Overall, four policies are then considered in the reported experiments:

- FT: splitting support is not available, the application is split according to the baseline setup and, to meet real-time constraints for the maximum heart rate (200 *bpm*) a starting system frequency is also selected. The resulting mapping is kept equal during execution, while the frequency can be tuned to optimize consumption.

- SSF: no frequency scaling neither splitting support are available. The baseline setup is used for splitting application and the system frequency is then selected in order to meet real-time constraints for the given maximum heart rate (200 *bpm*). The resulting mapping and frequency are kept equal during execution.

- ADAM-FF: the proposed ADAM approach for runtime adaptation is enabled and the frequency-first policy is considered, with the main goal of minimizing the system operating frequency while meeting real-time constraints.

- ADAM-IF: the proposed ADAM approach for runtime adaptation is enabled and the idle-first policy is considered, with the main goal of minimizing the number of idle cores while meeting real-time constraints.

Please consider that the supply voltage V_{dd} is not directly considered by the proposed approach, rather it is set according to the adopted frequency, as resulting from the preliminary study shown in Section 3.1.2 which led to the frequency-voltage pairing depicted in Figure 3.7.

6.4.2 Single Channel

In ADAM-FF, splitting is used at the start-up to balance the pipeline as much as possible using all cores. At 50 *bpm*, ADAM-FF minimizes the system frequency and, to comply with real-time constraints in this condition, it is settled around 9 *MHz*[2]. When the workload increases to 100 *bpm*, ADAM-FF compensates by increasing the frequency, doubling it to around 18 *MHz*. The same happens when moving to 200 *bpm*, requiring a frequency increase to 36 *MHz*.

Thus, 36 *MHz* corresponds to the frequency required to support the worst-case workload with complete splitting. When using ADAM-IF, this value is set for all the workload levels. At 50 *bpm* the system uses the baseline mapping setup represented in Figure 6.7. 5 out of 16 processors (DSPs) are in active mode while the others are in idle state. When passing to 100 *bpm*, 6 *helpers* are activated, to create the pipeline configuration represented in Figure 6.8. Obviously, for 200 *bpm*, all cores are activated.

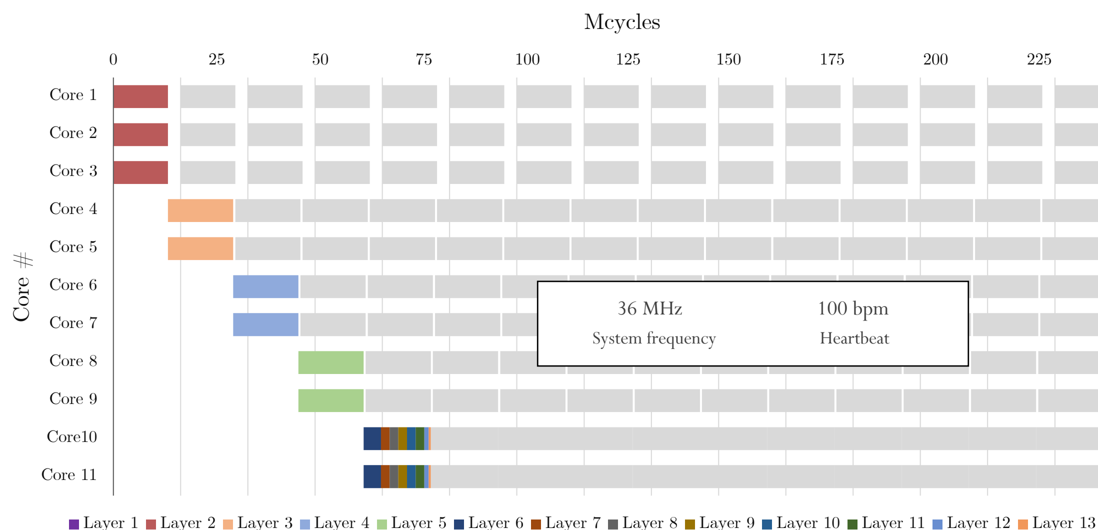


Figure 6.8: Dataflow on cores with a balanced pipeline and medium workload (ADAM-IF with maximum 100 *bpm*).

In FT, frequency is the only knob usable to comply with varying workloads. Five cores are always used, while frequency is set to 33 *MHz* for 50 *bpm*, 66 *MHz* for 100 *bpm*, and 132 *MHz* for the worst case.

Finally, in SSF, the system uses five cores and is always clocked to 132 MHz.

[2]Frequency numbers are rounded to consider that available precision in clock generation is 1 MHz.

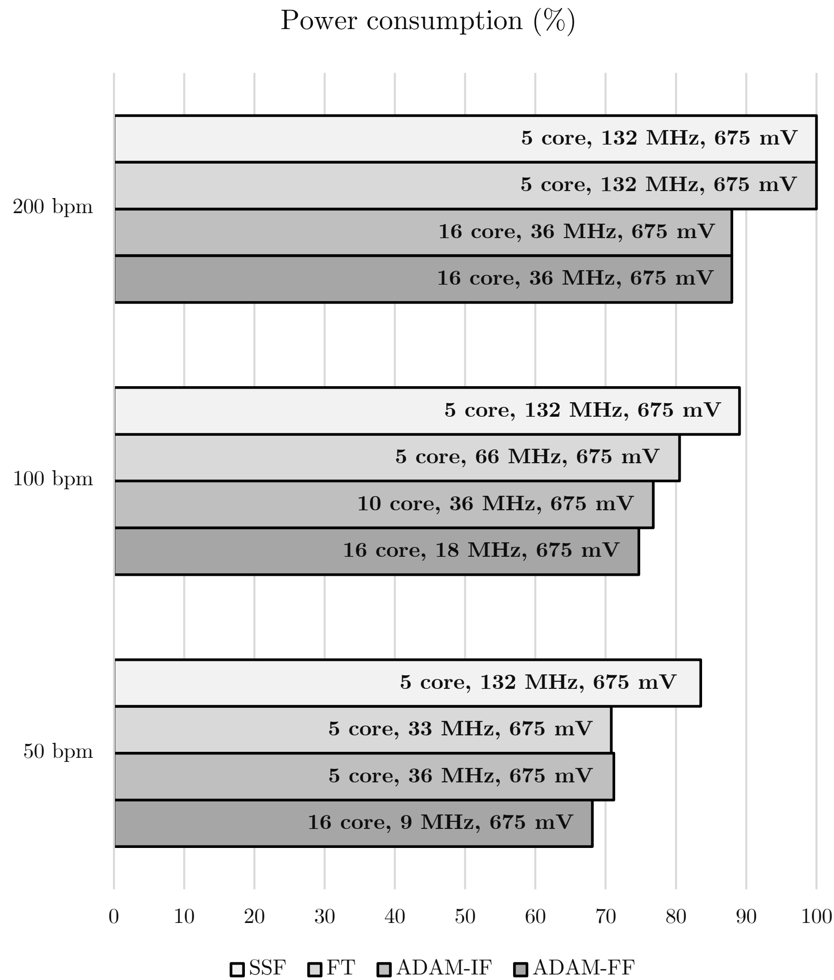


Figure 6.9: Comparison of power consumption considering different adaptation policies (with or without ADAM) and different workloads.

The results for single-channel power consumption are presented in Figure 6.9. For 50 bpm, ADAM-IF consumes slightly more than FT. In fact, the two policies lead, for a different system frequency, which in the ADAM-IF case is higher, to the use of the same number of cores. As may be noticed, in Orlando, in this case, ADAM-FF is the best solution for every workload condition. It saves around 5% on average when compared with ADAM-IF and up to 15% with respect to more static policies (SSF, FT). This is basically due to the amount of power that depends on the system frequency: by dividing the workload into several cores it's possible to achieve a significant system frequency reduction. In devices where some components cannot be completely shut down, this method can be very effective

for energy saving. When targeting devices that do not provide effective support for rapid and low-overhead frequency adaptation, changing the system frequency at runtime can be impossible. In this case, ADAM-IF can still be a good policy to save power consumption. It's possible to save up to 15% power with respect to SSF, by changing the partitioning and using splitting instead of frequency to improve performance.

Savings appear to be overall limited in the proposed experimental results. This is mainly due to the fact that for the considered use case Orlando works in a frequency region that is lower than 300 *MHz*. In this region, V_{dd} is always set to $675\,mV$, as depicted in Figure 3.7. Lower frequencies do not enable to use lower voltages, thus using splitting instead of increasing clock speed does not provide maximum benefits.

6.4.3 Multi-Channel

In order to explore the full potential of the proposed approach, we compare the proposed ADAM adaptation policies on a prospective benchmark that requires heavier workloads to be supported. For example, we can envision using Orlando to implement an embedded microserver analyzing multiple ECG channels (e.g. a single data collector in a hospital room, performing in-place analysis for all the patients). For this purpose, 6 different signals coming from 6 AD8232, each monitoring a different patient, are considered. These signals are computed by the hardware platform concurrently, stressing the available 16 cores and forcing the system to move to frequencies that imply a modification of the supply voltage. In this case, always considering the baseline setup depicted in Figure 6.7 as initial splitting for the application, the starting frequency necessary to meet real-time constraints with the maximum workload (200 *bpm*) is 789 *MHz*, which requires increasing V_{dd} to $843\,mV$ with respect to the $675\,mV$ of the single-channel experiments. This baseline setup and 789 *MHz* operating frequency are, as occurred for single-channel, the configuration adopted with the SSF policy for all the tests.

Figure 6.10 shows the results for the multi-channel experiments. ADAM-FF resulting frequencies are now equal to 52 *MHz* for 50 *bpm*, 108 *MHz* for 100 *bpm*, and 216 *MHz* for 50 *bpm*, all requiring the minimum supply voltage ($675\,mV$) and employing the whole 16 available cores. This is, again, the best policy for power consumption, reaching a saving which is more than 80% with respect to SSF in the 50 *bpm* case.

ADAM-IF, instead of minimizing frequency aims at minimizing active cores. For this reason, it employs only 5 cores for 50 *bpm*, 10 cores for 100 *bpm*, and all the 16 cores for 200 *bpm*, with an operating frequency equal to 216 *MHz*, requiring

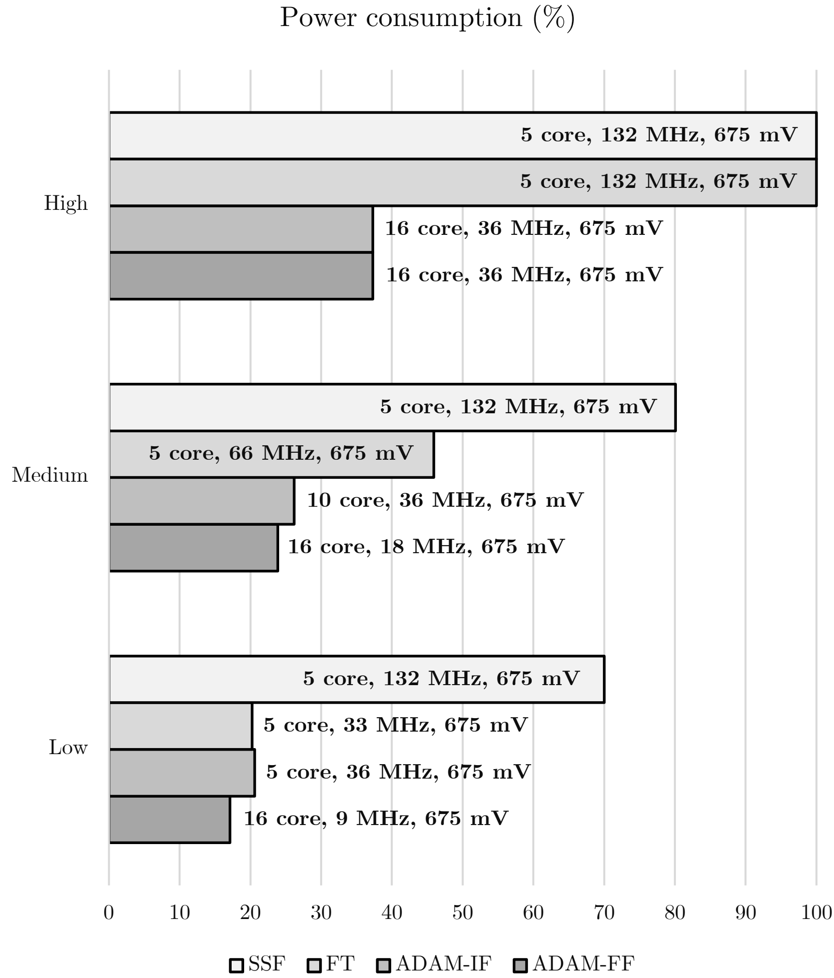


Figure 6.10: Comparison of power consumption in different ADAM configurations with high workloads (processing of six ECG streams).

the minimum supply voltage ($675\,mV$). The overall power saving of the ADAM-IF policy with respect to SSF is slightly lower than ADAM-FF, but still consistent and close to 80% in the best case. By using always the same amount of cores, 5, the FT policy is instead adopting a frequency which is 198, 395, and 789 *MHz* respectively for 50, 100, and 200 *bpm*, and requiring in turn a V_{dd} equal to $675\,mV$, $767\,mV$, and $843\,mV$. This, again, leads to a saving of the FT policy with respect to the ADAM-IF one for 50 *bpm* due to the overhead of running the same ADAM task. In any case, with higher workloads (100 and 200 *bpm*) ADAM-IF saves always more than 50% power with respect to FT.

Increasing the overall computing load within the hardware platform, for all the

considered workloads (50, 100, and 200 *bpm*), both ADAM-based policies provide much more significant savings with respect to non-splitting policies. This is true especially considering the highest workload cases: up to 60% power reduction is achieved when higher performance is required.

7 Conclusion

A hardware/software template for the development of a dynamically manageable IoMT node has been defined. It has been designed to execute in-place analysis of the sensed physiological and sensorimotor data. Its implementation has been tested on a low-power platform, able to exploit a CNN-based data analysis to recognize anomalies on ECG traces and to detect some specific physical exercises on a wobble board. The device is able to reconfigure itself according to the required operating modes and workload. The ADAM component, able to manage the reconfiguration of the device, plays a substantial role in energy saving. A quantized neural network reaches an accuracy value higher than 97% on MIT-BIH Arrhythmia dataset for NLRAV and NSVFQ diseases classification. An energy-saving up to 50% was measured by activating in-place analysis and managing the hardware and software components of the device. Very similar results were also obtained using fitness tracking on a wobble board. On a custom dataset, the quantized neural network achieves an accuracy value equal to 97.652%. Also in this case, in-place analysis and runtime reconfiguration leads to good results in terms of energy consumption, saving up to 60%. This approach has also been validated in multi-core platforms, dynamic workload management is based on pipelining techniques and parallelization of computation across the available cores. Once again, runtime management of reconfiguration leads to energy-saving benefits of at least 15%. In this thesis a proof of concept systems are presented. One of the main limitations of our system comes from the movement artifacts, since

we have chosen an acquisition configuration having a filtering that allows a low distortion of the signal, our system is therefore addressable in a hospital environment, where the patient remains relatively still during monitoring. An interesting future work is to make the most of the capabilities of the neural network to be able to recognize motion artifacts in the signal without having to manually search for the features that need to be monitored. Another step forward can be the validation our methodology with that provided by the ANSI/AAMI EC57:2012 or BHD standards for recognition of arrhythmias on ECGs. In conclusion, this thesis demonstrates how to increase battery life with near-sensor processing and highlights the importance of runtime management on data-dependent systems.

Repositories

Firmware integrating ADAM component for arrhythmias detection:
https://github.com/matteoscrugli/adam-iot-node-on-stm32l4

CNN training using static quantization for arrhythmias detection:
https://github.com/matteoscrugli/ecg-classification-quantized-cnn

CNN training using static quantization for sensorimotor exercises detection:
https://github.com/matteoscrugli/deepwobbleboard

Bibliography

[1] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, 2018. [Online]. Available: http://arxiv.org/abs/1801.06601 (Cited on pages vii, 28, 29, and 30)

[2] S. Jayaraman, V. Sangareddi, R. Periyasamy, and J. Joseph, "Modified limb lead ecg system effects on electrocardiographic wave amplitudes and frontal plane axis in sinus rhythm subjects," *The Anatolian Journal of Cardiology*, vol. 17, pp. 46–54, 01 2017. (Cited on pages vii and 44)

[3] S. E. Asp, K. Halldòrsdòttir, C. Hägg, M. L. Møller, B. P. Mickelsson, L. Boldt, and D. Skaarup, Eds., *WobbleActive*, 2007. (Cited on pages ix, 12, and 82)

[4] N. Nilsson, S. Serafin, and R. Nordahl, "Gameplay as a source of intrinsic motivation for individuals in need of ankle training or rehabilitation," *Teleoperators and Virtual Environments - Presence*, vol. 21, pp. 69–84, 02 2012. (Cited on pages ix, 10, and 82)

[5] M. A. Scrugli, D. Loi, L. Raffo, and P. Meloni, "A runtime-adaptive cognitive iot node for healthcare monitoring," 04 2019, pp. 350–357. (Cited on pages ix and 99)

[6] E. J. Benjamin, P. Muntner, A. Alonso, M. S. Bittencourt, C. W. Callaway, A. P. Carson, A. M. Chamberlain, A. R. Chang, S. Cheng, S. R. Das *et al.*, "Heart disease and stroke statistics—2019 update: a report from the american heart association," *Circulation*, vol. 139, no. 10, pp. e56–e528, 2019. (Cited on page 2)

[7] E. Wilkins, L. Wilson, K. Wickramasinghe, P. Bhatnagar, J. Leal, R. Luengo-Fernandez, R. Burns, M. Rayner, and N. Townsend, "European cardiovascular disease statistics 2017," 2017. (Cited on page 2)

[8] R. Maskeliūnas, R. Damaševičius, and S. Segal, "A review of internet of things technologies for ambient assisted living environments," *Future Internet*, vol. 11, no. 12, 2019. [Online]. Available: https://www.mdpi.com/1999-5903/11/12/259 (Cited on pages 2 and 6)

[9] Z. Yang, Q. Zhou, L. Lei, K. Zheng, and W. Xiang, "An iot-cloud based wearable ecg monitoring system for smart healthcare," *Journal of Medical Systems*, vol. 40, no. 12, p. 286, Oct 2016. [Online]. Available: https://doi.org/10.1007/s10916-016-0644-9 (Cited on page 6)

[10] L. Roberts, P. Michalák, S. Heaps, M. Trenell, D. Wilkinson, and P. Watson, "Automating the placement of time series models for iot healthcare applications," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, Oct 2018, pp. 290–291. (Cited on page 6)

[11] S. Macis, D. Loi, D. Pani, L. Raffo, S. L. Manna, V. Cestone, and D. Guerri, "Home telemonitoring of vital signs through a tv-based application for elderly patients," in *2015 IEEE International Symposium on Medical Measurements and Applications (MeMeA) Proceedings*, May 2015, pp. 169–174. (Cited on page 6)

[12] K. Kaewkannate and S. Kim, *The Comparison of Wearable Fitness Devices*, 10 2018. (Cited on page 6)

[13] K. Kaewkannate and S.-C. Kim, "A comparison of wearable fitness devices," *BMC Public Health*, vol. 16, pp. 1–16, 2016. (Cited on page 6)

[14] R. S. Romaniuk, "Iot – review of critical issues," *International Journal of Electronics and Telecommunications*, vol. vol. 64, no. No 1, 2018. [Online]. Available: http://journals.pan.pl/Content/102764/PDF/IJET_1_2018_15_1210.pdf (Cited on page 6)

[15] H. Sun, V. D. Florio, N. Gui, and C. Blondia, "Towards building virtual community for ambient assisted living," in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 2008, pp. 556–561. (Cited on page 6)

[16] H. Ajami, H. Mcheick, and K. Mustapha, "A pervasive healthcare system for copd patients," *Diagnostics*, vol. 9, no. 4, 2019. [Online]. Available: https://www.mdpi.com/2075-4418/9/4/135 (Cited on page 6)

[17] J. N. S. Rubí and P. R. L. Gondim, "Iomt platform for pervasive healthcare data aggregation, processing, and sharing based on onem2m and openehr," *Sensors*, vol. 19, no. 19, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/19/4283 (Cited on page 6)

[18] U. B. Baloglu, M. Talo, O. Yildirim, R. S. Tan, and U. R. Acharya, "Classification of myocardial infarction with multi-lead ecg signals and deep cnn," *Pattern Recognition Letters*, vol. 122, pp. 23 – 30, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016786551930056X (Cited on page 6)

[19] R. D. Labati, E. Muñoz, V. Piuri, R. Sassi, and F. Scotti, "Deep-ecg: Convolutional neural networks for ecg biometric recognition," *Pattern Recognition Letters*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865518301077 (Cited on page 6)

[20] Y. Li, Y. Pang, J. Wang, and X. Li, "Patient-specific ecg classification by deeper cnn from generic to dedicated," *Neurocomputing*, vol. 314, pp. 336 – 346, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231218308063 (Cited on page 7)

[21] K. M. R. Tabal, F. S. Caluyo, and J. B. G. Ibarra, "Microcontroller-implemented artificial neural network for electrooculography-based wearable drowsiness detection system," in *Advanced Computer and Communication Engineering Technology*, H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, and N. C. Pee, Eds. Cham: Springer International Publishing, 2016, pp. 461–472. (Cited on page 7)

[22] M. Magno, M. Pritz, P. Mayer, and L. Benini, "Deepemote: Towards multi-layer neural networks in a low power wearable multi-sensors bracelet," in *2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI)*, June 2017, pp. 32–37. (Cited on page 7)

[23] S. Lee, J. Hong, C. Hsieh, M. Liang, S. Chang Chien, and K. Lin, "Low-power wireless ecg acquisition and classification system for body sensor networks," *IEEE Journal of Biomedical and Health Informatics*, vol. 19, no. 1, pp. 236–246, 2015. (Cited on page 7)

[24] T. Chen, E. B. Mazomenos, K. Maharatna, S. Dasmahapatra, and M. Niranjan, "Design of a low-power on-body ecg classifier for remote cardiovascular monitoring systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 1, pp. 75–85, 2013. (Cited on page 7)

[25] N. Bayasi, T. Tekeste, H. Saleh, B. Mohammad, A. Khandoker, and M. Ismail, "Low-power ecg-based processor for predicting ventricular arrhythmia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 5, pp. 1962–1974, 2016. (Cited on page 7)

[26] T. Ince, S. Kiranyaz, and M. Gabbouj, "A generic and robust system for automated patient-specific classification of ecg signals," *IEEE Transactions on Biomedical Engineering*, vol. 56, no. 5, pp. 1415–1426, 2009. (Cited on page 7)

[27] A. Amirshahi and M. Hashemi, "Ecg classification algorithm based on stdp and r-stdp neural networks for real-time monitoring on ultra low-power personal wearable devices," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 6, pp. 1483–1493, 2019. (Cited on page 7)

[28] E. Kolağasioğlu, "Energy efficient feature extraction for single-lead ecg classification based on spiking neural networks," 2018. (Cited on page 7)

[29] G. R. Deshmukh and U. M. Chaskar, "Iot enabled system design for real-time monitoring of ecg signals using tiva c-series microcontroller," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2018, pp. 976–979. (Cited on page 8)

[30] U. Arun, S. Natarajan, and R. R. Rajanna, "A novel iot cloud-based real-time cardiac monitoring approach using ni myrio-1900 for telemedicine applications," in *2018 3rd International Conference on Circuits, Control, Communication and Computing (I4C)*, 2018, pp. 1–4. (Cited on page 8)

[31] U. Satija, B. Ramkumar, and M. Sabarimalai Manikandan, "Real-time signal quality-aware ecg telemetry system for iot-based health care monitoring," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 815–823, 2017. (Cited on page 8)

[32] G. Xu, "Iot-assisted ecg monitoring framework with secure data transmission for health care applications," *IEEE Access*, vol. 8, pp. 74 586–74 594, 2020. (Cited on page 8)

[33] V. Natarajan and A. Vyas, "Power efficient compressive sensing for continuous monitoring of ecg and ppg in a wearable system," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 336–341. (Cited on page 8)

[34] E. Spanò, S. Di Pascoli, and G. Iannaccone, "Low-power wearable ecg monitoring system for multiple-patient remote monitoring," *IEEE Sensors Journal*, vol. 16, no. 13, pp. 5452–5462, 2016. (Cited on page 8)

[35] H. Ghasemzadeh and R. Jafari, "Ultra low-power signal processing in wearable monitoring systems: A tiered screening architecture with optimal bit resolution," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 1, pp. 9:1–9:23, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501626.2501636 (Cited on page 8)

[36] T. Tekeste, H. Saleh, B. Mohammad, and M. Ismail, "Ultra-low power qrs detection and ecg compression architecture for iot healthcare devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 2, pp. 669–679, Feb 2019. (Cited on page 8)

[37] C. Wang, Y. Qin, H. Jin, I. Kim, J. D. Granados Vergara, C. Dong, Y. Jiang, Q. Zhou, J. Li, Z. He, Z. Zou, L. Zheng, X. Wu, and Y. Wang, "A low power cardiovascular healthcare system with cross-layer optimization from sensing patch to cloud platform," *IEEE Transactions on Biomedical Circuits and Systems*, pp. 1–1, 2019. (Cited on page 8)

[38] M. K. Adimulam and M. B. Srinivas, "Ultra low power programmable wireless exg soc design for iot healthcare system," in *Wireless Mobile Communication and Healthcare*, P. Perego, A. M. Rahmani, and N. TaheriNejad, Eds. Cham: Springer International Publishing, 2018, pp. 41–49. (Cited on page 8)

[39] K. G. Rani Roopha Devi, R. Mahendra Chozhan, and R. Murugesan, "Cognitive iot integration for smart healthcare: Case study for heart disease detection and monitoring," in *2019 International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC)*, 2019, pp. 1–6. (Cited on pages 8, 9, and 11)

[40] S. Sakib, M. M. Fouda, Z. M. Fadlullah, and N. Nasser, "Migrating intelligence from cloud to ultra-edge smart iot sensor based on deep learning: An arrhythmia monitoring use-case," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, 2020, pp. 595–600. (Cited on pages 8, 9, 11, 77, and 78)

[41] M. Deshmane and S. Madhe, "Ecg based biometric human identification using convolutional neural network in smart health applications," in *2018*

Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), 2018, pp. 1–6. (Cited on page 8)

[42] A. Walinjkar and J. Woods, "Personalized wearable systems for real-time ecg classification and healthcare interoperability: Real-time ecg classification and fhir interoperability," in *2017 Internet Technologies and Applications (ITA)*, 2017, pp. 9–14. (Cited on page 9)

[43] Y. Xitong, D. Yu, and Z. Jianxun, "A real - time ecg signal classification algorithm," in *2020 39th Chinese Control Conference (CCC)*, 2020, pp. 7356–7361. (Cited on pages 9 and 11)

[44] M. Naz, J. Shah, M. Khan, M. Sharif, M. Raza, and R. Damasevicius, "From ecg signals to images: a transformation based approach for deep learning," *PeerJ Computer Science*, vol. 7, 02 2021. (Cited on pages 9 and 11)

[45] C. Ma, X. Mu, and D. Sha, "Multi-layers feature fusion of convolutional neural network for scene classification of remote sensing," *IEEE Access*, vol. 7, pp. 121 685–121 694, 2019. (Cited on page 9)

[46] I. Azimi, J. Takalo-Mattila, A. Anzanpour, A. M. Rahmani, J. Soininen, and P. Liljeberg, "Empowering healthcare iot systems with hierarchical edge-based deep learning," in *2018 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, 2018, pp. 63–68. (Cited on pages 9, 11, and 78)

[47] A. Burger, C. Qian, G. Schiele, and D. Helms, "An embedded cnn implementation for on-device ecg analysis," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2020, pp. 1–6. (Cited on pages 10, 11, and 78)

[48] L.-R. Yeh, W.-C. Chen, H.-Y. Chan, N.-H. Lu, C.-Y. Wang, W.-H. Twan, W.-C. Du, Y.-H. Huang, S.-Y. Hsu, and T.-B. Chen, "Integrating ecg monitoring and classification via iot and deep neural networks," *Biosensors*, vol. 11, no. 6, 2021. [Online]. Available: https://www.mdpi.com/2079-6374/11/6/188 (Cited on pages 10 and 11)

[49] S. M. Mathews, C. Kambhamettu, and K. E. Barner, "A novel application of deep learning for single-lead ecg classification," *Computers in Biology and Medicine*, vol. 99, pp. 53–62, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010482518301264 (Cited on pages 10 and 11)

[50] G. Sannino and G. De Pietro, "A deep learning approach for ecg-based heartbeat classification for arrhythmia detection," *Future Generation Computer Systems*, vol. 86, pp. 446–455, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17324548 (Cited on pages 10 and 11)

[51] S. Kiranyaz, T. Ince, and M. Gabbouj, "Real-time patient-specific ecg classification by 1-d convolutional neural networks," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 3, pp. 664–675, 2016. (Cited on pages 10 and 11)

[52] D. Hou, M. Raymond Hou, and J. Hou, "Ecg beat classification on edge device," in *2020 IEEE International Conference on Consumer Electronics (ICCE)*, 2020, pp. 1–4. (Cited on pages 10, 11, and 78)

[53] World Health Organization (WHO). Global recommendations on physical activity for health. Accessed: 2021-11-11. [Online]. Available: https://www.who.int/dietphysicalactivity/global-PA-recs-2010.pdf (Cited on page 10)

[54] H. Abedtash and R. J. Holden, "Systematic review of the effectiveness of health-related behavioral interventions using portable activity sensing devices (PASDs)," *Journal of the American Medical Informatics Association*, vol. 24, no. 5, pp. 1002–1013, 02 2017. [Online]. Available: https://doi.org/10.1093/jamia/ocx006 (Cited on page 10)

[55] B. Blažica and P. Krivec, "Olok boardy – gamified sensorimotor training with affordable smart balance board," in *3rd Annual Scientific and Professional International Conference "Health of Children and Adolescent"*, September 2019, p. 185. [Online]. Available: https://www.hippocampus.si/ISBN/978-961-7055-73-3.pdf (Cited on page 10)

[56] N. Maclean and P. Pound, "A critical review of the concept of patient motivation in the literature on physical rehabilitation." *Social science & medicine*, vol. 50 4, pp. 495–506, 2000. (Cited on page 10)

[57] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2018, pp. 1–4. (Cited on page 12)

[58] G. Desoli, N. Chawla, T. Boesch, S. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal, "14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 238–239. (Cited on pages 12 and 101)

[59] Google®. (2020) Google tpu. [Online]. Available: https://cloud.google.com/tpu (Cited on page 12)

[60] NVIDIA®. (2019) Embedded systems for next-generation autonomous machines. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (Cited on page 12)

[61] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "Neuraghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on zynq socs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–24, 2018. (Cited on page 12)

[62] K. Vissers, "Versal: The xilinx adaptive compute acceleration platform (acap)," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 83. [Online]. Available: https://doi.org/10.1145/3289602.3294007 (Cited on page 12)

[63] A. Przybył, "Fixed-point arithmetic unit with a scaling mechanism for fpga-based embedded systems," *Electronics*, vol. 10, no. 10, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/10/1164 (Cited on page 12)

[64] NVIDIA®. (2020) Nvidia cudnn. [Online]. Available: https://developer.nvidia.com/cudnn (Cited on page 12)

[65] arm Developer, "Cortex microcontroller software interface standard," 2016. [Online]. Available: https://developer.arm.com/tools-and-software/embedded/cmsis (Cited on page 12)

[66] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," 2021. (Cited on page 13)

[67] S. Ward-Foxton, "Artificial intelligence gets its own system of numbers," Available online: https://www.eetimes.com/artificial-intelligence-gets-its-own-system-of-numbers/, 2020. (Cited on page 13)

[68] P. Dziwiński, A. Przybył, P. Trippner, J. Paszkowski, and Y. Hayashi, "Hardware implementation of a takagi-sugeno neuro-fuzzy system optimized by a population algorithm," *Journal of Artificial Intelligence and Soft Computing Research*, vol. 11, no. 3, pp. 243–266, 2021. [Online]. Available: https://doi.org/10.2478/jaiscr-2021-0015 (Cited on page 13)

[69] M.-L. Boukhanoufa, "Adaptabilité et reconfiguration des systèmes temps-réel embarqués," Theses, Université Paris Sud - Paris XI, Sep. 2012. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00758807 (Cited on page 13)

[70] X. Wang, M. Khalgui, Z. Li, and O. Mosbahi, "Automatic low-power reconfigurations of real-time embedded control systems," *Technical Reprot, Systems Control and Automation Group School of ElectroMechancial Engineering Xidian University.*, 2010. (Cited on page 13)

[71] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. (Cited on page 13)

[72] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Trans. Comput. Syst.*, vol. 5, no. 2, p. 121–150, mar 1987. [Online]. Available: https://doi.org/10.1145/13677.22723 (Cited on page 14)

[73] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, ser. CBSE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 135–144. [Online]. Available: https://doi.org/10.1145/2304736.2304759 (Cited on page 14)

[74] L. Morgenstern, P. Stefaneas, F. Lévy, A. Wyner, and A. Paschke, *Theory, Practice, and Applications of Rules on the Web: 7th International Symposium, RuleML 2013, Seattle, WA, USA, July 11-13, 2013, Proceedings*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013. [Online]. Available: https://books.google.it/books?id=Mkm6BQAAQBAJ (Cited on page 14)

[75] G. Tuveri, P. Meloni, F. Palumbo, G. P. Seu, I. Loi, F. Conti, and L. Raffo, "On-the-fly adaptivity for process networks over shared-memory platforms," *Microprocessors and Microsystems*, vol. 46, pp. 240 – 254, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933116300758 (Cited on page 14)

[76] J. Jahn and J. Henkel, "Pipelets: Self-organizing software pipelines for many-core architectures," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 1516–1521. (Cited on page 14)

[77] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld, "Adaptive task duplication using on-line bottleneck detection for streaming applications," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 163–172. [Online]. Available: https://doi.org/10.1145/2212908.2212932 (Cited on page 14)

[78] Arm, "Arm compute library," Available online: https://developer.arm.com/ip-products/processors/machine-learning/compute-library. (Cited on pages 15 and 16)

[79] OAID, "Tengine," Available online: https://github.com/OAID/Tengine. (Cited on pages 15 and 16)

[80] Tencent, "Ncnn," Available online: https://github.com/Tencent/ncnn. (Cited on pages 15 and 16)

[81] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big.little multi-core processors," 03 2019. (Cited on pages 15, 16, and 97)

[82] H.-I. Wu, D.-Y. Guo, H.-H. Chin, and R.-S. Tsay, "A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 46–49. (Cited on pages 15 and 16)

[83] H. Huang, V. Chaturvedi, G. Quan, J. Fan, and M. Qiu, "Throughput maximization for periodic real-time systems under the maximal temperature constraint," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2s, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2544375.2544390 (Cited on pages 15 and 17)

[84] H. Yu, R. Syed, and Y. Ha, "Thermal-aware frequency scaling for adaptive workloads on heterogeneous mpsocs," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–6. (Cited on pages 16 and 17)

[85] H. Yu, Y. Ha, and J. Wang, "Quality optimization of resilient applications under temperature constraints," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 9–16. [Online]. Available: https://doi.org/10.1145/3075564.3075577 (Cited on pages 16 and 17)

[86] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu, "Improving system-level lifetime reliability of multicore soft real-time systems," vol. 25, no. 6, 2017, pp. 1895–1905. (Cited on pages 16 and 17)

[87] A. Weissel and F. Bellosa, "Process cruise control: Event-driven clock scaling for dynamic power management," 10 2002. (Cited on pages 16 and 17)

[88] K. D. Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "The energy/frequency convexity rule: Modeling and experimental validation on mobile devices," 2014. (Cited on pages 16 and 17)

[89] S. M. Nabavinejad, H. Hafez-Kolahi, and S. Reda, "Coordinated dvfs and precision control for deep neural networks," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 136–140, 2019. (Cited on pages 16 and 17)

[90] M. Motamedi, D. Fong, and S. Ghiasi, "Machine intelligence on resource-constrained iot devices: The case of thread granularity optimization for cnn inference," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3126555 (Cited on pages 16 and 17)

[91] K. Bong, S. Choi, C. Kim, and H.-J. Yoo, "Low-power convolutional neural network processor for a face-recognition system," *IEEE Micro*, vol. 37, no. 6, pp. 30–38, 2017. (Cited on pages 16 and 17)

[92] G. Santoro, M. R. Casu, V. Peluso, A. Calimera, and M. Alioto, "Design-space exploration of pareto-optimal architectures for deep learning with dvfs," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. (Cited on page 17)

[93] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," 2017. (Cited on page 28)

[94] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," 2016. (Cited on page 28)

[95] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," 2016. (Cited on page 29)

[96] J. E. Hall and M. E. Hall, *Guyton and Hall textbook of medical physiology e-Book*. Elsevier Health Sciences, 2020. (Cited on page 42)

[97] A. L. Goldberger, Z. D. Goldberger, and A. Shvilkin, *Clinical electrocardiography: a simplified approach e-book*. Elsevier Health Sciences, 2017. (Cited on page 42)

[98] G. Moody and R. Mark, "The impact of the mit-bih arrhythmia database," *IEEE Engineering in Medicine and Biology Magazine*, vol. 20, no. 3, pp. 45–50, 2001. (Cited on page 43)

[99] A. Goldberger, L. Amaral, L. Glass, S. Havlin, J. Hausdorg, P. Ivanov, R. Mark, J. Mietus, G. Moody, C.-K. Peng, H. Stanley, and P. Physiobank, "Components of a new research resource for complex physiologic signals," *PhysioNet*, vol. 101, 01 2000. (Cited on page 44)

[100] D. Li, J. Zhang, Q. Zhang, and X. Wei, "Classification of ecg signals based on 1d convolution neural network," in *2017 IEEE 19th International Conference on e-Health Networking, Applications and Services (Healthcom)*, Oct 2017, pp. 1–6. (Cited on page 56)

[101] J. Pan and W. J. Tompkins, "A real-time qrs detection algorithm," *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, pp. 230–236, 1985. (Cited on page 56)

[102] V. Gupta, M. Mittal, and V. Mittal, "R-peak detection using chaos analysis in standard and real time ecg databases," *IRBM*, vol. 40, no. 6, pp. 341–354, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1959031818303166 (Cited on page 58)

[103] J. Laitala, M. Jiang, E. Syrjälä, E. K. Naeini, A. Airola, A. M. Rahmani, N. D. Dutt, and P. Liljeberg, "Robust ecg r-peak detection using lstm," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1104–1111. [Online]. Available: https://doi.org/10.1145/3341105.3373945 (Cited on page 58)

[104] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2016. [Online]. Available: https://books.google.it/books?id=Np9SDQAAQBAJ (Cited on pages 61, 66, 84, and 86)

[105] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design Test*, vol. 34, no. 1, pp. 77–90, Feb 2017. (Cited on page 94)

[106] P. Meloni, D. Loi, G. Deriu, A. D. Pimentel, D. Sapra, B. Moser, N. Shepeleva, F. Conti, L. Benini, O. Ripolles, D. Solans, M. Pintor, B. Biggio, T. Stefanov, S. Minakova, N. Fragoulis, I. Theodorakopoulos, M. Masin, and F. Palumbo, "Aloha: An architectural-aware framework for deep learning at the edge," in *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications*, ser. INTESA '18. New York, NY, USA: ACM, 2018, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/3285017.3285019 (Cited on page 94)

[107] S. Goodfellow, A. Goodwin, D. Eytan, R. Greer, M. Mazwi, and P. Laussen, "Towards understanding ecg rhythm classification using convolutional neural networks and attention mappings," 08 2018. (Cited on page 99)