POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Politecnico di Torino

Master's Degree Thesis

Energy-Efficient Deep Learning-based Heart-Rate Estimation on Wearables

Supervisors

Prof. Daniele Jahier PAGLIARI

Dr. Alessio BURRELLO

Dr. Matteo RISSO

Candidate

Noemi TOMASELLO

Academic Year 2021-2022

Abstract

Nowadays, Deep Learning (DL) is predominant in many fields like Computer Vision or Natural Language Processing, thanks to its excellent predictive performance. On the other hand, deploying a network in a real-world embedded system still poses several challenges. First, the data collected is often corrupted and can hinder a correct prediction of the network. Second, Deep Neural Networks (DNNs) are usually too big to fit the tight constraints of an embedded platform (for instance, a limited memory of few MBs) and need manual tuning to be optimized and reduce their dimensions while still achieving good accuracy.

Integrating AI predictions directly on edge devices like wearables can be really helpful in many situations where the real-time monitoring of the user is needed. For instance, Heart-Rate (HR) monitoring is becoming increasingly more linked to the analysis of PhotoPlethysmoGraphic (PPG) signals, which can be extracted from wrist-worn devices. Such a technique is portable, cheap and comfortable, compared to the previously dominant one based on Electrocardiogram (ECG), which is more intrusive and whose collection impairs the daily life of the subject. However, the benefits of the PPG-based HR monitoring do not come without downsides. The main problem of PPG is the presence of Motion Artifacts (MA) generated by the movements (especially in wrist-worn wearables) and the infiltration of light between the skin and the sensor, that create noise in the collected signal.

The focus of this work is the development of a complete system able to collect the PPG and track the HR of the subject while removing MAs. The system consists of i) a Client-Server interaction based on the Bluetooth Low Energy (LE) communication protocol to send PPG data / HR estimation to a collecting device (e.g., a smartphone), ii) a deep neural network model (specifically a Temporal Convolutional Network – TCN) optimized for the execution on the edge, and iii) a simple controller based on a Finite State Machine (FSM) that manages the collection of data, the prediction of the HR and the transmission of data. The selected edge devices are the STM32WB55 Nucleo development board, for the prototyping phase, and a real wearable device called H-Watch, developed by ETH Zurich for the final deployment. Specifically, both devices feature 1 MB of Flash Memory and a STM32WB55RGV6 System on Chip (SoC) by ST Microelectronics with two

independent cores: an ARM® Cortex™-M4 at 64MHz and ARM® Cortex™-M0+ at 32MHz, dedicated to the Bluetooth Low Energy (BLE) stack. Additionally, the H-Watch comes with different sensors, among which a pulse oximetry (MAX30101) and 6-axes IMU (LSM6DSM), allowing the collection of data.
The dataset that we used to benchmark our results is PPGDalia, the largest publicly available collection of PPG and tri-axial accelerometer data obtained during daily life activities. The neural network chosen to be deployed is a Temporal Convolutional Network, known to work well with time series-like data, such as the stream of PPG signals over time. The model is optimized for edge execution, and it consists of around 97K parameters, occupying 374.75 KB of the total 1MB memory space in the MCU. On the PPGDalia dataset, the network running on the STM32WB55 achieves a Mean Absolute Error (MAE) of 2.36 Beats per Minute (BPM) compared to the golden HR computed with an ECG band.

Table of Contents

List of Tables VII

List of Figures VIII

Acronyms XI

1 Introduction 1

2 Background 5
2.1 Deep Learning . . . 5
2.1.1 From the Perceptron to Deep Neural Networks . . . 5
2.1.2 Backprogation and stochastic gradient descent . . . 8
2.2 Convolutional Neural Networks . . . 10
2.2.1 Convolutional layers . . . 11
2.2.2 Pooling layers . . . 13
2.3 Microcontrollers and Embedded systems . . . 14
2.3.1 Embedded Neural Networks constraints . . . 16
2.4 Optimization for the Embedded Deployment of DNNs . . . 17
2.4.1 Quantization . . . 17
2.4.2 Pruning . . . 21
2.5 H-Watch . . . 22

3 Related Works 24
3.1 Classical Approaches . . . 25
3.2 Deep Learning-based approaches . . . 28
3.2.1 ActPPG . . . 29
3.2.2 Q-PPG . . . 31

4 Porting HR Detection on Wearable Devices 32
4.1 Hardware setup . . . 32
4.2 Bluetooth . . . 36

4.2.1 BLE on Hardware device - *Server* 38
4.2.2 BLE on host device - *Client* 41
4.3 STMCube.AI 42
4.4 Dalia Dataset 43
4.5 Procedure 44
4.6 Neural Network Design 47
4.6.1 Moving Average 49
4.7 GUI 49

5 Experimental Results 52
5.1 Models tested: size and complexity 52
5.2 Evaluation 53
5.3 Energy Consumption 57

6 Conclusions 63

Bibliograpy 66

List of Tables

4.1 Table of the service and characteristic UUIs for the P2P server application [73]. 39
4.2 Table of P2P profiles' specification [73]. 40
4.3 PPGDalia activities summarized. [12] 44
4.4 Architectures' structure tested 48

5.1 Results of models' complexity 53

List of Figures

2.1 Biological vs Artificial neuron [14] 5
2.2 Artificial Neural Network architecture vs Deep Neural Network [17]. 8
2.3 Gradient descent. The figure shows how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum [16]. 9
2.4 Example of convolution operation (a) and equivalent transposed convolution operation (b) for a 3 × 3 filter kernel size applied to a 4 × 4 feature map [19] 11
2.5 A sparse layer vs a fully connected layer [16] 12
2.6 Padding example with a 2 × 2 border of zeros using unit strides [21] 13
2.7 Typical structure of an embedded system [30] 15
2.8 The roofline model. FLOP stands for Floating Point Operations. . . 16
2.9 Post-Training quantization vs.Quantization-aware training [34] . . . 18
2.10 (A)Weights grouping; (B) Sparse weight matrix after pruning [36]. . 19
2.11 Quantization-aware training with Straight-Through Estimator [34] . 20
2.12 Node pruning example with masked layer. Node $a-3$ with$\alpha_3 = 0$ can be removed. [36] 21
2.13 H-Watch logic schematic [41] 22

3.1 Illustration of a PPG sensor 25

4.1 Hardware setup 33
4.2 Evaluation board .ioc configuration taken from the tool STM32CubeMX provided by STMicroelectronics 34
4.3 HWatch .ioc configuration taken from the tool STM32CubeMX provided by STMicroelectronics 35
4.4 BLE protocol stack 36
4.5 STM32CubeProgrammer tool in section Firmware Update Service used to upgrade the FUS version and flash the `BLE_stack` binary file. 38
4.6 STM32CubeAI tool Framework 42
4.7 Flowchart firmware logic 46

4.8 Illustration of a convolutional block in TEMPONet with two dilation factor $d = 4$, stride $s = 2$ and average pooling. [83] 47
4.9 Screenshot of the GUI created 50

5.1 STM32Cube.AI Inference of Target results on 5 samples taken from PPGDalia. 54
5.2 Comparison of the MAE values of all the three architectures 55
5.3 Prediction comparison between the True value of PPGDalia (blue) and the predicted values from inference on STM32WB55 (red). For visualization sake the first 256 example are shown. 56
5.4 Prediction comparison between the True value of PPGDalia and the predictions from the Moving Average algorithm. For visualization sake the first 256 example are shown. 56
5.5 Current consumption of performing inferences with the Nucleo board. 58
5.6 Current consumption of transferring a long list of floats numbers. . 58
5.7 Current consumption of running the MCU in busy waiting 59
5.8 Current consumption of Nucleo board setted to Low Power Standby mode. 59
5.9 Temporal graph when transmitting 64 samples (each consisting of 4 float values) through the BLE stack and successive 0.5 s of Low-Power mode. 60
5.10 Temporal graph when performing inference on edge, transmission of the predicted value (i.e., one float) and a successive 0.5 s of Low-Power mode. 61
5.11 Graphic scheme of the minimum inference time required for the network 61

Acronyms

DL
Deep Learning

DNNs
Deep Neural Networks

AI
Artificial Intelligence

HR
Heart Rate

PPG
photoplethysmography

ECG
Electrocardiogram

MA
Motion Arifacts

LE
Low Energy

TCN
Temporal Convolutional Network

FSM
Finite State Machine

SoC

System on Chip

BLE

Bluetooth Low Energy

MAE

Mean Absolute Error

BPM

beats per minute

GPU

Graphic processing unit

TPU

Tensor Processing Units

MAC

Multiply and accumulate

ReLU

Rectified Linear Unit

MLP

Multi-layer Perceptron

RNNs

Recurrent Neural Networks

SGD

Stochastic Gradient Descent

CNNs

Convolutional Neural Networks

i.i.d.

idependent identically distributed

MCU

Microcontroller unit

MPU

Microprocessor unit

CPU

Central processing unit

ROM

Read Only Memory

RAM

Random Access Memory

GPIOs

General Purpose input/output pins

AC/DC

Alternating current/Direct current

IoT

Internet of Things

DMIPS

Dhrystone MIPS

RISC

Reduced instruction set computer

CISC

Complex instruction set computer

ISA

Instruction-Set Architecture

OI

Operational intensity

FLOP

Floating Point operation

ALU

Arithmetic linear unit

PTQ

Post-training quantization

QAT

Quantization-aware training

STE

Straight-Through Estimator

RMS

Root Mean Square

SIMD

Single Intruction, Multiple Data

IMU

Inertial Measurement Unit

ELU

Exponential Linear Unit

LSTM

Long Short-Term Memory

ICA

Independent Component Analysis

MMV

Multiple Measurement Vector

NAS

Neural Architecture Network

GUI

Graphic User Interface

USART

Universal Synchronous Asynchronous Receiver Transmitter

I2C

Inter-Integrated Circuit

SPI

Serial Peripheral Interface

GAP

Generic Access Profile

GATT

Generic Attribute Profile

L2CAP

Logical Link Control and Adaptation Protocol

ATT

Attribute Protocol

UUID

Universally Unique Identifier

SMP

Security Manager Protocol

HCI

Host Controller Interface

LL

Link Layer

PHY

Physical Layer

ISM

Industrial, Scientific and Medical

FUS

Firmware Upgrade Service

P2P

Peer-to-peer

RMSE

Root Mean Squared Error

VMs

Virtual Machines

Chapter 1

Introduction

Over the past decades, Deep Learning has given incredible results in many different fields. Let us think about the increasingly advanced technologies that are used to make our daily life activities smarter and faster. Some of these can be found in tools that we use on a daily basis, like spam email detection in the mail providers, user likes and dislikes profiling used for marketing strategies, and voice assistants that help us automating tedious task and quickly answering to our questions. This list is far from being complete, in fact we can find even more complicated scenarios, like autonomous driving, virtual and augmented reality used both for entertainment or for education and medical training. All these kinds of deep learning use-cases require a huge computational power, usually provided by means of powerful Graphic Processing Units (GPUs) or even specific custom circuits designed specifically to deal with the heavy deep learning workloads like the Google's Tensor Processing Units (TPUs). Nevertheless, many systems need to be near the user to quickly collect and analyse data. In fact, most kind of sensors requires a short distance from theirs targets to detect usable information, unless noise or any kind of interference can ruin the data collection results. Examples can be thermometers [1], capacitive sensors [2], or optical cameras. The amount of data generated from all these edge devices is increasing fast, requiring methodologies that are able to properly analyse them.

A common solution is to rely on cloud computing technologies, where both the network's training and inference phases are offloaded to powerful servers. The training of neural networks is highly computational expensive since it needs to analyse and optimize millions of parameters several times. On the other hand, the inference step is usually less impacting in terms of space and complexity yet can be difficult to fit in hardware-constrained devices. Cloud computing can ease the burden of the edge hardware restriction but comes with other problems that do not have straightforward solutions. Based on the works [3] and [4] we can identify four main challenges:

- *Latency*, is the time required to send data from the source to the cloud that introduces delays making real-time responses critical. At the same time, transferring all the data directly to the cloud is yet an inefficient solution for resource utilization.

- *Scalabilities* issues come when multiple devices try to connect to the same cloud server making bottleneck situations a real concern. At the same time, transferring all the data directly to the cloud is yet an inefficient solution for resource utilization.

- *Privacy* is another sensible problem, especially for the end-users. They need to be warned about sensitive information like biometric data, faces or speech tracks and informed of how the data is intended to be used.

- *Energy* is required to send and receive data from the cloud server. This can significantly affect the efficiency of the entire solution, especially considering that most embedded devices are battery-powered. The key issue in terms of energy consumption is to identify the right trade off between the computation energy consumption and the transmission energy consumption.

Therefore, experts are exploring the opposite solution: porting the deep learning networks directly to edge devices addressing all four aforementioned solutions. The latency issue is solved by the proximity of the device with the data source making real-time services a possibility. To address scalability issues and network bottlenecks hierarchical architectures of edge compute nodes and cloud data centres has been proposed[3]. Finally, having the device close to the source avoids sending data over the public Internet and the data is analysed locally reducing a lot of privacy issues and security attacks.
It goes without saying that porting Artificial Intelligent (AI) solution to edge devices comes with another set of difficulties. The first thing to note is that almost all the networks created without having in mind the specific edge computing applications are too big and too complex to be deployed into embedded devices. Hence, several studies have been undertaken that exploit the innate error resiliency of deep networks that make possible to reduce their sizes and complexity making possible to be contained in the available memory space of the edge device [5], [6], [7]. These methodologies can be performed during the training or post-training and entail quantization [8] [9], which is a methodology for precision reduction that transforms floating-point computations into integer computations; pruning [10] [6], which instead is based on cardinality reduction, meaning the removal of redundant or low-informational nodes in the network architecture.
In other cases, the solution can simply be designing more lightweight architecture e.g., taking into consideration the number of multiply-and-accumulate (MAC)

operations that each layer performs, leading to the choice of simpler operations like depthwise separable convolutions over standard convolutions in CNNs models [11]. A common and notable example of edge device is represented by smartwatches and in general wrist-worn devices. These are often used in fitness-related or health monitoring activities. This represents exactly the field where this work is inserted. For many years, the de-facto standard for heart rate detection and heart rate variability monitoring used to be based on Electrocardiogram (ECG) signals. Despite providing reliable and accurate results, this kind of signal, is non-portable and intrusive requiring direct contact with the patients' bodies with different electrodes to be placed on specific parts of the body. Today, an alternative and more viable technique to ECG is represented by PhotoPlethysmoGraphy (PPG), which can be performed with cheap and very portable pulse oximeter sensors. Unfortunately, PPG data comes often with interference with different kinds of noise sources, mainly related to the so called Motion Artifacts (MA) generated by the movements of the sensor usually placed on wearable devices such as the wrist-worn ones.
State-of-the-art solution to this problem is represented by a sensor fusion approach where the PPG data are collected and processed with other kinds of sensors' data like accelerometers. These additional sources of information comes to help in mitigating the irregularities of the PPG signal due to the high motion of the activity being performed. In this context, the use of Deep Learning solutions comes to help thanks to their well known raw sensor data processing ability, which allows to find useful patterns and to generalize well the data, avoiding complicated feature engineering steps.
The monitoring of the patient is performed over a relatively short period of time, forming time-series-like data streams. A fairly new network topology that is known to work well with this kind of data is represented by Temporal Convolutional Network (TCN), which are networks that use Convolutional layers like Convolutional Neural Networks (CNNs) but that introduce two new parameters, namely causality and dilation.
This work aims to create a complete system based on a Client-Server architecture linked by Bluetooth LE communication protocol. The client is represented by an edge device, like a smartwatch where all the PPG and accelerometer data are measured and directly processed to track hear rate. Instead, the server is a generic device with relaxed hardware constraints like a laptop or a smartphone where processed data can be streamed. Bluetooth LE is a technology present in almost every device used nowadays, making the application highly portable and energy efficient. The work was extended to analyse every part of the project:

- The low-level development of a Finite State Machine (FSM) on the edge device that is in charge of collection of data, the HR estimation and the transmission of data.

- Server-Client framework based on Bluetooth LE communication. Where the client is in charge of creating a visualisation tool (i.e., a dashboard) to visualize the data from another device like a computer.
- the deployment of a deep neural network model, specifically a TCN, optimized specifically to be executed on edge device

The rest of this work is organized as follows. Chapter 2, provides a brief explanation of the technical knowledge about Deep learning frameworks, microcontroller and the optimization techniques required to integrate AI solution on edge devices. Chapter 3 details the related works to this thesis discussing about the analysis of PPG and accelerometer data with both classical algorithms and deep networks. In Chapter 4, the steps undertaken to develop the Server-Client systems are explained in details, starting from the hardware setup, the Bluetooth LE protocol explanation, the procedure description, the development of the network and the dashboard. Chapter 5 presents the results obtained with the application created, using the dataset PPGDalia[12] as benchmark to get evaluation performances. Finally, Chapter 6 concludes the work presenting future directions.

Chapter 2

Background

2.1 Deep Learning

Deep Learning represents an evolution of Machine Learning, thought to overcome the problem of classical algorithms to deal with complex tasks involving raw data (e.g, time-series or images). Deep learning uses representation learning at different levels, automatically extracting patterns and features in the representation of data by means of a cascade of non-linear modules. Starting from the raw input more and more abstract representations are extracted and combined to achieve the desired task [13].
Deep learning models are based upon the neural network abstraction. Therefore, we start with a brief review of neural networks operating principles.

2.1.1 From the Perceptron to Deep Neural Networks

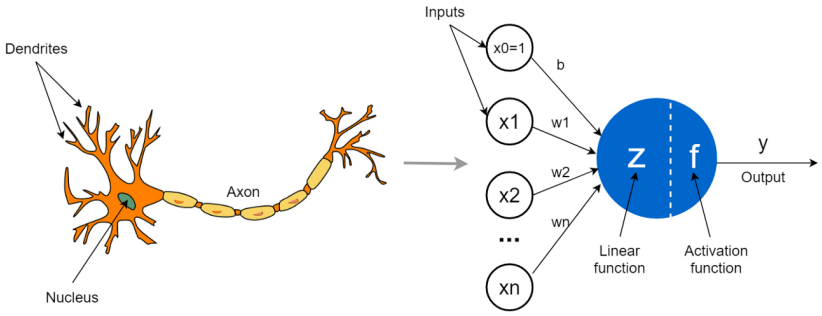


Figure 2.1: Biological vs Artificial neuron [14]

The basic unit of a Feedforward neural network is a single-layer perceptron. It was first introduced in 1958 by Rosenblatt [15] and extended ever since. The perceptron model is inspired by the human brain learning system, specifically made of a multitude of biological neurons and their connections. Nevertheless, this model does not represent by any means an attempt to model the real functionalities of the brain. Bearing this in mind, it is worth to better understand the relation between the biological and the artificial neuron.

The biological neuron is a cell that is electrical stimulated and communicates with other cells via specific connections. It is composed of a soma, the cell that combines signals, the dendrites, the combinations of inputs from other cells, the synapses, iterations between neurons, the axon, an elongated fibre that extends from the soma to the terminating endings and transmits electrical impulses along its length, the axon hillock, the part that connects the soma with the axon and the purpose to activate the neuron. (Figure 2.1)

A biological neuron first receives inputs in the dendrites, makes some computation in the soma and the result is sent to the axon hillock. The latter decides based on a threshold if the cell will activate or not. The state will finally be propagated through its axon and communicated to the other neurons using synapses.

Mathematically, this procedure has its representation in the perceptron, also called artificial neuron or unit. It receives some inputs x_i (the dendrites) associated with a weight w_i (the synapses), which allows some units to have more importance than others, and an extra constant b called bias. The activation function, aims to model the axon hillock, deciding whether the unit will fire or not. This function applies an element-wise transformation to the data [16]. The perceptron equation can be formalized as:

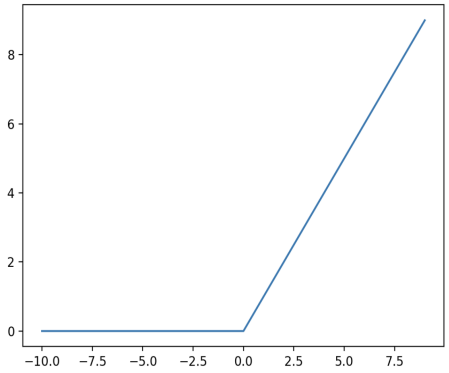
$$y = h\Big(\sum_{i=1}^{n} x_i w_i + b\Big) \tag{2.1}$$

Where h is this non-linear function applied to the output of the neuron.

The literature is plenty of different proposed activation functions with different purposes and performance. Nevertheless, all these different choices share a common purpose which is to adding a non-linearity to the model, leading the network to be able to learn and represent non-linear relationships, present in most of real world data.

ReLU, i.e. **rectified linear unit**, it is usually the recommended choice for most

of neural network. It is defined by the following function:



$$h(x) = max\{0, x\} \tag{2.2}$$

Despite actually adding a non-linear transformation to the data, the rectified linear unit still remains very close to a linear function, being composed of two linear pieces. This helps preserving the useful properties like easy-optimization and good generalization typical of standard linear models [16].
The Perceptron proposed by Rosenblatt et al. [15], is the father of artificial neural networks thought to perform a binary classification task. Hence, it analyses the input data and classifies it in a class whether than the other. In general, an artificial neural network is characterized by three different types of layers:

- *Input Layer*: is the first layer of a network. It takes the input data and pass it to the following layer.

- *Hidden Layer*: is the layer that performs the non-linear transformation to the input data, as said in 2.1.

- *Output Layer*: is the final layer of the network, the output predictions are based on the values of its neurons. The way we obtain our predictions may change based on the task the network has to perform.

If more than one hidden layer is present in the network, we talk about **Deep neural networks** (**DNNs**). When the input data flows through the hidden layer and finally to the output layer, the network is called **feedforward**, hence we call it Deep feedforward network or **Multi-layer Perceptron** (MLP). If we extend the hidden layers to include feedback connection, we talk instead of **Recurrent Neural Networks** (RNN). As a common practice, feedforward networks organize the layers as a stack. The number of stacked layer gives us the depth of the model. In general, we speak of *deep learning* [16] when the number of hidden layers is greater than one.

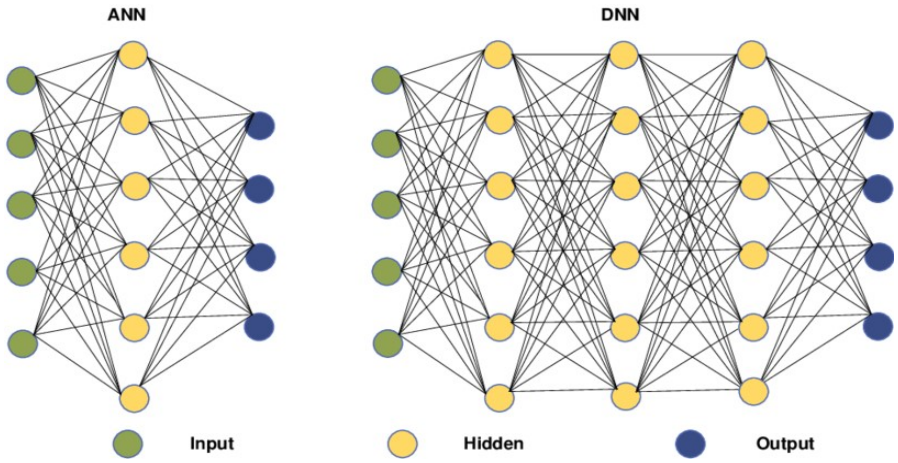


Figure 2.2: Artificial Neural Network architecture vs Deep Neural Network [17].

2.1.2 Backprogation and stochastic gradient descent

How machine learning/deep learning models *learn* is also inspired by the way humans learn. We, as humans, are instinctively drawn to look for a pattern to memorize and learn concepts. The same thing is done for letting machine learns. Given the input data, the model tries to find a pattern, and use a function to understand whether the result is correct or incorrect.

This function comes by different names: objective function, criterion, loss function or error function[16], denoted with $\mathbf{J}(\theta)$. Intuitively, what we want to do is to reduce as much as possible the incorrect answers, so mathematically we want to either minimize or maximize this function. To do so, in machine learning algorithms, it is used the optimization technique called **gradient descent**, shown in Figure 2.3. Basically, the algorithm computes the derivative to reduce the loss function, performing small steps towards the negative direction. Since the networks have many inputs, we speak of gradient instead of single derivatives. The **gradient** of f is the vector containing all the partial derivatives of the input vector, $\nabla_x f(x)$. These are *directional derivates* in the direction $\mathbf{u}$. In this way, to minimize the loss function, we need to find the direction in which f decreases the fastest. The optimization function is reduced to $min_{\mathbf{u}} cos\theta$, where θ is the angle between $\mathbf{u}$ and the gradient. The optimization problem results to be optimized when $\mathbf{u}$ points to the direction of the negative gradient. For deep learning models, this optimization algorithm is extended in order to deal with a much larger amount of data, typical of deep learning, which allows to generalize better the pattern but is definitely more computationally expensive. **Stochastic Gradient Descent** (SGD) takes into consideration a small group of data size *m'*, called **minibatch**, sampled uniformly from the training set. In this way, the model can be fitted using updates computed

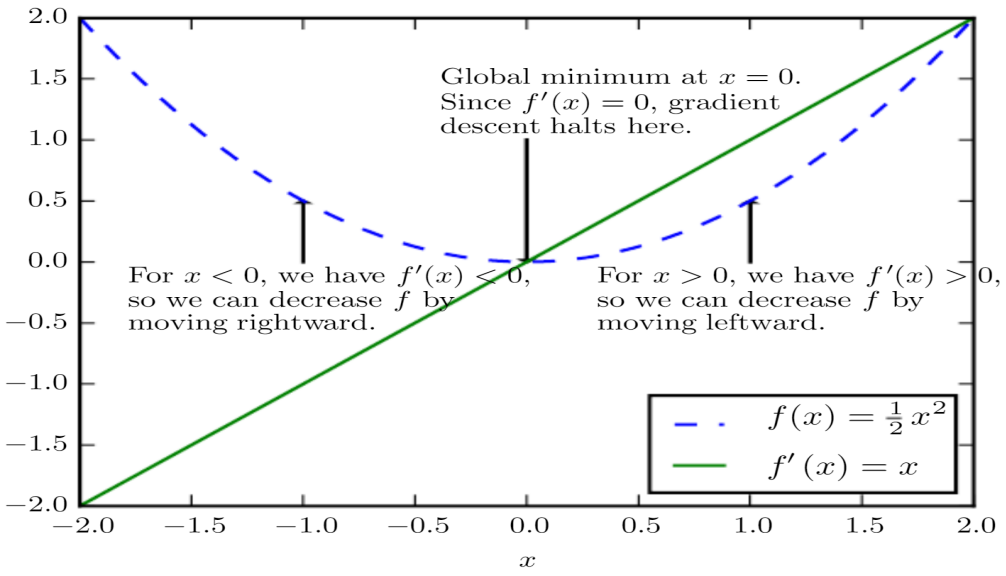


Figure 2.3: Gradient descent. The figure shows how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum [16].

on a small number of examples.

$$g = \frac{1}{m'}\nabla_\theta \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \theta^{(i)}) \tag{2.3}$$

Where $\mathbf{x}$ is the input, y is the output.
Then, the SGD optimization algorithm update at each iteration the parameter following the estimated gradient downhill (i.e., in the negative direction):

$$\theta \longleftarrow \theta - \epsilon\hat{g} \tag{2.4}$$

Where ϵ is the *learning rate*, a positive scalar hyper-parameter of the optimization algorithm which is multiplied by the step and determines the size of the step.
The main problem of stochastic gradient descent is that when working with the non-linearity present in the activation function of the deep neural network, the loss function becomes *non-convex*. Hence, the algorithm is not guaranteed to converge. Nevertheless, most of the time the algorithm found very low minima, that still result in a very useful result.
The **back-propagation** algorithm is often confused as the learning algorithm itself, but as a matter of fact, it is only the way to compute the gradients. It basically performs the **chain rules** of calculus with a specific order of operation.

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z \tag{2.5}$$

Where $\nabla_x z$ is the gradient of variable x, $\frac{\partial y}{\partial x}$ is the Jacobian matrix that is multiplied by the gradient $\nabla_y z$.
Back-propagation usually applies the chain rule to tensors of arbitrary size. The difference of a tensor with a vector is that the numbers are arranged in a grid-form. When applying the algorithm to tensors we can imagine to flatten each tensor into a vector-form before running back-propagation that finds the gradient, then the latter is reshaped back into a tensor-form.
To sum up, during training the model perform a forward pass to calculate the output of the network and get the loss function. Then, uses back-propagation to calculate the gradients with respect to the loss, and finally uses stochastic gradient descent (which use the gradients just calculated) to update the parameters.

2.2 Convolutional Neural Networks

Convolutional Neural Networks [18], (CNNs) are a specific type of neural network thought to deal with grid-like data, such as images or time series. To deal with them, fully connected layers would require several tens thousands of weights and the memory requirements may not meet the hardware implementations. To tackle this problem, LeCuun et al. identify three main key ideas Convolutional Nets are focused on:

- **local receptive fields**, useful to extract visual features such as edges, endpoints or corners. Combining these features in the following layers the network manage to create higher order features. Moreover, a local feature useful to one part of the image, is likely to work well over the entire image.

- **shared weights** (or weights replications), in fact, units in the same layer are organised in planes so that they share weights. The output of such layers is called a *feature map*. This units are constrained to perform the same operation over different part of the image, so that each feature can be extracted at all possible location on the input. The units of a feature map are connected to an $x \times x$ area, called receptive field of the unit, this is centered on corresponding units in the previous layer, making successive units overlap.

- **spatial or temporal subsampling**, to reduce the resolution of the feature map and hence reduce the sensitivity to shifts and distortions. In fact, one the feature has been discovered, its exact location is not highly important, but only its approximate location is considered relevant. For example, we can say the input data contains a number 7 on the upper left corner if it contains a roughly a horizontal segment and a vertical segment, connected at the top-right endpoints. Additionally, knowing the exact position of the feature

could be potentially harmful, since the number 7 can change its conformation based on handwriting style.

This kind of networks take their name from the particular linear operation they perform in at least one layer, called the *convolution*.

2.2.1 Convolutional layers

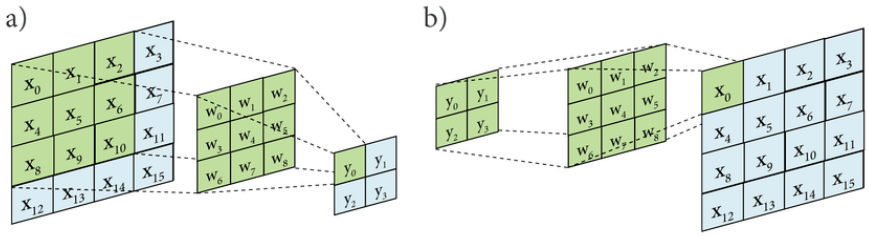


Figure 2.4: Example of convolution operation (a) and equivalent transposed convolution operation (b) for a 3 × 3 filter kernel size applied to a 4 × 4 feature map [19]

Convolutional layers represent the most important element of CNNs. Their main purpose is implementing a convolution operation. Generally speaking, convolution is an operator acting between two real functions.

$$s(t) = (x * w)(t) \tag{2.6}$$

The first term is the **input**, while the second is referred to as the **kernel** or **filter**. The **feature map** or **activation map** will be the output of this operation. The convolution specifically performs a dot product between the parameters of the kernel and the input, therefore it converts all the values in its receptive fields into a single value, in this way reducing the size of the input. Figure **??** visually present the convolution operation over 2D matrices.
In a single convolutional layer, there can be many filters, each filter has height and width smaller than those of the input and it is convolved with the whole input, producing a multidimensional feature map. A complete convolutional layer stack all the feature maps of each filter, so that multiple features can be extracted at each location.
Convolutions are often computed over more than one axis at a time. Therefore, taking for instance a two-dimensional image as input, the formula of the convolution becomes:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.7}$$

Where I is the input image and K is the kernel.
The key idea behind convolutional layers are *sparse interaction* and *shared weights*.

- **sparse interaction** or sparse connectivity or local connectivity refers to the fact that CNNs do not use matrix multiplication between each input and each output, but making use of the kernel, usually smaller than the input, they limit each neuron to interact with a small portion of its nearby neurons. An visual representation of the a sparse layer is shown in Figure 2.5. In this way, the parameters stored are way less, giving an important advantage in memory storage requirements. Moreover, computing the output values requires much less operation, resulting also in a computational improvement.

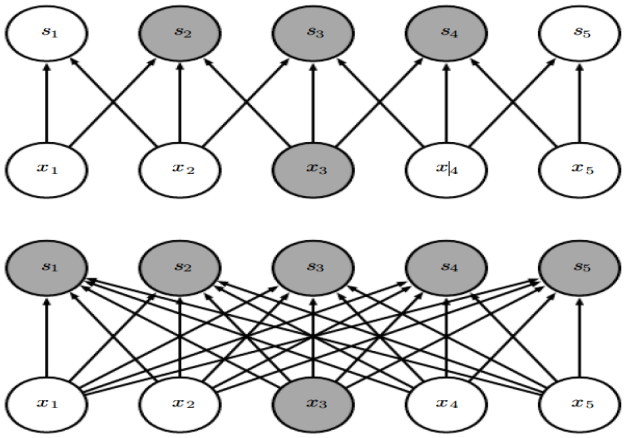


Figure 2.5: A sparse layer vs a fully connected layer [16]

  For each neuron s in layer m, the following neurons in layer $m+1$ that are affected by the output of s constituting the **receptive field** of neuron s. Usually, the receptive fields in deeper layers in the convolutional networks are larger than the receptive fields in the shallow layers of the same network. Thus, the neurons of the deeper layers are still, even if *indirectly*, connected to all or most of the input units. In this way, the network is able to describe local features, for example, edges or corners, and combine them in the following layers to create higher order features, allowing convolutional networks to describe complicated interactions.

- **shared weights** indicates that the same parameters are used for more than one function in the network. Differently from MLPs where each weight is used only once to compute the output, in convolutional networks the values of the kernel are used all over the input dimension. The units of the same feature map are forced to share the same set of weights [18]. In this way, the

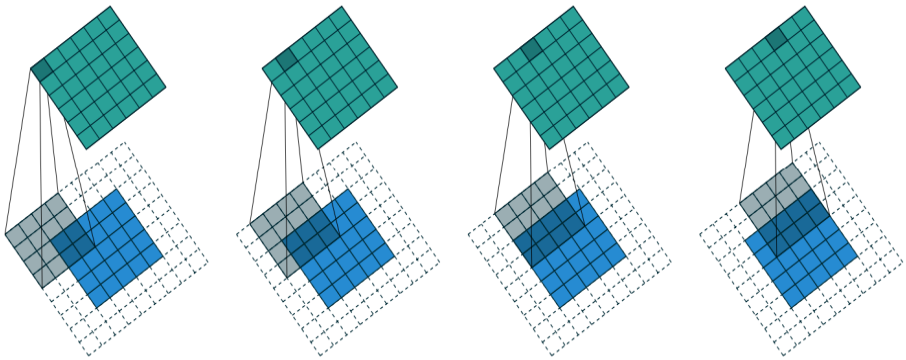


Figure 2.6: Padding example with a 2 × 2 border of zeros using unit strides [21]

network is able to detect the feature regardless of its position in the input. This procedure additionally reduces the number of parameters that needs to be stored.

The size of the output of the convolutional layer is controlled by three specific hyperparameters, which are set before the training of the neural networks. These are the *stride*, *depth*, and the *padding*. The stride indicates the number of pixels/values the filter moves over the input. The depth is the number of filters used for each layer, this affects the final dimension of the features map. For example, if three filters are used, the output will be a feature map of depth three. Finally, sometimes it is useful to pad the input to allow control over the final size of the output and to avoid loosing information during the convolutional operation, especially in the perimeter of the input. Usually the input is padded with zeroes, as shown in Figure 2.6, where the output dimension depends on the following formula [20]:

$$\mathcal{O} = 1 + \frac{N + 2P - F}{S} \tag{2.8}$$

Where P is the number of pixel to be zero-padded, N is the input size, F is the filter size and S is the stride size.

2.2.2 Pooling layers

Typically in convolutional networks, the main building blocks are composed of a convolutional layer, a non-linear activation function such as the Rectified Linear Unit, and a pooling function that modifies the previous output to pass it to the following layers. Pooling divides its input data into small non-overlapping rectangles

and for each of them, it outputs a statistical summary of values. In fact, it is also known as the downsampling layer, since it reduces the dimensionality of the input. The most common function used are **max-pooling** [22], which takes the maximum values in the subset, and **average-pooling**, which computes the arithmetic average of values in the subset. The pooling operation introduces *invariance* in the network, i.e., it makes the network robust to small translations in the input data. In this way, the network takes more into consideration whether a feature is present or not, rather than considering its exact location.
It is possible to increase the *stride* between the pooling regions to k pixels (in the case of images) instead of 1. Note that increasing the value of k reduces the computational complexity of the network, since the number of input pixels to be processed is reduced approximately by a factor k^2 [16].

2.3 Microcontrollers and Embedded systems

Microcontroller units (MCUs) represent *complete computing systems* delivered on a single board and usually workirng without any operating system. The main components of an MCUs are: a *central processing unit (CPU)*, a Read Only Memory (ROM) that holds the software and which is non-volatile, meaning that it retains the its content even without power; a Random Access Memory (RAM) i.e., a volatile memory that stores data and code generated for execution. To interact with sensors and other devices MCUs are provided also with a vast range of peripherals like General purpose input/output pins (GPIOs), timers, AC/DC converter, etc. In this way, MCUs do not require any additional components to interact with sensors and different components, making them popular in several applications such as Internet of Things (IoT) [23], healthcare [24], and drones [25].
It is easy, due to the similar nomenclature, to confuse Microcontrollers and Microprocessor Units (MPUs). Normally, the first is associated with *embedded systems*, while the second is commonly used to denote the computing core of personal computers [26]. On the other hand, the difference also depends on the performance, MCUs are less powerful than Microprocessors. Indeed, the computational power of MPUs is generally much higher. For example, [27] compares two devices in terms of processing power, measured in Dhrystone MIPS (DMIPS): a ARM Cortex-M4-based microcontroller such as Atmel's SAM4 MCU is rated at 150 DMIPS while an ARM Cortex-A5 application processor (MPU) such as Atmel's SAMA5D3 can deliver up to 850 DMIPS.
The market is plenty of different types of microcontroller with very different capabilities. In general, the following thumb rule holds: the larger is the data bus width, the more complex the Instruction Set Architecture (ISA) of the MCU [28] would be. Usually the bus width ranges from 4-bit to 64-bit. ARM is the dominant

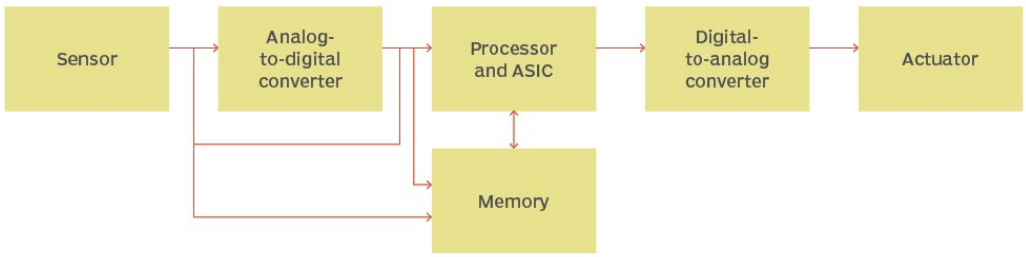


Figure 2.7: Typical structure of an embedded system [30]

family of processors designed by the company Arm Limited. They are based on a Reduced Instruction Set Computer (RISC) architecture, which considers each instruction as a singular function. On the contrary, in a Complex Instruction Set Computer (CISC) a single instruction can perform multiple low-level functions (this is the architecture used by processors for desktop computers like Intel). The ARM ISA is used for the vast majority of embedded devices like smartphones and wearables. Nevertheless, recently RISC-V, a new architecture, is gaining popularity. RISC-V proposes a new ISA where the basic instruction-set is based upon fixed 32-bit instruction and must be aligned on a 32-bit boundary, but its encoding method can support the extension of the instruction set using variable length instructions. This kind of architecture is very suitable to met IoT devices constraints having low power consumption, low cost and good scalability [29]. As said, MCU's main use is in embedded systems. But, what is really an **Embedded system**? In Figure 2.7 we can see an example of a typical structure of an embedded system. Generally speaking, it is a combination of hardware and software that aims to model a specific function and is usually part of broader systems, hence the name *embedded*. In particular, as pointed out by F. Vadid et al. [31], embedded systems share three main common characteristics:

- *Single function*: MCUs perform a function repeatedly which can vary from domain to domain: calculator, microwave oven, printers, and so on.
- *Tight constraint*: the cost of the embedded system should be very low and this lead to very tight constraints in terms of power, memory, performance or battery.
- *Reactive and/or real-time*: many applications require real-time or nearly-real-time reaction from the embedded system, to calculate metrics like accelerations or to react quickly to changes in the systems.

Despite originally embedded systems was quite simple, nowadays they are becoming

more and more complex allowing the designer to implement complex functions and decision-making programs.

2.3.1 Embedded Neural Networks constraints

In recent years, there was a big trend to port neural networks into embedded systems, also called edge devices, like wearables, smartphones and the IoT nodes [32]. When performing training and inference, neural networks require a great amount of computational power, hence they are usually executed on cloud servers or GPUs. Therefore, a strong trend is to move the training and especially the inference of networks on edge devices to provide benefits in terms of latency, scalability, energy consumption, and privacy. Doing so is not easy, the majority of current devices are not able to run the inference phase of deep natural networks in a real-world scenario. Hence, there is a need to optimize the memory occupation, computational complexity and energy consumption. Since embedded systems often deal with real-time (or nearly real-time) applications a significant problem is the overall processing latency. Often the main latency bottleneck is linked to the transfer of data from the memory to the CPU and vice versa. The **roofline model**[33] is often used to visualize the bound of networks in terms of **operational (or arithmetic) intensity** (**OI**):

$$OI = \frac{N.ofOperations}{BytesTransferred} \cdot \left[\frac{FLOP}{Byte}\right] \tag{2.9}$$

This metric is typical for each implementation of a given task, hence different implementations can have different OIs. The model helps to understand the upper bound considering hardware characteristics. In Figure 2.8, we identify two main

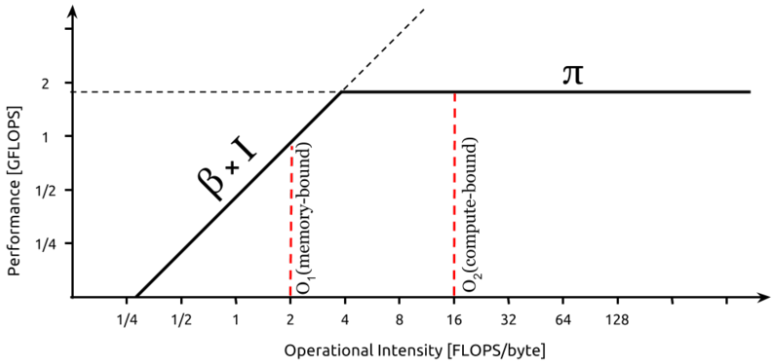


Figure 2.8: The roofline model. FLOP stands for Floating Point Operations.

elements:

$$\pi = peakperformance[FLOPs/OrGigaFLOPperSecond] \tag{2.10}$$
$$\beta = peakbandwith[Bytes/s] \tag{2.11}$$

Looking at the graph in Figure 2.8, we can differentiate two behaviours:

1. **Memory bound**: in the leftmost side of the model the performance are limited by the memory, hence the maximum speed achievable depends on the bandwidth of the platform and not on the computational capabilities.

2. **Compute bound**: in the rightmost side of the model, instead, the performance are limited by the computation, hence we do have enough bandwidth to transfer the data, but the Arithmetic Linear Units (ALUs) will be always active.

As indicated by [32], the energy consumed by the operations and data fetches are the real bottleneck of neural network implementations on embedded systems. This problem is still open,requiring both hardware and algorithm optimizations.

2.4 Optimization for the Embedded Deployment of DNNs

Between all the aforementioned good features of Deep learning models, we concentrate now on the well-known approximation resiliency to different kinds of approximations. In fact, changing in small ways the input data or the computations does not lead to significant changes in the final result. We can exploit this behaviour to make the models faster, smaller and more efficient. Here, we focus on two main categories of approaches, that aim to yield benefits for general purpose hardware like MCUs.

2.4.1 Quantization

Much work has been done to reduce the size of networks. The problem with standard models is the extra memory and computational power required for floating-point scaling operation [34]. One of the common solutions is to compress the network and to reduce the many MAC operations present in the training and inference of deep networks. In this context, quantization has proven to work well for both training and inference steps. It consists in reducing the precision of the computations, usually performed at 32-bit float. If done properly, reducing the bit-width of operand we can gain a lot in terms of performance while loosing a very little in

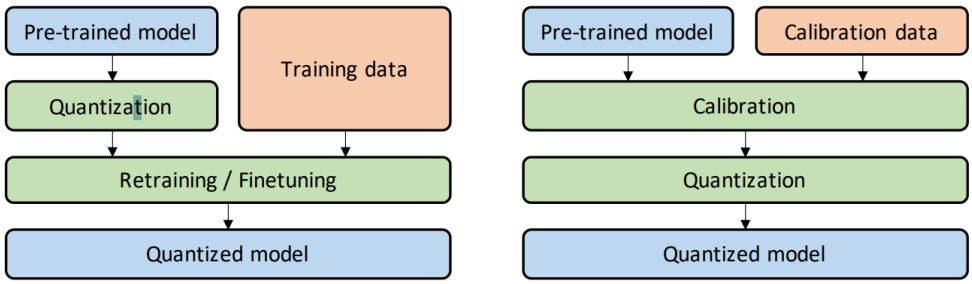


Figure 2.9: Post-Training quantization vs.Quantization-aware training [34]

accuracy. Hence, switching from 32-bit to 8-bit data can theoretically reduce the time and the energy to transfer data from and to the memory of a factor of 4. The main benefits of quantization are the higher power efficiency and the memory storage reduction which does not require to change the network topology. Nevertheless, extreme reduction in precision may lead to some or severe information loss, resulting in significant drops in the accuracy values. This is referred to as *quantization error*. Quantization can be applied to both weights and activations of the network and can be done either after training or also during it. The first method is known as *post-training quantization*, while latter is referred to as *quantization-aware training*. We now describe the two methods in more details.

Post-training quantization

The model is normally trained in floating-point, then it is compressed to a lower precision like *int8*, which is usually considered the limit for Post-Training Quantization (PTQ) [34]. This is the simplest and fastest way to quantize a model and can be used to quantize pre-trained models with limited data available. The downside is that quantization may cause a significant drop in the accuracy of complex tasks. As said, we can quantize both weights and activations. In details:

- **Weights**: only the weights of the networks are converted to integers while the input, the output and the activations remain stored in float32 format. Quantization can be either *uniform*, which uses integers of fixed-point, and *non-uniform*, which instead requires a code book, a sort of look-up table to be used to dequantize the values before doing higher precision computations [35]. At the core of this last method, there is a format called *dynamic fixed-point*, which tries to embed the different distribution of the parameters in the layers of the network. The term *dynamic* refers to the fact that the set of quantization parameters are kept static with the same tensor (or sometime within the same channel of the tensor), but changes concerning the other (or

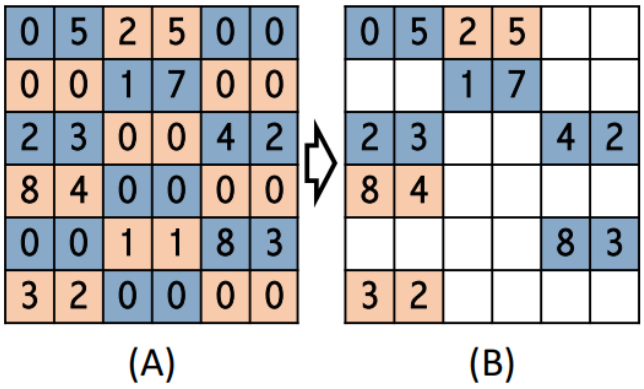


Figure 2.10: (A)Weights grouping; (B) Sparse weight matrix after pruning [36].

channels in the same tensor).Mao et al.[35] identify the quantization process as a two-step process: first, the range of numbers to be quantized is chosen, then the real values are mapped into integers represented by the b-width of the chosen quantization. A common function to represent the real values into integer is given by $f(x) = s \cdot x + z$. This is called *affine quantizer*, where s is a scale factor and z is a zero offset, identifying the integer in which the real value zero is quantized. The latter are defined as:

$$s = \frac{2^b - 1}{\alpha - \beta} \tag{2.12}$$

$$z = -round(\beta \cdot s) - 2^{b-1} \tag{2.13}$$

The final quantization is defined by:

$$clip(x, l, s, u) = \begin{cases} l, & x < l \\ x, & l \geq x \leq u \\ u, & x > u \end{cases} \tag{2.14}$$

$$x = clip(round(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1) \tag{2.15}$$

- **Weights and activations**: in this case, since the quantization is performed post-training there is the need for some *representative* data selected from the ones used during training to give to the converter to properly perform quantization. Thus the scaling factor s and the offset z can be calculated as a statistic of the provided input images. Usually, a small range of data (100-1000) is sufficient.

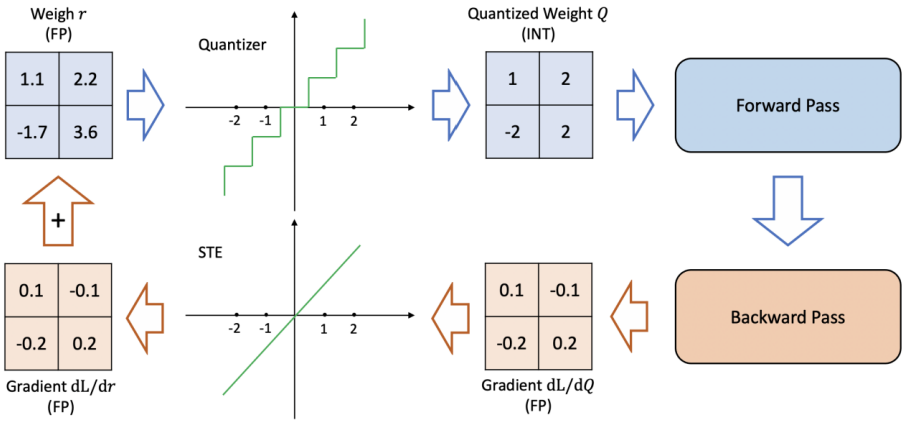


Figure 2.11: Quantization-aware training with Straight-Through Estimator [34]

Quantization-aware training

Quantizing the parameters of the network when the training is already completed introduces noise in the network's parameters. This effect of this interference occurs during inference when the network operates in a point which can be relatively far from the point reached with the same model with floating-point precision [37]. To mitigate these problems the *quantization-aware training* (QAT) technique has been introduced, which nowadays is the de-facto standard procedure for quantization of complex tasks, since they suffer a more severe accuracy drops when post-training quantization is used.

Quantization-aware training compresses the parameters of the network at each step of the gradient so that in the end a better loss value is reached. But it is important to still perform the backward pass in floating-point precision, since the gradient of quantized values is almost always zero in all differentiable points. In fact, as it shown in Figure 2.11, the function is a thresholding function, so it is constant in intervals. This means that its derivative is zero almost everywhere making the network not learning at all. A common way to treat this problem is to use the **Straight-Through Estimator** (STE) technique. Gholami et al. [37] explain clearly how STE simply ignores rounding operations and uses an identity function to approximate the values. Therefore, the training is still performed in float, but some nodes have *fake* quantization operations since the forward pass is performed as if the operations are done with integer data.

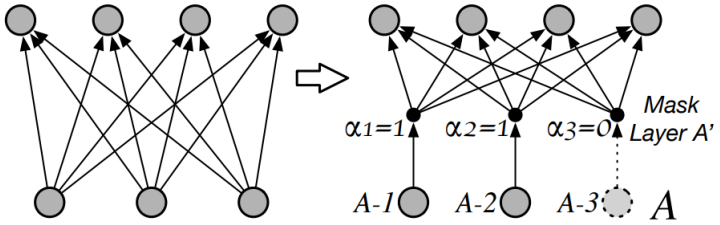


Figure 2.12: Node pruning example with masked layer. Node $a-3$ with$\alpha_3 = 0$ can be removed. [36]

2.4.2 Pruning

Quantization techniques allow to reduce the size and the complexity of the network by reducing respectively the precision of the weights to be stored and the precision of MAC operations, but the computational complexity depends also on the overall architecture of networks. Pruning is an optimization technique that first trains the network regularly and then prunes the unnecessary or redundant connections. This leads to getting good level results with fewer parameters (and also fewer MAC operations) and better generalization [38], thanks to the well-known regularization effect offered by the utilization of smaller networks. Pruning is based on the innate *sparsity* of neural networks. Sparsity, in fact, helps in improving the generalization ability and in improving, in general, the performance at inference time and/or training time. Recent works have proven that the number of pruned parameters of the models can reach up to 90% with negligible loss in performance [39].
The first categorization of pruning techniques can be done on the object of pruning, which can be the weights or the activations. The first allows having a good reduction in the model size, changing the magnitude of the weights, and completely eliminating the weights corresponding to zero values. It is very beneficial in terms of memory occupancy but it is done during training and requires re-training the model. The second is to zeroing some parts of the activations that have a small magnitude. On the contrary of the weights, this is done at inference time.
The second categorization of pruning techniques is done between **unstructured pruning** and **structured pruning**. The first does not follow a specific constraint, and whatever element of the network can be pruned. On the contrary, the second one defines specific locations that can be pruned. We now present two examples of structured pruning at different granularity:

- **Block-based pruning**: shown in Figure 2.10. It consists of grouping the weights into aligned groups that follow the underlying hardware parallelism. After that, the weight groups are pruned based on the value of an aggregated

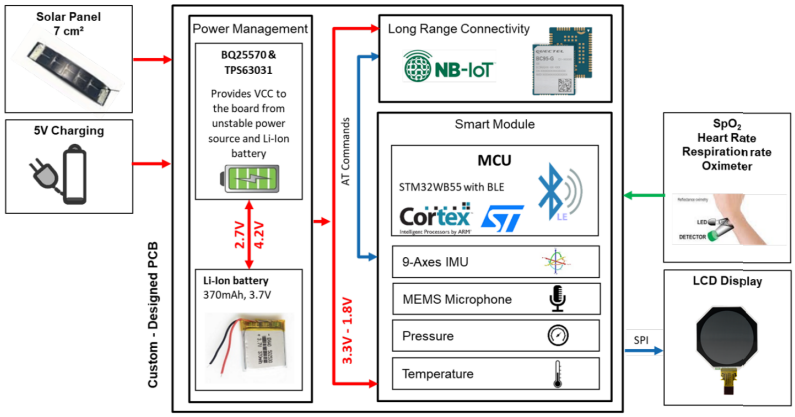


Figure 2.13: H-Watch logic schematic [41]

metric, the common choice is to use Root-Mean-Square (RMS) to measure the importance of the group and if it is below a threshold the entire weight group is removed. Finally, the whole pruned weight matrix will be retrained [36]. This kind of pruning works best with *Single Instruction, Multiple Data* based (SIMD-based) processors and can reduce the model and execution time of deep natural networks, since it can load a weight group with a single load instruction.

- **Channel pruning**: shown in Figure 2.12. It removes entire nodes instead of single weights. We refer to each neuron in a fully connected layer and to each feature map in a convolutional layer as a node [36], [40]. Inserting too much sparsity in the weight matrix may hurt the computation performance of the layer. Hence, node pruning exploits a *dynamic mask layer* to discover less significant nodes dynamically, and multiply the input with a Boolean value. This does not actually increase the sparsity of the layer, it simply shrinks it. The final step of node-pruning is to remove the masked layer and re-train the network. The advantage is that the pruning procedure can come up with the optimal layer size automatically.

2.5 H-Watch

The system presented in this elaborate is specifically based on the work done by Polonelli et al. in [41]. The authors propose the design and implementation of the Health Watch, or H-Watch, a hardware-firmware open-source smartwatch which combines different sensors for health monitoring. In Figure 2.13 it is repre-

sented the H-Watch logical architecture. The smartwatch-form device consists of a Li-Ion 370mAh battery and a solar panel of 7 cm^2, the SoC (Sistem on Chip) STM32WB55RGV6 by ST Microelectronics, and several sensors. Among these, a 6-axes Inertial Measurement Unit (IMU) featuring 3D digital accelerometer and 3D digital gyroscope (LSM6DSM), an integrated low-power SoC for pulse oximetry and heart rate monitoring (MAX30101). Moreover a LCD display as user interface is present. As communication protocols the H-Watch provides an NB-IoT transceiver (NB stands for NarrowBand and is a wireless communication standard specifically made for IoT) and BLE interface.
H-Watch comes with four operation modes. The *Sleep mode* is the state with the least power consumption, requiring only 97 μW of consumed power. It considers all the sensors turned off except for the real-time clock and the display. The *Advertising mode* of the BLE, consuming 226μW. The *motion detection mode* consumes 1.75mW since it uses accelerometer and skin temperature sensors. Finally, the *full operation mode* keeps active all of the possible health classification tasks, increasing the power consumption up to to 10mW.

Chapter 3

Related Works

Computer vision, Natural Language Processing, and Healthcare are just some examples of applications where Deep learning has made plenty of improvements. Among the new improvements introduced together with the deep learning, there are the wearable devices for healthcare monitoring, which are the main focus of this work. Wearable devices are able to track data from different sensors in real-time. Thanks to this, they can monitor the user's health, track fitness progresses or encourages proactive behaviour for a healthy lifestyle.
Heart Rate monitoring is very popular in this field since it is useful to either track fitness-related information, and monitor heart health. HR is a vital sign able to give straight away informations about a person's health. For instance, the experts indicate that a normal HR in adults resting is in the range between 60 and 90 bpm [42]. To estimate the HR, usually the distance between two consecutive heartbeats is computed. These are captured from the waveform of different kinds of biological signals (e.g., from electrocardiogram or photopletismographic signal), but noise can mask the real heart rate signal. Hence, the quality of the HR is dependent on the quality of the waveform [43], which is in turn a consequence of the amount of noise present in the signal.
Electrocardiogram was the early predominant signal employed to get the heart rate. It uses electrodes positioned on particular parts of the body (chest, wrists, arms and legs) to record impulses and calculate the heartbeats. It is a fast, simple and effective way, but at the same time it is expensive and requires contact with the skin of the person, making it non-portable and uncomfortable (wearing an ECG chest band would impair the subject daily activities).
To overcome this problem, in recent years, the use of a PhotoPlethysmoGraphic signal to compute the heart rate of the user wearing a smartwatch is becoming progressively more employed. Compared to ECG, the collection of the PPG is not invasive and more comfortable. In particular, PPG uses LED lights and a photo-detector as a receiver to detect volumetric changes in blood streams [45].

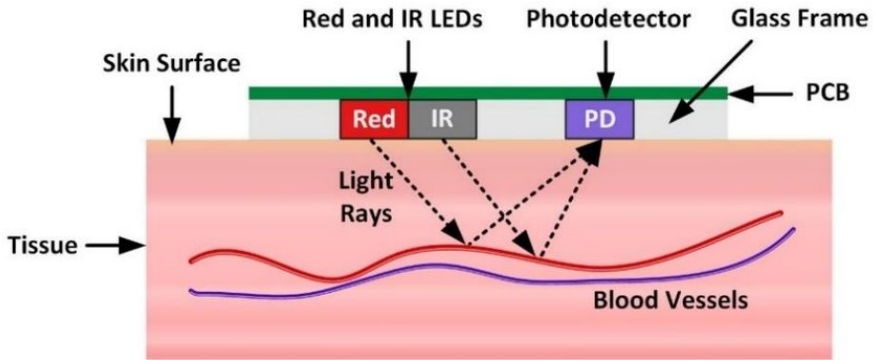


Figure 3.1: The principle behind a reflection-type PPG sensor. The pulse signal obtained from a PPG sensor comprises an AC (pulsatile) and a DC (slowly varying) component. The AC component is attributed to changes in the blood volume synchronous with each heartbeat, whereas the DC component is related to respiration, tissues, and average blood volume. The two commonly LED are red and infrared (IR), which gives different absorption properties of the blood stream. The photodetector captures light and it is used to estimate blood volume changes. [44]

Figure 3.1 shows the logic behind the functioning of a PPG sensor. The period of the light impulses indicates the HR. On the other hand, the presence of different motion artifacts, i.e. distortions in the signal, in most waveform signals captured by wrist-worn devices can affect the fidelity of the signal [46]. Nowadays, the state-of-the-art procedure to deal with these noises is to apply sensors fusion techniques (Deep Learning being one of them) of the PPG signal and accelerometer data.
The algorithms that deal with this type of data can be divided into two main categories. On one hand, there are the *classical approaches*, which are based on time or frequency related features and on filtering and peak detection functions. On the other hand, the second category is based on *deep learning approaches*, which is still a relatively new field the researchers are increasingly exploring.

3.1 Classical Approaches

A first example is represented by **TROIKA**[47], which is a framework composed of three parts from which it takes the name: decomposiTion, sparse signal ReOnstructIon and spectral peaK trAcking. The first module partially removes the MA from the PPG data and scatters its spectrum in the range $[0.4 - 5]Hz$. The second module makes TROIKA robust to noise interference calculating by means of calculating a high-resolution spectrum of the PPG signal. Finally, the spectral peak tracking module is the part of the framework that aims to find the peaks

corresponding to HR values. The authors collected data samples from 12 subjects and showed that TROIKA achieved significant better results with a standard deviation error of 3.07 BPM compared to previous works.

Zhilin Zhang explored in [48] a deep neural architecture to specifically address PPG signals heavily affected by MA. In fact, previous techniques like Independent Component Analysis (ICA)[49] or Kalman filtering [50] fit mainly weak MA scenarios. Hence, the author proposed a new approach based on JOint Sparse Spectrum reconstruction, or **JOSS**. The underlying idea is that PPG and acceleration signals may have common characteristics of the spectrum structure. Hence a model called multiple measurement vector (MMV) is used in joint spectrum estimation (a similar, but less powerful model applied to a single spectrum was presented also in TROIKA). The MMV model identifies the spectral peaks related to MA in the PPG spectra using the spectral peaks extracted from the acceleration spectra. The author bases the method on the common sparsity constraint, which *"encourages the frequency locations of MA in the PPG spectra to be aligned well with some frequency locations in acceleration spectra".*

The author evaluated the algorithm on 12 different datasets, each of them containing PPG signal, accelerometer data, and a channel of ECG data. As a pre-processing step the raw data was filtered with a bandpass filter form 0.4 to $4\,HZ$. In the paper, an explicit comparison with the TROIKA results on the same datasets were presented to show the improvements in the MAE values compared to it. In particular, JOSS reached a MAE $1.28\,BPM$ in contrast to the $2.42\,BPM$ of TROIKA over all the datasets.

Another example of a classical approach is represented by [51]. In this case, to address the MA issue the authors exploited a Singular Value Decomposition (SVD) as MA cancellation step to get periodic MA components. A successive adaptive filtering step suppresses the these components in order to get a two-channel PPG clean signal. These two results are sent in input to a Spectral Analysis step, consisting of yet another two substeps. First, the Iterative Method with Adaptive Thresholding (**IMAT**) is used to get a higher resolution and to denoise the spectrum of the input signal. Then, a Peak Selection step that apply a decision mechanisms based on the frequency harmonic of HR and considering that the HR does not have abrupt jumps between two successive windows. In this way, the proposed algorithm achieves a MAE of $1.25\,BPM$, obtaining a slight improve in comparison with previous works.

A successive study [52] focused on the reconstruction of MA-corrupted PPG signals proposing a new algorithm called Spectral filter algorithm for Motion Artifacts and heart rate reconstruction (**SpaMA**), which is divided in five steps. First there is the Time-Varying Spectral analysis that perform a downsampling step on the PPG and accelerometer data and compute the power spectral density. Then, the Spectral Filtering step considers only the largest frequency peak of the

accelerometers' spectra to feed to the successive step, i.e. the Motion Artifact Detection that compares the frequencies between the PPG and the accelerometer spectra. In particular, if the first or second largest peaks in the PPG spectrum are very similar to the accelerometer's one, than the MA is present in the PPG, and the corresponding peak is discarded. After that, the Heart Rate Tracking and Extraction step is used to identify the HR frequencies. The final step is the PPG Signal Reconstruction for the hear rate variability analysis. This procedure outperforms previous algorithm obtaining a MAE of 0.89 BPM on the same dataset (the SPC 12 subjects dataset).

Chung et al. propose in [53] a finite state machine (**FSM**) framework that leverages the crest factor (i.e., the prominence of the peak) from the periodogram obtained after a MA removal step. The framework is thought to address especially the estimation of HR during high intensity physical exercises. In particular, the FSM activates after the calculation of the HR and the crest factor, changing states based in their values. There are four states: the *stable state* used when the HR has a good level of confidence to be accurate, the *recovery state* indicate that the HR value is somewhat likely to be accurate, the *alert state* denotes that the HR is well be inaccurate. Finally, the *uncertain state* implies that HR is probably wrong. The framework manages to reach an average MAE 0.99 BPM.

More recently, [54] proposed a new algorithm, called CUrve Tracing On Sparse Spectrum (**CurToSS**). The algorithm has been developed considering also the more complex DaLia dataset and it includes a sparse spectrum that detects on the PPG spectra which frequencies are relevant to the HR, extending the work previously done in JOSS. In this way the CurToSS achieves a MAE 2.2 BPM on the SPC dataset, and a MAE 4.6 BPM on the more complex DaLia dataset proposed in [12].

A final mention is to the work presented in [55]. Huang et al. introduce in a new algorithm called Time-domain based method involving Adaptive filtering, Peak detection, Interval tracking and Refinement or simply **TAPIR**. The PPG and the corresponding simultaneous accelerometer data are filtered with an least mean square (LMS) adaptive filter to remove MA, then peak detection is performed suing a MATLAB function called `mspeaks`, in conjunction with a thresholding step based on the *refractory period*. After that, the peak adjustment is performed by tracking the average peak-to-peak interval over time, specifically if the interval width is less than a percentage, the peak is likely to have been wrongly selected and therefore removed. A second interval tracking step but performed on a longer time scale is used to get a preliminary hear rate estimation. The final step consists in enhancing the previous HR estimation with a notch filter. The method lead to a MAE of 2.5 BPM on the SPC dataset used in the previous methods, and achieves a MAE of 4.6 BPM on DaLia.

3.2 Deep Learning-based approaches

The use of Deep learning for PPG analysis is still relatively rare, but an increasing number of works are now considering it thanks to the great generalization capabilities of networks that outperform classical methods.

One of the early proposed deep learning models is DeepPPG[12], a Convolutional Neural Networks that takes as input PPG- and accelerometer-signals, and gives as output the predicted heart rates. The model has been analysed extending the investigation to each hyperparameters to obtain optimal results. For instance, they investigated the optimal number and size of the filters in the convolutional layers, the type of activation functions, the pooling layers' size and so on. Note that the first convolutional layer is thought to unify the PPG and accelerometer channels, while the second is in charge of detecting the interesting segments for the heart rate detection. As activation function the authors decided to use the Exponential linear unit (ELU) [56], and decided to insert a dropout rate of 0.5. Nevertheless, the model is too large to be embedded into devices, counting $8.5M$ parameters and $69.5M$ computations required to perform a single inference. Therefore, the authors proposed a constraint-aware model by reducing the number of channels and other layer-specific parameters and by removing the dropout layer, the resulted model is significantly reduced to $26K$ parameters and only $190K$ operations needed per second. The network manages to reach a MAE of $7.65\,BPM$ on the DaLia dataset while on the smaller datasets like SPC the model did not resulted in an improvements compared with the classical methods. A possible reason the authors give is the insufficient amount of data per activity in the datasets.

Another example of deep learning applied to PPG-based HR estimation is given by **CorNet** [46]. The framework has been created with the aim to outdo the limitation created by deep CNNs architures, namely the vanishing gradient, and RNNs which still do not deal well with long term dependency with long sequential input data. Therefore the architecture of CorNet consists of a two-layer CNN, a two-layer Long Short-Term Memory (LSTM) and a final dense layer. The CNN can be thought as a feature extractor able to discern useful features, yet CNN features do not usually work well with time-series-like inputs, here is where it comes to help the LSTM module of CornetNet. LSTMs have been decisive in capturing long term temporal dependencies and help in finding them also in the cardiac activity recorded by PPG sensors. The outputs of this module is finally fed to the dense layer used as a regression layer and customized for HR detection. The resulted architecture is therefore quite performing but highly computational expensive. In fact, the algorithm achieve an average error of 1.47 ± 3.37 on all the subject of the SPC dataset.

One other approach based on Deep Learning applied to PPG signals and acceleration data for HR detection is represented by [57]. Here, two spectra of PPG in

conjunction with acceleration data are used as inputs. As in previous works, the ground truth are provided by ECG signals. Similarly to the previous network, the network proposed by Chung et al. consists of eight layers: one 2D-convolutional layer and one 1D-convolutional layer, two LSTM layers, one concatenation layer and three fully connected layers with an ending SoftMax. In fact, to reduce evaluation loss the HR values are represented as Gaussian distribution, representing each HR values into 222 frequency bins. This approach has as drawback that the predicted frequencies obtained may not represent exactly the ground truths, keeping in mind that this could also be attributed to the difference between HR representations from PPG values and ECG values. The deep learning model is once again quite large, counting $3M$ parameters, but achieves a good MAE of $1.46\,BPM$ on the SPC test data.
The last explorations in the field of deep learning for heart-rate estimation based on deep learning approaches went into the direction of the embedding of these algorithm on low-power and memory-constrained platforms. In particular, in the following of this chapter, we will briefly analyse two works, [58] Very recently a set of models were optimized and tested with the specific goal of employing them in memory-constrained embedding systems. We now, briefly recap two of these models.

3.2.1 ActPPG

Risso et al.[59] propose a collection of *Temporal Convolutional Networks* to estimate HR from raw PPG signals and acceleration data. TCNs are 1D-CNN with the addition of *causality* and *dilation* parameters in the convolutional layers. The first forces the output of the layer $\mathbf{y_t}$ to depend exclusively on the inputs $\mathbf{x_{\tilde{t}}}$ with $\tilde{t} \leq t$, while the dilation consists in inserting a gap d between the input samples processed by the convolution, increasing the receptive field without the need of inserting new parameters. Hence, a convolutional layer in a TCN is implemented by the following function:

$$y_t^m = \sum_{i=0}^{K-1} \sum_{t=0}^{C_{in}-1} x_{t-di}^l \cdot \mathbf{W}_i^{l,m} \tag{3.1}$$

Where x and y are respectively the input and the feature maps, t is the output time-step and m the output channel. $\mathbf{W}$ is the filter weights, $C_{i,n}$ the number of input channels, d the dilation factor and K the filter size. In the paper, the authors propose the adaptation of a popular TCN called TEMPONet, originally used for gesture recognition. Their TEMPONet takes as input raw data of a PPG sensor and the raw input of the accelerometer sensor on three axes. Moreover, the last classification layer is replaced with a single neuron to perform regression. Finally, the loss used during training is the *LogCosh*. In order to reduce the size and complexity and be able to use the network in embedding systems, the authors

decided to use MorphNet, a Neural Architecture Search (NAS) algorithm that is able to automatically prune the channels of every layer, strongly reducing the occupation of the seed network, the TEMPONet in that case. Furthermore the author applies full-integer post-training quantization to switch the outputs from *float32* to *int8*.
In [60], the work of [59] is extended to create two main contributions: *TimePPG* and *ActPPG*. The first is a collection of TCNs architecture to predict HR taking as input raw PPG values and tri-axial accelerometer data. The different architectures are generated using the NAS algorithm called MorphNet [61] with TEMPONet used as TCN seed. In particular, Morphet takes as input the training dataset, and the original TEMPONet, which gives the starting point for architecture exploration. The optimization of this first TCN produces several optimized models with different trade-offs of HR predictions and model complexity. This set of optimized models is what the authors call *TimePPG*. Finally, a smoothing post-processing step is added to further improve the accuracy values. This consists in applying a threshold P_{th} over the maximum variation of the predicted HR with respect to the averaged HR estimated during the previous steps. Results show that the biggest model called TimePPG-Big reaches a Mean Absolute Error (MAE) of 4.88 bpm and has around 232k parameters, while the smallest model requiring only 5.09k parameter still reach promising results with a MAE of 5.63 bpm.
The second contribution given in [60] is a framework called **ActPPG**. It combines the use of different TimePPG models based on the quality of the PPG waveform, hence on the high presence of Motion Artifacts, exploiting the movement data given by the accelerometer. The framework is composed of two modules:

- Movement detector: a lightweight Random Forest model, composted of 8 trees, is fed with the accelerometer data with the goal to categorized the movement levels in a scale of $[0, N-1]$, where N is set to 2. The accelerometer values do not provide any information related to the estimation of the HR, but they have recently become the state-of-the-art signal to clean the PPG values from MAs caused by the movement of the devices, especially wrist-worn.

- Predictors: the models that actually perform the estimation of the heart rates. The module is fed with the previous window difficulty calculated w_d, the training data, and optionally the accelerometer data. Then, based on w_d a specific predictor is chosen, where the higher w_d the more performing predictor will be chosen. Note that the ordering of the predictors is made offline.

The framework is thought to be dependent on two assumptions. Firstly, as the number of movements increases, the MAs increase too and hence the predictions of the HR become less accurate. Second, the difference between the bigger models and the smaller ones is mostly dictated by the ability of the two to recognize the

high movements of some input examples.
Inspired by the big-little neural networks, *ActPPG* improves both small and big models, by using the more appropriate model based on the situation. Despite the improvements in the MAE values may seem meaningful, the reduction in complexity and size, and therefore energy consumption, is significant.

3.2.2 Q-PPG

Following the proposed methodology in [59], the authors extended the work to further reduce the complexity of the models for PPG based HR estimation in [58]. They propose a three-folded contribution:

- The NAS optimization space is extended to consider the *dilation* parameter for convolutional layers. This helps to further reduce the model complexity with a small drop in accuracy.
- Hardware-friendly *quantization* is added to reduce the model size, thus enriching also the Pareto frontier.
- They deployed the result on a real embedded smartwatch form device with a STM32WB55 MCU from ST Microelectronics.

As in the previous works, the seed network for the NAS is TEMPONet. The final output is a set of quantized TCNs and hence the name of the methodology, *Quantized*-PPG (Q-PPG). In particular, the method implemented is the *linear quantizer*, which takes the floating-point tensor from the range $[\alpha_t, \beta_t]$ into N-bit integer tensor $\hat{t}$ as:

$$\hat{t} = round\Big(\frac{t - \alpha_t}{\epsilon_t}\Big) \tag{3.2}$$

where $\epsilon_t = (\beta_t - \alpha_t)/(2^N - 1)$ is the smallest value the quantizer tensor can assume. In [58], writers apply first *uniform quantization* and the quantization-aware training is repeated with formats `int2`,`int4` and `int8`. Then, the algorithm searches for the optimal data format for each layer. Finally, a last step of post-processing aims to remove the inevitable and unpredictable errors given by data-driven models like the TCNs. Hence, a filter is applied based on the natural dynamics of the heart that provide a reasonable range of the HR over time.
Thanks to the quantization optimization the resulted networks' size range from the biggest 1MB (in float) to the smallest $<$ 1kB. The largest model able to fit in the target embedded device STM32WB55 required $\approx$ 412kB reaching a MAE of 4.41 BPM.

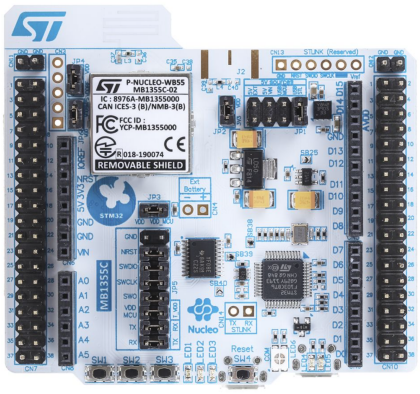
Chapter 4

Porting HR Detection on Wearable Devices

The focus of this work is the development of a complete system made of three main blocks. An interaction framework between a server and a client exploiting the Bluetooth LE communication protocol. The deployment of a TCN in a real embedded system and the development of a FSM that allows the collection of data, HR prediction and the transmission of data using the BLE stack. Finally a Graphic User Interface (GUI) has been created to enable the streaming of both the input data and the predictions in real-time. Moreover, the GUI allows to monitor the performance during the process which is a mandatory requirement when dealing with long periods of processing and acquisition, in order to make sure everything is working properly.
The aim of this chapter is to explain the steps and the setup used to create the aforementioned system. The on-board firmware was entirely developed using the STM32Cube environment and C code, while the client-side and the GUI were developed in Python 3.8.

4.1 Hardware setup

The hardware used during the development of this work were two: an evaluation board by ST Microelectronics and a smartwatch-form embedded systems (shown in Figure 4.1(b)), developed by ETH Zurich, which is an open source firmware and hardware project. The latter is a perfect solution to stream and directly evaluate real data, while the first is the perfect solution during the prototyping stage since its computing core is the same of the H-Watch. The **STM32WB55xx**[64], shown in Figure 4.1(a), is an ultra-low-power MCU that embeds Bluetooth™Low Energy. It is based on two independent cores, an ARM® Cortex™-M4 running at 64MHz



((a)) Evaluation Board [62].

((b)) HWatch [63].

Figure 4.1: Hardware setup

(called CPU1) and a ARM® Cortex™-M0+ running at 32MHz (called CPU2). The device includes $1MB$ high speed flash memory and different communication interfaces. Among these, the most used during the development of the system were:

- **USART** [31], which stands for Universal Synchronous Asynchronous Receiver Transmitter. It is a peripheral that receives serial data and stores them as parallel , and takes parallel data and transmits them as serial data. The main parameter to be set is the transmission and reception rate, called baud rate, that has as standardized values like 2400, 4800, 9600, 115200. The baud rate value used in this work is 115200. Then a parity bit can be used, which is an extra bit that may be added to each data word to detect transmission errors.
- **I2C**, inter-integrated circuit interface, which handles communications between the microcontroller and the serial I^2C bus controlling the sequencing, protocol, arbitration and timing. Despite being constrained in its hardware requirements, I2C bus provides good support for communication with several slow, on-board peripheral devices that are accessed intermittently [65].
- **SPI** [66], stands for Serial Peripheral Interface. It implements a full-duplex master and slave protocol, which means that both the agents can communicate at the same time. The data is synchronized using the rising/falling edge of the clock.

The first was used to simulate the reception of data in the evaluation board, while the other two interfaces interact with the sensors present in the smartwatch.
The choice of the evaluation board used was not arbitrary. In fact, the **H-Watch** has the same MCU of the board, **STM32WB55RGV6**. Additionally, the H-Watch embeds two sensors that allow the acquisition of signals needed for HR prediction:

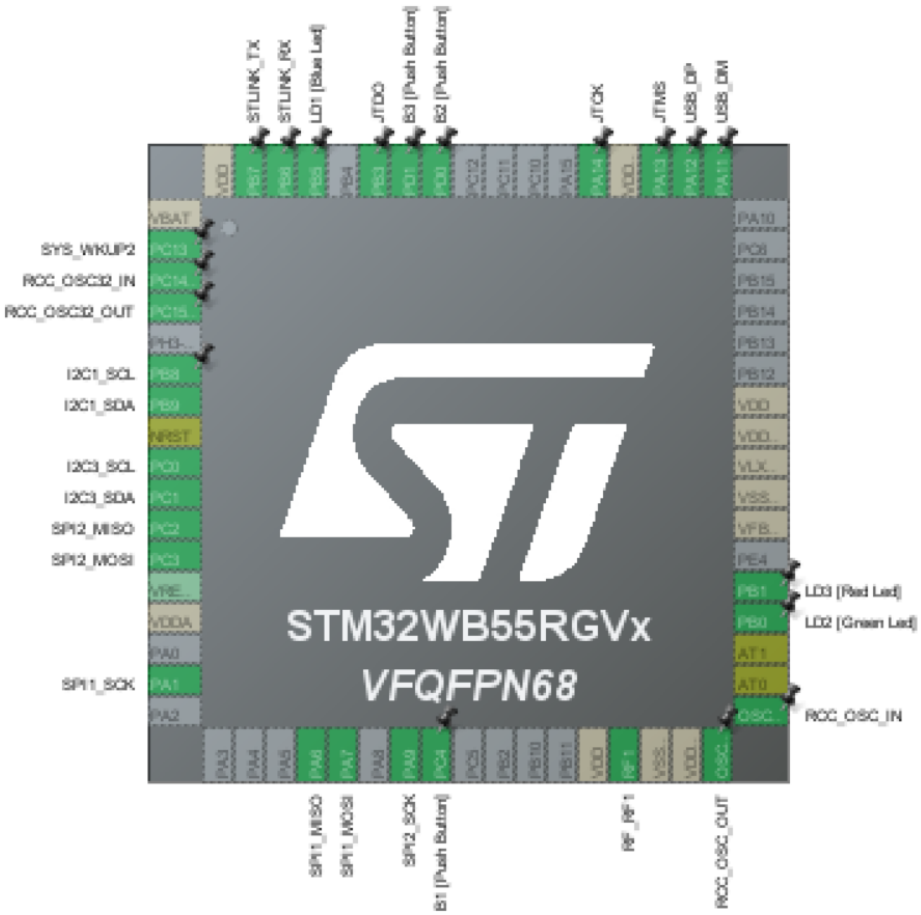


Figure 4.2: Evaluation board .ioc configuration taken from the tool STM32CubeMX provided by STMicroelectronics

- **MAX30101EFD+** [67], an High-Sensitivity Pulse Oximeter and Heart-Rate Sensor for Wearable Health. It includes internal LEDs, photo-detectors, optical elements, and low-noise electronics. Communication happens through a standard I2C-compatible interface.
- **LSM6DSM** [68], Ultra-low power, high accuracy and stability iNEMO 6DoF inertial measurement unit. Specifically is a system-in-package featuring a 3D digital accelerometer and a 3D digital gyroscope. For this sensor the communication is performed through SPI interface.

The framework provided by STMicroelectronics called STM32CubeIDE [69] is useful to write/generate, compile and debug the code for the STM32 microprocessors and microcontrollers. It is a all-in-one development tool based on Eclipse®/CDT™and

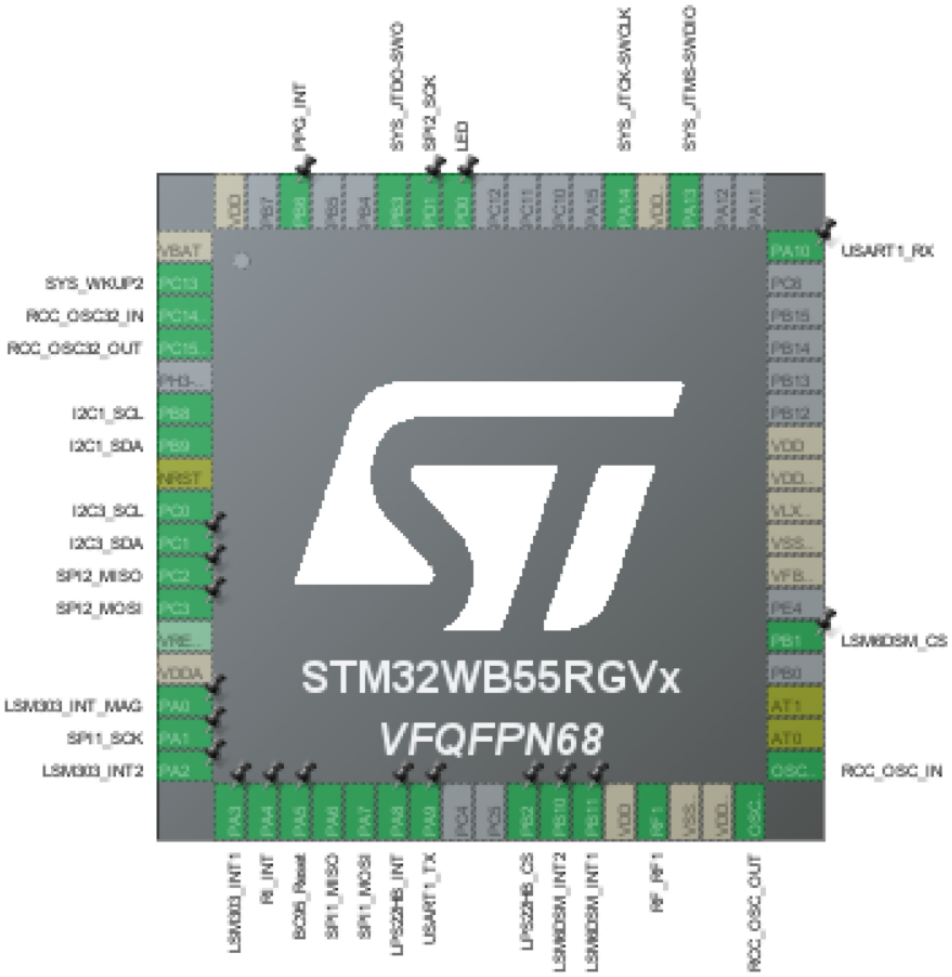


Figure 4.3: HWatch .ioc configuration taken from the tool STM32CubeMX provided by STMicroelectronics

GCC compiler tool-chains. To set the parameters and the configuration of all components in the MCU, STM32CubeMX [70] was used. This is a graphical tool that can be used as a stand alone software, or directly in the STM32CubeIDE integrated editor.
The easy-to-use interface allows, for example, to set the pin-out configuration with the help of automatic conflict resolution, or to set the configuration of peripherals and middle-ware functions. In Figure 4.2 and Figure 4.3, the pin-out configurations of respectively Evaluation Board and H-Watch are shown, which are quite similar except for the sensors-specific configurations.

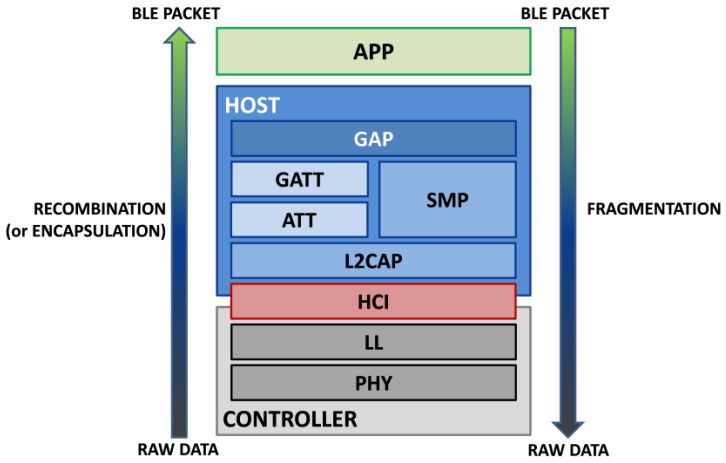


Figure 4.4: BLE protocol stack. The three main blocks are the Controller (grey), the Host (blue) and the App (green). The HCI (red) is the interface that manages the communication between the Controller and the Host. The rectangular frames represent the different layers of the protocol, and they are ordered in a stack, which starts from the bottom, with the PHY part, and ends at the higher level, that is the App. The arrows show how encapsulation and fragmentation work. [71]

4.2 Bluetooth

To communicate data and predictions between the FSM and the client device, the Bluetooth Low Energy [71] protocol was chosen. Nowadays, the vast majority of devices include BLE among their technologies making it a brilliant candidate to port the system to several applications. Moreover, as the name suggest BLE is characterized by low power consumption which allows it to be embedded in small devices with small batteries [71].
BLE is a communication protocol composed of a stack with three main blocks, which we can see in Figure 4.4. The architecture structure is very similar to the classic Bluetooth, so that it is possible to develop applications compatible with both protocols. Each block is composed of some sub-layers, and each layer incorporates its own lower layers.

- *Application*: the highest block of the stack, it allow direct interaction with the user, defining some profiles and common reusable functions.

- The *Host* block includes:

  – **Generic Access Profile** (GAP): the highest layer within the Host block. It has the main role to interface with the Application layer and hence, with

the user. It specifies roles, modes, procedures, connection establishing and security.

  - **Generic Attribute Profile** (GATT): it defines the methods to exchange profiles' information and data. Profiles are a specific structure of data transmitting, they are hierarchically organized into *services*, organized as well into *characteristics*. During the connection establishment, the server exposes its services and characteristics to the client in order to define how the connection will be structured. Each characteristic describe the transmission of a specific type of data, and it is made of a descriptor, a value and some properties. The latter are used to communicate to the client what operations are allowed. The most common properties are: *readable*, the client can only read the value; *writable*, the client can write a new value; *notifiable*, the client receives a notification when the server update the characteristic value.
  - **Logical Link Control and Adaptation Protocol** (L2CAP): it takes the data from the lower layers and transform it into standard BLE packet format, and vice versa.
  - **Attribute Protocol** (ATT): this layer defines the *client-server* architecture, where the latter receives the data from the server, which in turn sends data to the client. *Universally Unique Identifier* (UUID) is a set of permissions and a value to organize the data into attributes. The GATT layer incorporates the ATT to perform connections.
  - **Security Manager Protocol** (SMP): a set of security algorithms with the goal of encrypting and decrypting data packages.
  - **Host Controller Interface** (HCI), Host side: is the layer that controls the communication between the Controller's side and the Host's side. Specifically, it defines commands and events to translate raw data into data packets via serial port to the Controller layer.

- The *Controller* block includes:

  - **Host Controller Interface** (HCI), Controller side: is the opposite of the Host side. Hence, it sends data packets from the Controller to the Host.
  - **Link Layer** (LL): is a combination of a hardware and software defining the type of communication to be instanced between the devices.
  - **Physical Layer** (PHY): by construction BLE operates in the Industrial, Scientific and Medical (ISM) band in $2.4 - 2.5GHz$, the same as classical Bluetooth (BR/EDR) and Wi-fi. In particular, the BLE frequency goes from $2.400GHz$ and $2.4835GHz$ and is divided into 40 channels. A

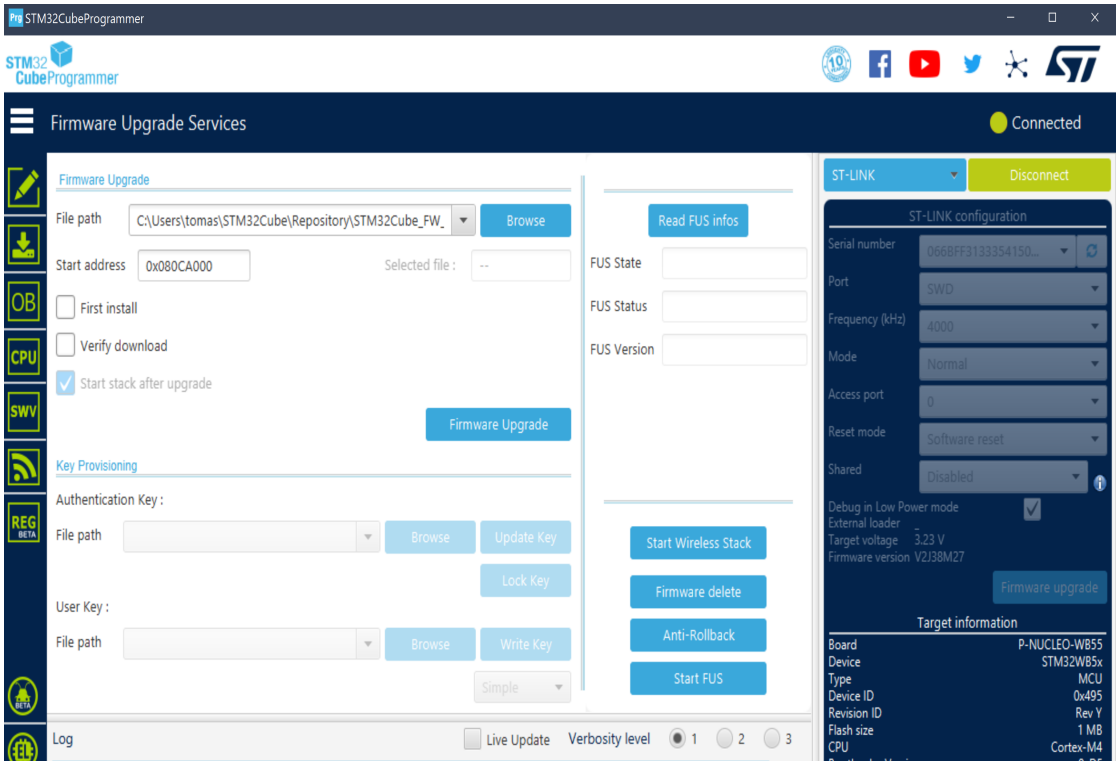


Figure 4.5: STM32CubeProgrammer tool in section Firmware Update Service used to upgrade the FUS version and flash the `BLE_stack` binary file.

strategy called Adaptive Frequency Hopping is used to define pseudo-randomly a communication channel so to avoid interference with other wireless protocols.

4.2.1 BLE on Hardware device - *Server*

Both the h-watch and the development board are programmed to play the *server* side of the whole application. The first mandatory step to be performed on this side, is to upgrade the Firmware Upgrade Service (FUS) binary and flash the devices with the right BLE-firmware stack. The versions compatible with both devices are respectively `stm32wb5x_FUS_fw_for_fus_0_5_3` and `stm32wb5x_BLE_Stack_full_fw.bin` .

To do so, the STM32CubeProgrammer [72] (shown in Figure 4.5) was used. This is a graphical configuration tool to read, write and verify debug interfaces, boot-loader interfaces and memory internal and external and so on.

ST Microlectronics provides several template application to help the user develop any kind of application. In this work, the `P2P_server` template (where P2P stands

for peer-to-peer) was explored to build the server side application. This is based on a **sequencer** that execute a background scheduling function and enter a secure Low-power mode when there is no activity to be executed [73]. The implementation of the sequencer requires many steps where some of them are mandatory:

1. Set the maximum number of supported functions. In this work the default parameter is used, i.e., `UTIL_SEQ_CONF_TASK_NBR > 32`
2. Register a function to be executed by the sequencer which is associated with a signal in the sequencer itself: `UTIL_SEQ_RegTask( 1≪ CFG_TASK_SEND_DATA_ID, UTIL_SEQ_RFU, P2PS_Send_Data)`
3. Start the sequencer to run a background while loop.
4. Call `UTIL_SEQ_SetTask()` every time the function need to be executed. The sequencer will decide which task to call depending on the priority flag associated with each task (0 is the highest).

To establish a connection with the client, the server works at the **GAP** level of BLE, which can discover the remote device and initiate the connection.
Regarding the configuration of services and characteristics configuration the user can choose between two types of profiles. The first is the *Standard* profile, defined by Bluetooth SIG [74]. In this case, the specifications of the characteristics and the services cannot be changed. The second is the *Proprietary* profile, this defines non-standard profiles, hence the user can define custom services and characteristics. This work is based upon a standard profile.
The `P2P_server` template implements the services and characteristics listed in Table 4.1 and Table 4.2.

Groups	**Service**	**Characteristic**	**Size**	**Mode**	**UUID**
Led Button control	P2P service	-	-	-	0000FE40-cc7a-482a-984a-7fed5b3e58f
	-	Write	2	Read / Write	0000FE41-8e22-4541-9d4c-21edae82ed19
	-	Notify	2	Notify	0000FE42-8e22-4541-94dc-21edae82ed19

Table 4.1: Table of the service and characteristic UUIs for the P2P server application [73].

	Octets LSB	**0**	**1**
Write	Name	Device selection	LED control
	Value	- 0x01: P2P server 1 - 0x02: P2P server 2 - 0x0x: P2P server x - 0x00: All	- 0x00 LED off -0x01 LED on - 0x02 Thread
	Octets LSB	**0**	**1**
Notify	Name	Device selection	LED control
	Value	- 0x01: P2P server 1 - 0x02: P2P server 2 - 0x0x: P2P server x - 0x00: All	- 0x00 switch off -0x01 switch on

Table 4.2: Table of P2P profiles' specification [73].

In this work, the *Notify* standard profile has been used. In particular, the client receives a notification every time a new sample (both PPG and accelerometer data) is available. To do implement the Notify profile is exploited the function `P2PS_STM_App_Update_Char`, shown in Listing 4.1. This takes as inputs the UUID of the characteristc and the BLE payload, i.e. the data to be sent.

Listing 4.1: Snippet of code implementing the send data functionality over BLe using the Notify service

```
tBleStatus P2PS_STM_App_Update_Char(uint16_t UUID, uint8_t *pPayload)
{
  tBleStatus result = BLE_STATUS_INVALID_PARAMS;
  switch(UUID)
  {
    case P2P_NOTIFY_CHAR_UUID:

     result = aci_gatt_update_char_value(
            aPeerToPeerContext.PeerToPeerSvcHdle,
            aPeerToPeerContext.P2PNotifyServerToClientCharHdle,
            0, /* charValOffset */
            20, /* charValueLen */
            (uint8_t *)  pPayload);

      break;

    default:
      break;
  }

  return result;
```

```
22 }
```

In Listing 4.1 note that the parameter *charValueLen* depends on the amount of data sent. In this case, we want to sent five floating-point values, where 1 value is for the PPG signal, 3 values are for the tri-axial accelerometer values, and finally 1 value is reserved for the HR prediction. Each float requires 4 bytes, hence the required length of the payload is set to 20 bytes.

4.2.2 BLE on host device - *Client*

In the system developed in this work, the client side is represented by a program in Python on an host computer, but could be extended to any other device equipped with BLE stack like a smartphone.
To establish the connection the **bleak** [75] library is used. Bleak stands for *Bluetooth Low Energy platform Agnostic Klient* and is a free software supporting Windows 10, Linux, and OS X/macOS. 5 The implementation of Bleak provides a **GATT client** that gives support for reading, writing and getting notifications from the server. The library is thought to work in an **asynchronous** fashion.
Asynchronous programming is based on the **async/await** pattern introduced in Python 3.5. This feature allows the programmer to write a single continuous set of statements in a direct programming style that will be performed in the correct order, even when they are run asynchronously as a set of separate events [76]. In addition, **Coroutines** are functions that allow suspension and resume of their execution through the use of the keyword `async` and `await`. By all means, if these tasks are not coordinated properly, the interleaving between tasks from different activities can be source of bugs [77].
In Python, this kind of programming style can be implemented with the use of **AsyncIO**[78] library. This library gives the possibility to write concurrent code, hence allowing the development of coroutines and awaitable tasks.

Listing 4.2: asyncio coroutine simple example

```
1 async def main(address):
2     async with BleakClient(address) as client:
3         model_number = await client.read_gatt_char(MODEL_NBR_UUID)
4         print("Model Number: {0}".format("".join(map(chr,
    model_number))))
5
6 asyncio.run(main(address))
```

The library `bleak` is thus combined with `asyncio` to manage the Bluetooth connection with the server. As we can see in the example code in Listing 4.2, we

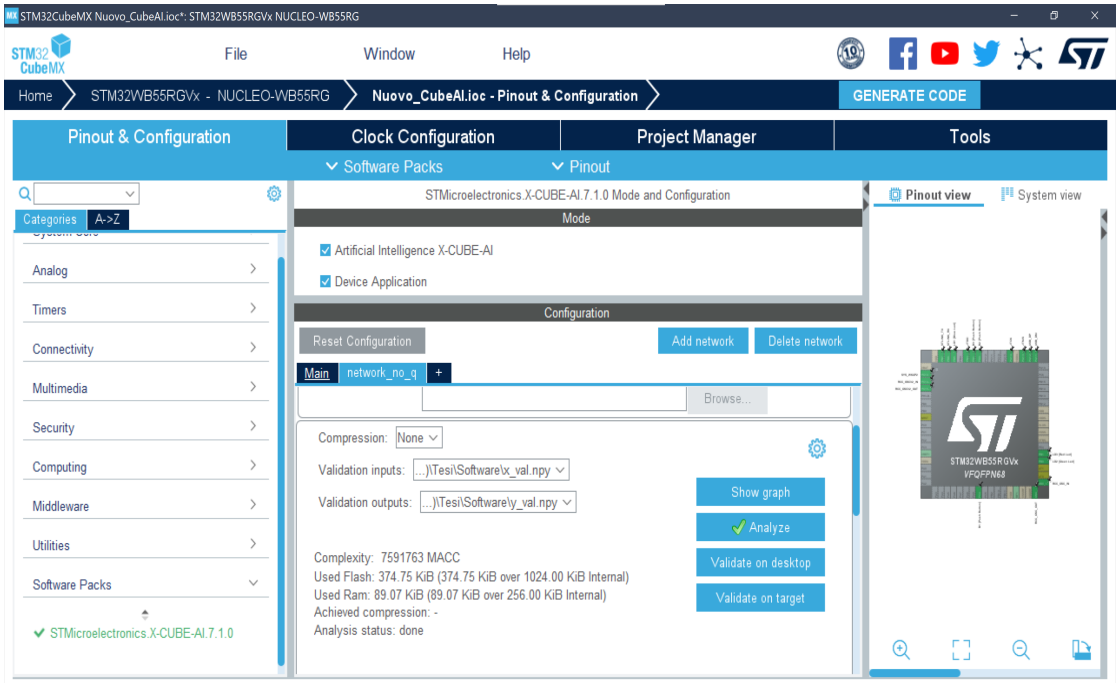


Figure 4.6: STM32CubeAI tool Framework

create a coroutine to make the function used to establish the connection with the MCU awaitable. Also the functions that send and receives data to/from the server depending on the specific application are implemented in this way.

4.3 STMCube.AI

To deploy the DNN on the MCU, we employed the STMCube.AI [79] environment provided by STMicroelectronics within the same CubeIDE coding environment. The software provide a graphic interface to automatically load a pre-trained model without writing all the C code required for the creation of the model, the manage of weights and activation functions, generating the STM32-optimized neural network. Different models' formats are supported, among which Keras and TensorflowLite™and ONNx.
The tool used as extension in STM32CubeMX framework is shown in Figure 4.6. Beyond the codegen capabilities, it also provides some useful methods to analyse the network: checking the architecture with analyse, the validation on desktop and validation on target device. The *analyse* button allows to run an analysis of the model providing information on its dimension (RAM and FLASH memories) and the complexity (number of MAC operations). The *validation on desktop*

button gives the possibility to validate the translated C model. Moreover, both the a reference ground truth or auto-generated random samples are provided some metrics are calculated, among which the MAE. The *validation on target* can be performed only when the device is connected through the USART to the ST-Link, allowing to compile, program and run the network [80]. This features resulted useful when analysing the compatibility of the network with the MCU before generating the actual code. Moreover, note that Cube.AI gives the possibility to directly compress the given model to `int4` or `int8`, but since the model was compressed with TensorflowLite, none compression was applied at this step.

4.4 Dalia Dataset

The dataset used as a benchmark in this work is PPGDalia proposed by A. Reiss et al.[12]. The dataset comes to fill the lack of extended datasets that comprehends both PPG and accelerometer data. The name is an acronym for *dataset for motion compensation and heart rate estimation in Daily Life Activities*. In fact, it consists of records of 8 different activities recorded in periods in between. The subjects included in the dataset are 15 with a total of 36 hours of recorded data. The collection of data was performed employing a chest-worn device and a wrist-worn device. The first was used to measure ECG signals, respiration capture with an inductive plethysmograph sensor, and 3D-accelerometer data. The wearable device, instead, provided a PPG sensor with four kinds of LEDs (two red and two green), and an inertial three-axis acceleration sensor sampling at $32Hz$.

As said, 15 subjects were involved in the data collection, 8 females and 7 males, all of them students or employees in good health conditions, with age ranging from 21 to 55 years old. Further information about the subject's height, weight, skin, and fitness level are provided in the documentation of the dataset itself.

The activities recorded are divided into three levels of intensity: low, medium and high. Moreover, the authors decided to consider both periodic and aperiodic activities with different physical efforts required to generate highly diversified heart rates. The different activities are briefly described in Table 4.3. The data collection protocol took around 150 min per subject, except for one of them, for which only 90 min of data are valid due to a hardware issue. In order to provide reliable golden labels, the authors used R-peak correction on the ECG signals to obtain the ground-truth heart rates. Moreover, data are segmented using a sliding window approach where each window has length of 8s, and a shift of 2s. Therefore, there is an overlap of 6s between two consecutive windows.

Activity	**Duration** [min]	**Description**
Sitting	10	sitting still and reading on a laptop or magazines
Ascending and descending stairs	5	climb 6 floors of stairs up and going down twice
Table soccer	5	the game was played one-on-one with the supervisor
Cycling	8	the route was about 2 km length with different road conditions
Driving	15	the drive was 15 min long with a common car.
Lunch break	30	includes queuing to get food and eating at a table
Walking	10	-
Working	20	working activities were mainly at a computer.

Table 4.3: PPGDalia activities summarized. [12]

4.5 Procedure

This section aims is to explain the whole logical and practical reasoning behind the development of the control FSM.
Figure 4.7represents the flowchart of control algorithm running on the STM32Wb55. The algorithm starts with the initialization of all components, e.g cube.AI and BLE component along with all the standard interfaces and configurations needed by the MCU. Then, the sequencer of the BLE application runs in the background, looking for a connection. When the connection is established with the client, this sends a command to start the timer on the MCU. The timer `TIM16` on the board is used to set the sampling of data at $32Hz$. Every time the timer is called it sets a task that is in charge of running the function that manages the sending and reception of data to/from the client. Moreover, to re-create the same sliding window approach used in the PPGDAlia dataset, two global counters are created.
In the flowchart, we can visually identify two main blocks, the first representing the very first batch arriving in the MCU, and the second managing all the consecutive batches. The MCU receive a sample at a time through a serial connection with the client.
This sample consists of a PPG signal, and three values of the 3D-accelerometer data. Hence, to recreate a window size of $8s$, we need 256 samples $(8 * 32)$. Every time the sample arrives, the `counter_1` is checked to see whether is up to 256, if it

is not, the window is not yet completed so the sample is sent through Bluetooth LE to the client in order to be plotted; if the `counter_1` is 256, the window is completed and the entire batch is sent as input to the *Cube.AI* network, which returns the predicted HR. Then, the last received sample and the predicted HR are sent to the client. A global flag is set to indicate that the first batch is completed, so that the next time the function is called it will manage successive batches.
The last block of the flowchart represents the successive batches after the very first. Since in PPGDalia dataset the windows are shifted of $2s$, we need to take $6s$ seconds of data from the previous batch and $2s$ of *new* data. This results in coping 192 *old* samples and receiving 64 *new* samples, forming the window of 256 samples required. Thus, as previously done, `counter_2` is used to check the number of new data arrived, if it is less than 64, the sample only is sent to the client through Bluetooth; when `counter_2` is 64, the new window of data is sent to the Cube.AI and the new prediction is obtained. The latter and the last received sample are sent to the client. This procedure continues until no more data is received in the MCU.
Note that this whole procedure is needed to simulate the sampling of data since the Nucleo STM32WB55 cannot do it directly. When using the H-Watch smartwatch, the receiving of data from the serial port is substituted by sampling directly from the PPG sensor (MAX30101) and the three-axis accelerometer sensor (LSM6DSM) data. Nonetheless, the managing of the sliding window is maintained.
From the client side, the algorithm starts by scanning for available devices and creating a connection with the provided IP address belonging to the Nucleo board or the H-Watch. Once the connection is established, the GATT client send a command, defined as the char value `x00x02`, to start the timer `TIM16`. After that, the client subscribes to the *Notify* profile providing the Notify characteristic UUID and a callback. Then, if the server is the Nucleo board, send a sample through serial communication and wait to receive a notification with the data from the server. The callback is in charge of taking the value from the payload, cast it to float and finally save it in a local dataframe.

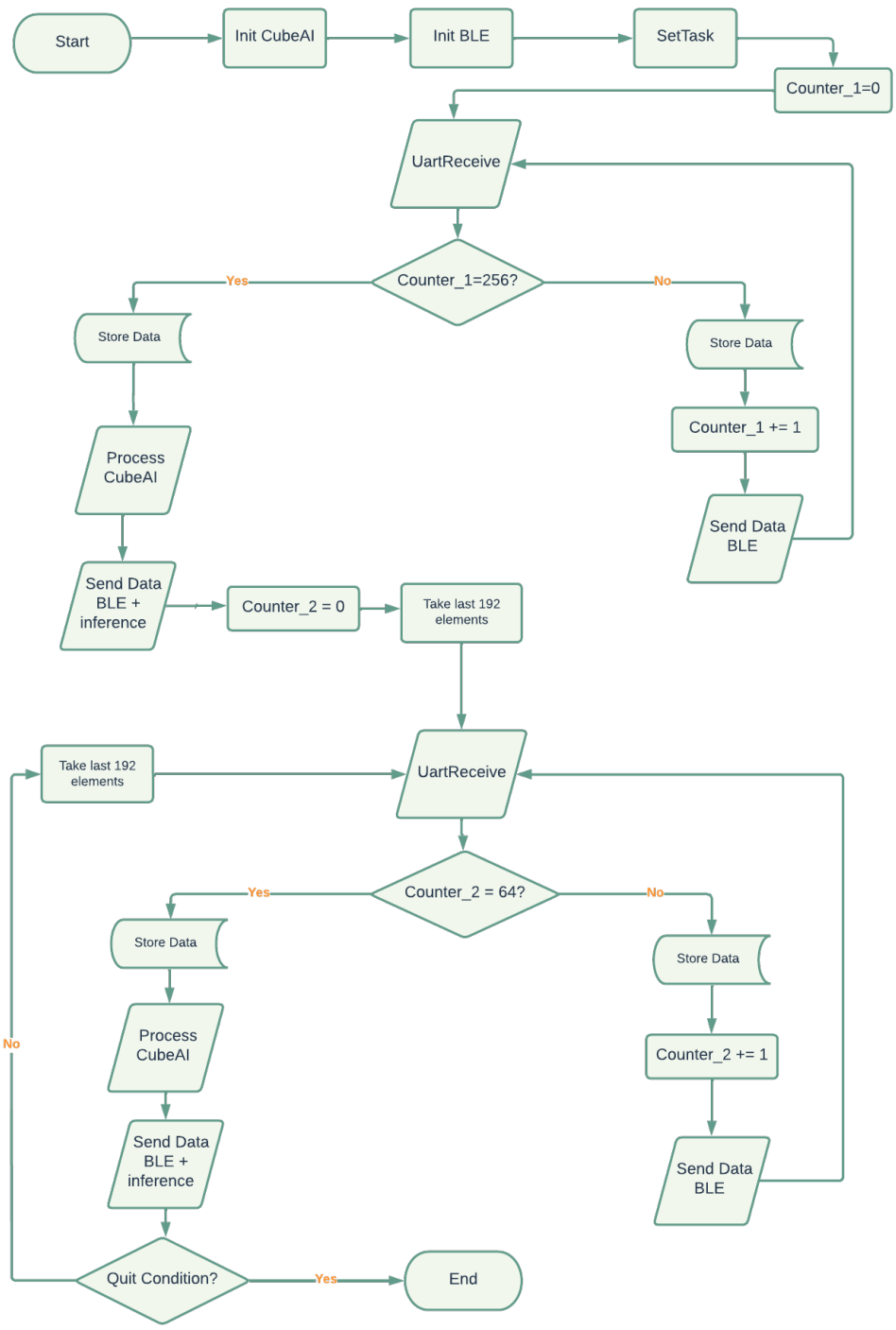


Figure 4.7: Flowchart firmware logic

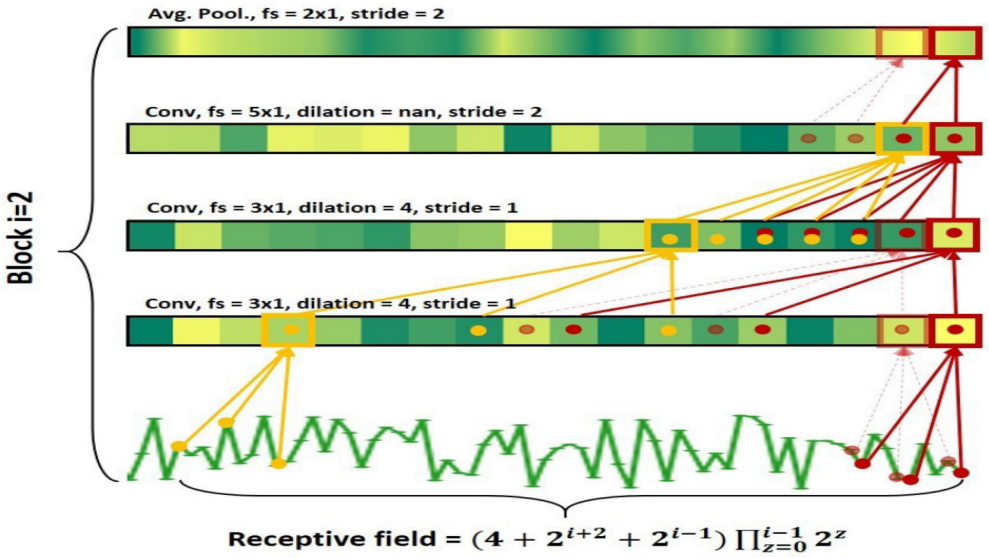


Figure 4.8: Illustration of a convolutional block in TEMPONet with two dilation factor $d = 4$, stride $s = 2$ and average pooling. [83]

4.6 Neural Network Design

Based on the works presented in [59],[60], and [58] already discussed in sections 3.2.1 and 3.2.2 respectively, the model chosen to be deployed in the STM32WB55 is a Temporal Convolutional Network. According to [81], the three main characteristics of TCNs are:

1. Layer-wise computation instead of per-frame sequential update. This means that every step is updated simultaneously.
2. Convolutions computed across time.
3. For each frame there is a prediction that is function of a fixed period of time, called receptive field.

The model is written Tensorflow 2.8 [82], which is one of the most famous open source library that provides different levels of abstraction to create neural networks. Among these, the *Sequential* API allows to easily write block-based networks which fits perfectly the innate modular structure of a TCN. The TCNs used in this work is strictly correlated to the TEMPONet model mentioned in section 3.2.1. The model was first described in [83] and was implemented for a gesture recognition task. The architecture is made of TCN and Convolutional blocks (an example of Convolutional block is shown in Figure 4.8). This kind of structure enables a more

powerful processing of times series, since the temporal dimension is consumed at a lower pace [83]. The input and output layers have been adapted to the shape of the input and output of the PPGDalia dataset. Moreover, the last layer is substituted with a single neuron in order to perform the regression task of HR prediction.
The architecture chosen for the final deployment has three main modules, each of them made of:

- Two Temporal convolutional blocks with variable dilation factors and zero padding.
- One Convolutional block, made of a 1-dimensional Convolutional layer and a average-pooling layer.

Then, as in the original TEMPONet we have two regression blocks, made of a Dense layer, ReLU non-linearity and Batch Normalization.
To test the network on the sTM32WB55 two main configurations of this basic network have been considered. The two network configurations are summarized in Table 5.1. Where the dilation factors are used on the 1D-Convolutional layers in

	Dilation Factors	**Channels**
Architecture 1	[2, 2, 1, 4, 4,8, 8]	[32, 32, 64, 64, 64, 128, 128, 128, 128, 256, 128]
Architecture 2	[2, 2, 1, 4, 4,8, 8]	[27, 26, 60, 58, 64, 80, 27, 29, 38, 44, 57]

Table 4.4: Architectures' structure tested

the TCN blocks. Finally, all layers include the ReLU non-linear activation function and a Batch Normalization layer.In this way, we can test a bigger and a smaller network to see if they are able to fit into the embedded system. The architectures were obtained following the work of [60], exploiting the use of the NAS tool called MorphNet, which is a lightweight tool that concentrates the hyper-parameters optimization on the *number of channels* per layer. In particular, a structured-channel pruning approach is used, divided into two main steps. First, the size of the network is reduced forcing all the weights of a channel to a small magnitude, and the channels whose total magnitude is under a tuned threshold are removed. Then, to mitigate the performance drop due to the pruning step, an *expansion* step is performed, up-scaling the number of channels by a constant factor. The loss function employed for the training is the *Log-Cosh function*. This function has proven to outperform the more common Root Mean Squared Error (RMSE) loss,

favoring the convergence near the minimum, thanks to its smoother behavior in that point [60]. The Log-Cosh loss is defined by:

$$L = \sum_{i=1}^{n} log(cosh(y_i - \hat{y}_i)) \tag{4.1}$$

The models have been trained for 100 epochs, with *Adam* optimizer and starting learning rate set to 0.001.

4.6.1 Moving Average

In order to compare the results obtained with the networks, a simple moving average-based algorithm has been applied to the Dalia dataset to get the predictions with a classical method. The moving average is a technique commonly used in statistics and data analysis to get an idea of the trend of a stream of data. Specifically, the deterministic algorithm calculates the average of a subset of values from the dataset, the size of the subset is called *window*. The average is called *moving* since the window is always shifted by a certain step (in our case we considered 2s of overlap between two consecutive windows, following the DaLia dataset segmentation) to include new data and "forget" the old data.
In the contest of our application, the mean values are used afterwards for the computation of the R-R peak interval, which is the time interval between two consecutive peaks. In fact, the mean value of each window is used as a threshold to see whether each point can be considered a peak or not.

4.7 GUI

The Bluetooth LE client application is integrated with a Graphical User Interface, that aims to plot in real-time the data received from the server MCU.
The GUI is created as a Dashboard based upon the library Plotly Dashboard [84]. The dashboard created is rendered as a web page running on local host, but could also be deployed on Virtual Machines (VMs) or Kubernetes clusters, and accessed by URLs [85].
Dashboards are a great way to quickly visualize data and capture insights information from it. In particular, if the data are collected in real-time having an immediate response of the data quality and consistency can help in finding mistakes on the fly and in avoiding measurements' errors.
Plotly Dash is built upon Flask, Plotly.js and React.js, and allows to create interactive plots in Python, using predominantly the *plotly.py* library. The two main building blocks of Dash are the *Layout* and *Callbacks*. The first indicates which components are used and how they are organized, creating a hierarchical

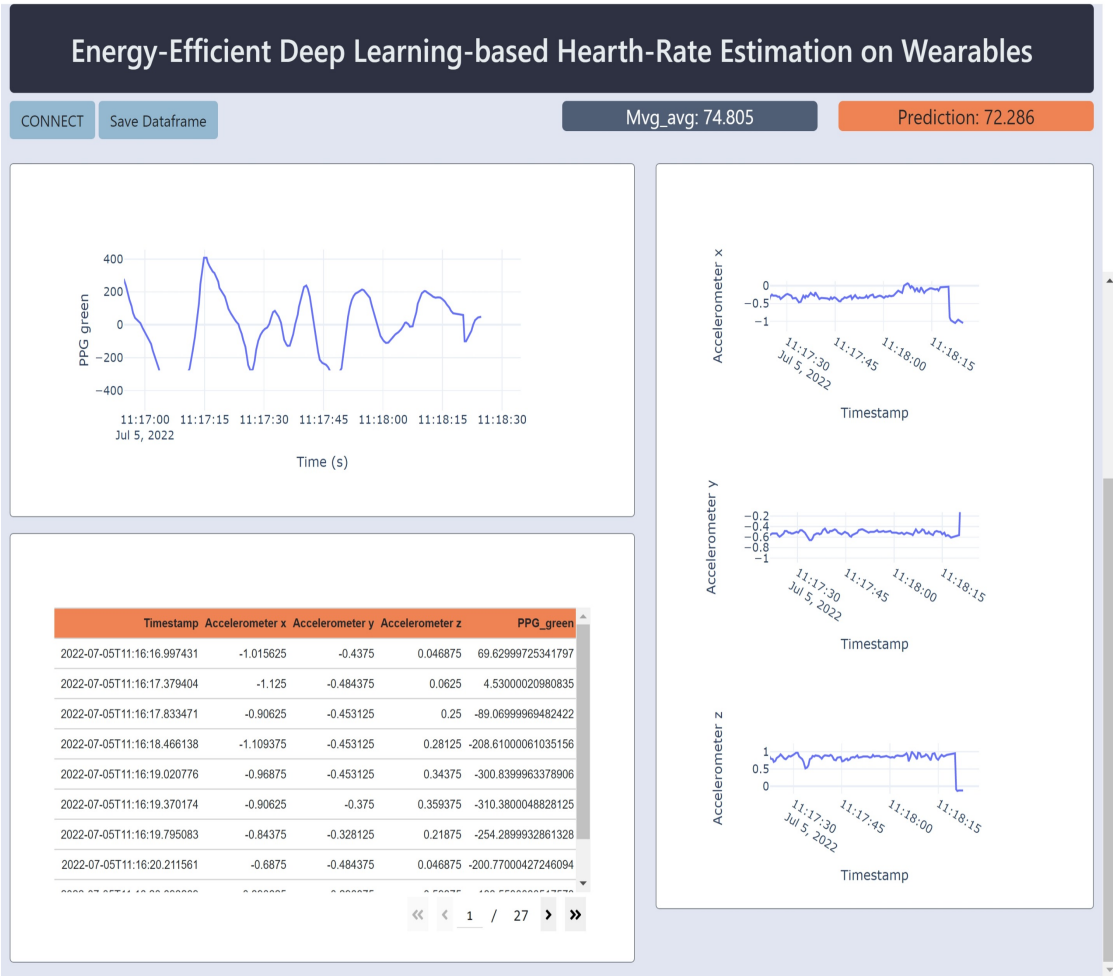


Figure 4.9: Screenshot of the GUI created

representation. Dash provide the use of HTML components to create plots, tables, headings and so on, or specific Dash Core Components that are thought to be interactive. The callbacks function are used to make the the dashboard interactive. In fact, they take as parameters both input and output properties of a component, every time a input is changed the function gets called automatically and it updates the output component based on the specified logic. Since the outputs *react* to changes, this is also called *"Reactive Programming"*[85].

Figure 4.9 shown the developed GUI. As we can see, there are three graphs on the left, each showing an axis of the 3D-accelerometer, and one graph on the right showing the waveform of the PPG signal. Moreover, we found a table listing all the numerical values of the data. The table is placed underneath the PPG graphs and gives the possibility to save the raw data as a *.csv* file using the toggle button

named *"Save Dataframe"*. On top of the page, the HR value obtained the simple moving averaging algorithm is shown in *blue*, while the predicted value using the TCN deployed on the MCU is shown in *orange*.
The connect button triggers the scanning for the server's IP and the connection with it through Bluetooth LE. Once the connection is established, the client sent a command to the MCU to start the Timer16 as explained in Section 4.5. After that it creates a subscription to the notify service of the BLE server. In this way, every time a new sample is sent from the server to the client, the latter receives a notification and the new data is added to the DataFrame and plotted on the graphs.

Chapter 5

Experimental Results

This chapter analyses the results obtained with the two architecture defined in 4.6. In particular, we first explore the dimension and the complexity of the two architecture, and then we analyse the power consumption of the chosen model in different configurations.
The Bluetooth LE communication protocol is used in the experimental set-up to send the data samples between the server, i.e. the MCU, and the client, i.e. the application on a second device like a smartphone or a computer. The transmitted data is characterised according to whether the network inference is performed on the embedded system or not. In case of inference on edge, the BLE stack would be used to send the single predictions performed by the network, i.e. a single float value. On the other hand, if we need to send all the data to the client to make the inference, we will need to send 256 or 64 samples of values (1 PGG value and 3 accelerometer values) in order to make the very first prediction and all the successive ones respectively.
As we will see in the following sections, the architectures consume more power in comparison with the continuous stream of the data since the inference time is considerably long.

5.1 Models tested: size and complexity

Recalling that the total memory available of the STM32WB55 MCU is $1MB$, the two architecture has been verified exploiting the STM32Cube.AI tool that allows to easily check the size of the models.
In Table 5.1 are summarized the results in terms of size, number of MAC operations and parameters. In particular, we can see that *Arch. 1*, the biggest, counts a total of $429k$ parameters and a size of $1.63MiB$. This unfortunately overflows the total memory available in the MCU. The consequent step was to apply some

	Size	Parameters	MAC
Arch. 1	1.63 MiB	429,185	13997953
Qarch. 1	917.75 KiB	939,466	35614977
Arch. 2	374.75 KiB	97,241	7591763

Table 5.1: Results of models' complexity

optimization techniques discussed in section 2.4. Therefore, both quantization-aware and post-training quantization were implemented in this architecture to understand if the reduced model can fit the memory of the MCU, and still achieves a good accuracy. As a downside, the dilation factor had to be removed by setting the parameter $\mathbf{dil = 1}$ since the tool STMCube.AI does not support quantized model with dilation factor different from 1 yet. The quantization-aware training of Convolutional layers is not yet directly supported by TensoflowLite, hence a custom quantization function [86] has been exploited to quantize the specific layer. In particular, the function allows describing the behaviour of the network to quantize weights, activations and outputs of a layer. All the layers except for the Batch Normalization have been quantized to `int8`. This quantized version of *Arch. 1*, indicated as *Qarch. 1* reaches a size of $917.75KiB$ with $939k$ parameters, which unfortunately was still too much to fit in the embedded MCU.
To tackle the issue, *Arch. 2* directly reduces the number of channels in each Convolutional layer and therefore the number of parameters, consequently reducing the size and latency of the network by simply changing its architecture and without reducing the precision of the computations. In fact, *Arch. 2* consists of $97k$ parameters taking up $375.75KiB$ of memory space. In this case, we are able to fit this second architecture into the embedded system.
Arch. 2 is therefore chosen to be deployed on the MCU. In Figure 5.1, we can see the results of the analysis of the network using the graphical interface provided by STM32Cube.AI. The inference of the network resulted to be quite slow, having an average time of around $2s$.

5.2 Evaluation

To evaluate the performance of all the architectures we used as benchmark dataset the DaLia dataset presented in section 4.4. The predictions obtained on the PPGDalia dataset are based on the MAE metric, used in mostly all the works on

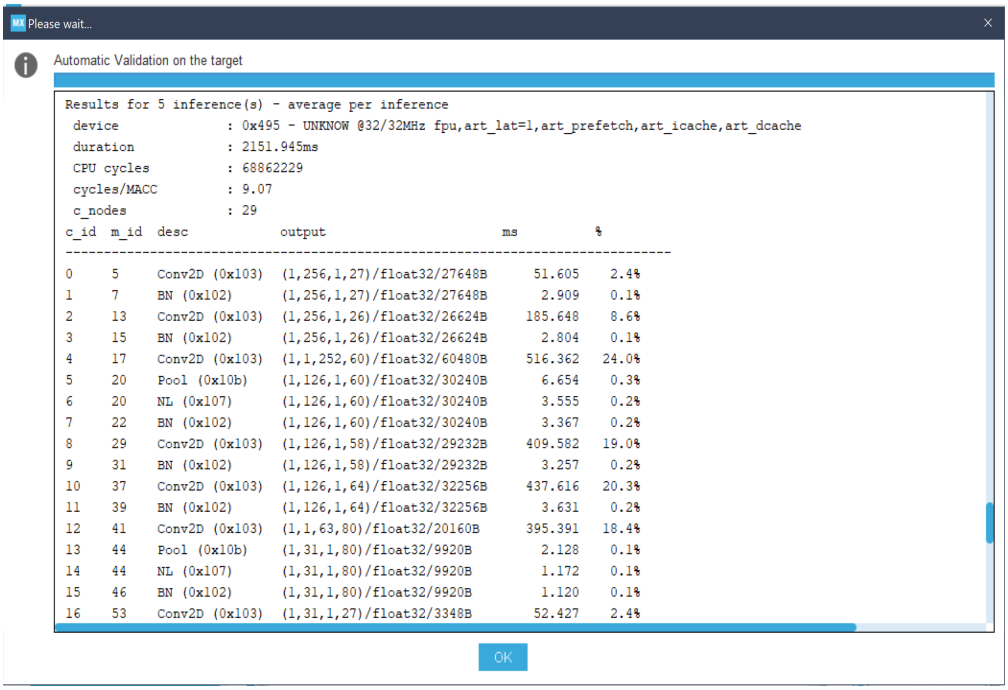


Figure 5.1: STM32Cube.AI Inference of Target results on 5 samples taken from PPGDalia.

PPG data analysis. Moreover, this metric is one of the most common metrics used for regression tasks in general and it gives an averaged magnitude of the error of the predictions.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2 \tag{5.1}$$

Where n is the total number of examples, y_i is the true value and $\hat{y}_i$ is the predicted value. In our case, we obtain the absolute error between the golden HR represented by the ECG values, and the predictions obtained with each architecture from the PPG signals.
As illustrated in Figure 5.2, *Arch. 1* which is the biggest, presents a MAE of $2.442\,BPM$, while its quantized version has a predictable slightly higher MAE value of $2.519\,BPM$. Despite having a long inference time, the *Arch. 2* reaches a good value of MAE **2.364** on the whole dataset, without making any difference neither on subjects nor the activity. The variation of the metric values between the two architectures is accountable to some overfitting during the training of *Arch. 1*. In Figure 5.3 we can see the two waveforms that represent the value of the first 256 HR predictions compared with the true values taken from the golden HR in the DaLia dataset, having on the y-axis the BPM values and on the x-axis the

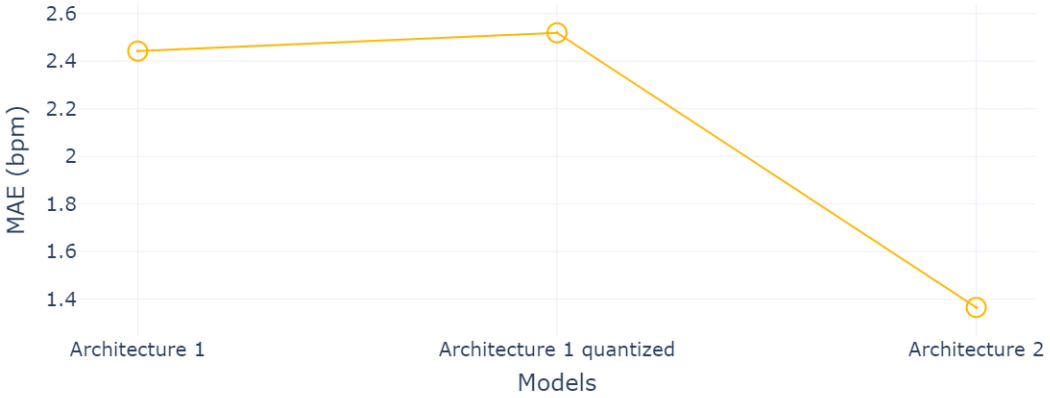


Figure 5.2: Comparison of the MAE values of all the three architectures

predictions' number. We can see that they are almost overlapped, mirroring the good level of MAE reached.
Moreover, as anticipated in section 4.6.1, the results of the network are compared with the results obtained with an algorithm based on a simple moving averages and the so-called R-R interval. The R-R interval is the time between two successive heartbeats. The RR interval and HR are hyperbolically related ($HR \times R - Rinterval = 60000$) [87]. The MAE resulted with this algorithm is **13.36** due to errors caused by MA-induced noise, which is definitely much higher than the result obtained with the TCN model. In fact, in this case, Figure 5.4, that has, as before, on the `y`-axis there is the BPM value and on the `x`-axis the predictions, shows the waveforms differ significantly on many points.

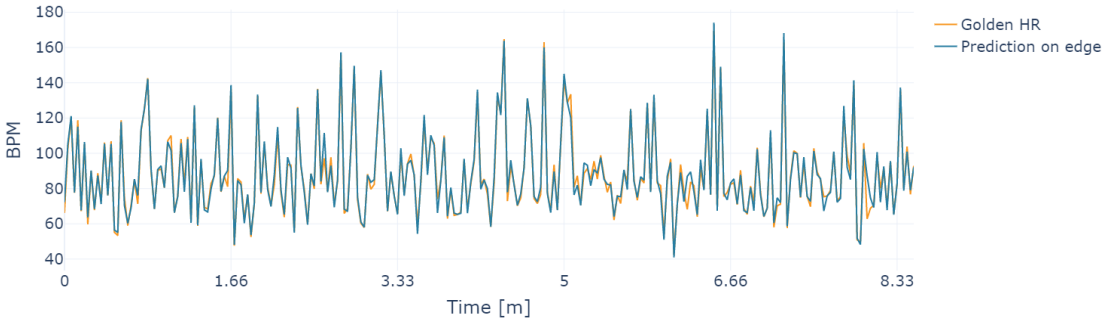


Figure 5.3: Prediction comparison between the True value of PPGDalia (blue) and the predicted values from inference on STM32WB55 (red). For visualization sake the first 256 example are shown.

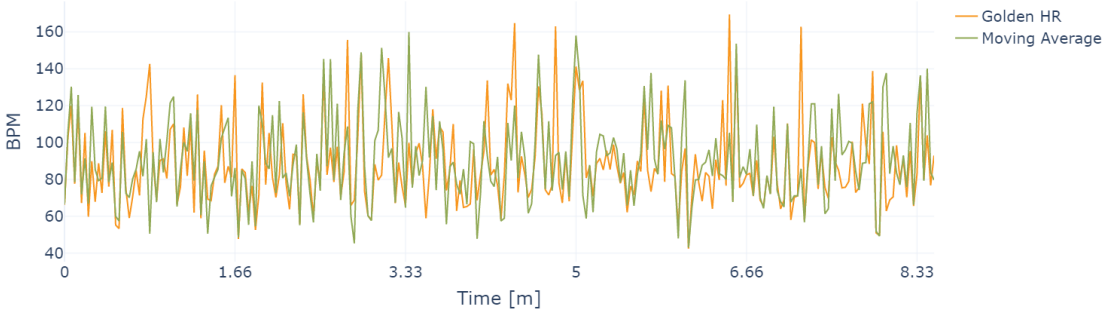


Figure 5.4: Prediction comparison between the True value of PPGDalia and the predictions from the Moving Average algorithm. For visualization sake the first 256 example are shown.

5.3 Energy Consumption

The analysis of energy consumed by deep learning model has gained importance over the past years. Especially when dealing with hardware-constrained and battery-powered systems the energy is strictly linked to the lifetime of the target device. Therefore, the following section aims to explore the power consumption of the system developed in this work considering its two main characteristics, the BLE stack and the deep learning inference.
The measurements have been carried out with a multimeter Keysight HP 34401a [88] and with the Nucleo board STM32WB55, sampling the values of the current for about 2 minutes. The following parameters are set for all the configurations:

- Sampling at $20Hz$
- Full scale at $100mA$
- Fast mode with 4 maximum digits
- External power supply at $5V$ for the board

We tested four different configurations to compare the energy efficiency of the model deployed on the MCU.
The first configuration tested was the inference of the Network. The project on the STM32CubeIDE was modified in order to run only the network deployed once, make it wait for $1000ms$, and re-run the inference. The graph in Figure 5.5 shows the current consumption of this configuration. Note that the average current consumed is $7.078\,(mA)$, therefore the average power consumption is $\mathbf{25.481}\,(\mathbf{mW})$, considering that the power supply of only the MCU is $3.6\,V$ [89].
The second configuration, visible in Figure 5.6, is used to test the power consumption of data transmission through BLE stack. To get the measurements we continuously send data through the BLE. In this case, the average power consumption is a bit higher than when making an inference with the network, reaching $8.852\,(mA)$ and hence $\mathbf{31.867}\,(\mathbf{mW})$.
Figure 5.7 shows the third configuration tested. In this case, the MCU is running the main while loop without anything in it, so it is set in a busy wait state. In fact, the power consumption is lower than the first two with a mean value of $6.643\,(mA)$ and $\mathbf{23.915}\,(\mathbf{mW})$. This configuration is thought to get a *baseline* consumption, therefore the other configurations where the MCU is actually performing actions must have higher power consumption than this baseline.
The last configuration, shown in Figure 5.8, is obtained by making the MCU enter a *Low-power* StandBy mode. This consists of a state where both CPU1 and CPU2 cannot execute any code, but only a standard set of peripherals are active [90]. As expected, the power consumption of this configuration is much lower than the previous ones, having a mean value of $0.284\,(mA)$ and therefore of $\mathbf{1.022}\,(\mathbf{mW})$.

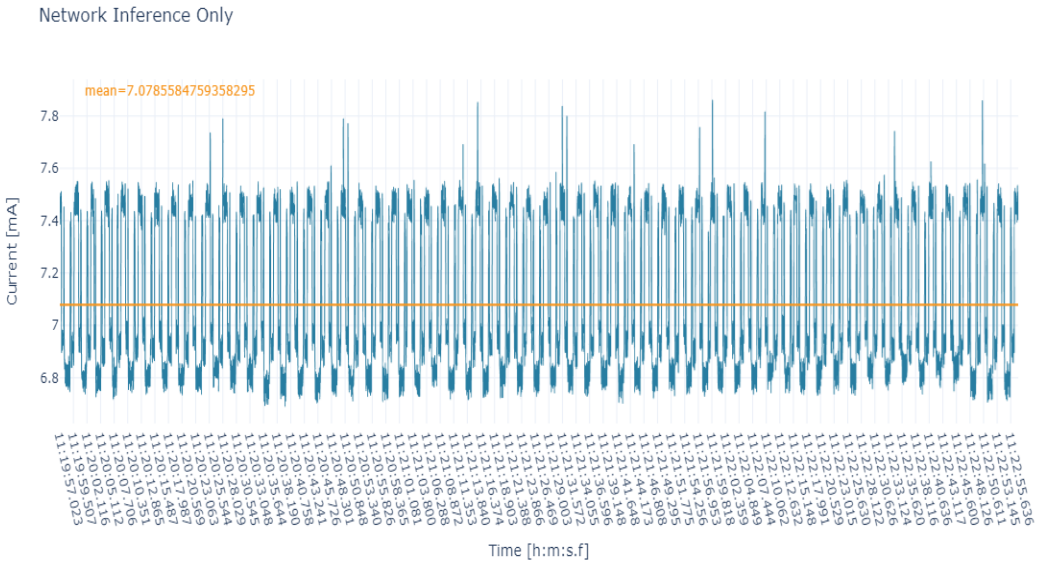


Figure 5.5: Current consumption of performing inferences with the Nucleo board.

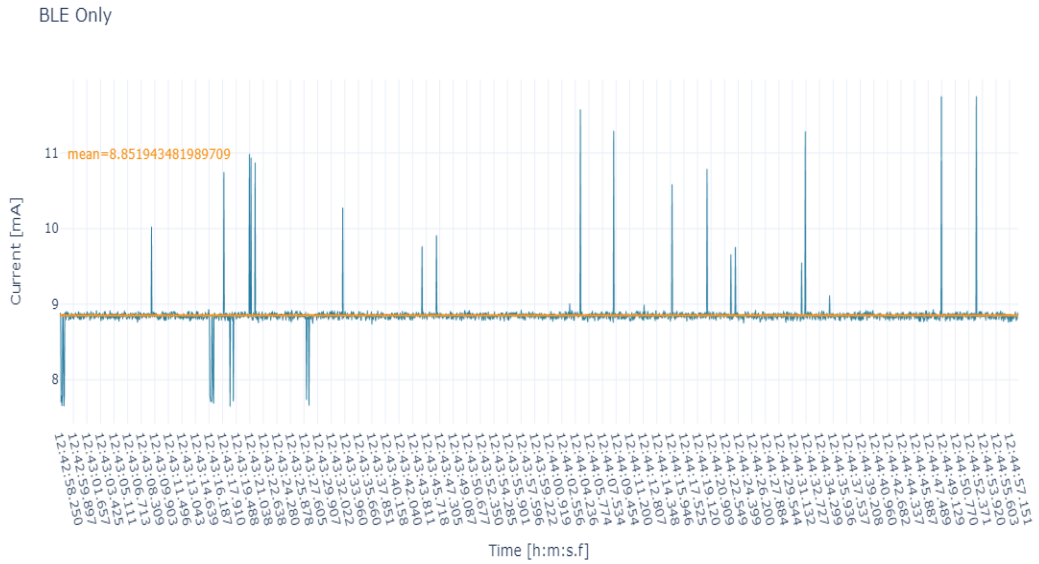


Figure 5.6: Current consumption of transferring a long list of floats numbers.

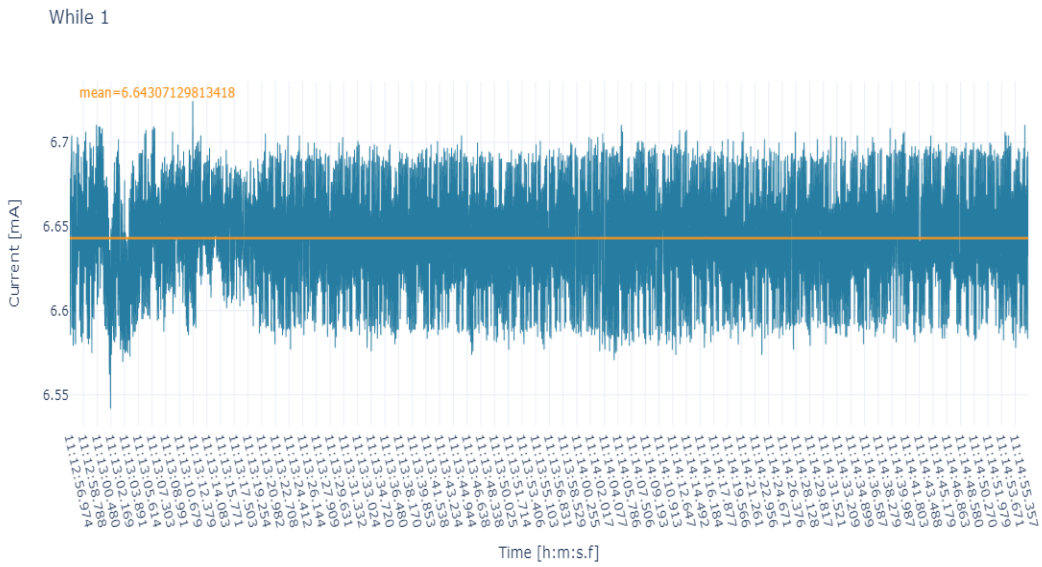


Figure 5.7: Current consumption of running the MCU in busy waiting

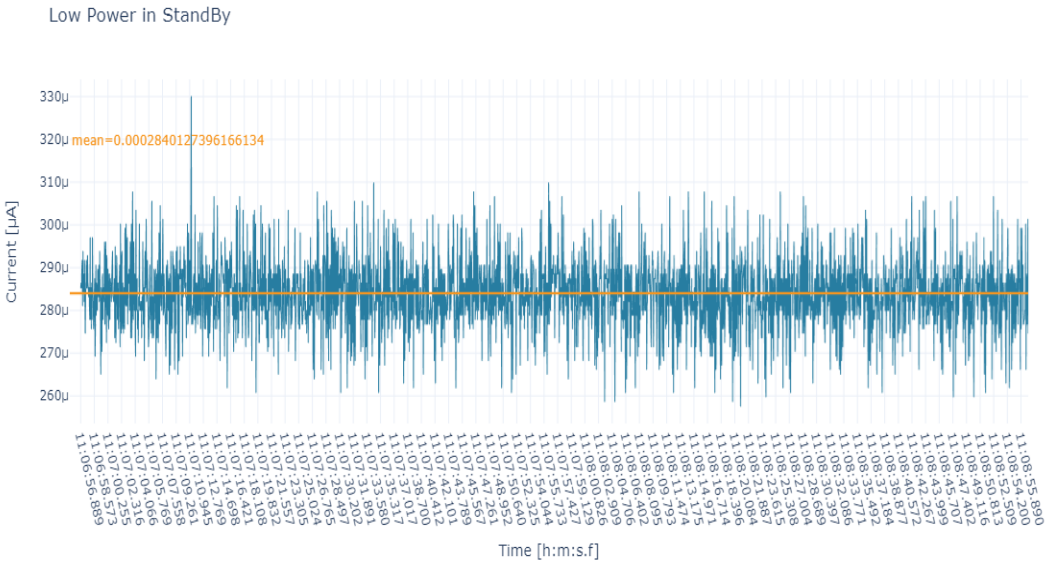


Figure 5.8: Current consumption of Nucleo board setted to Low Power Standby mode.

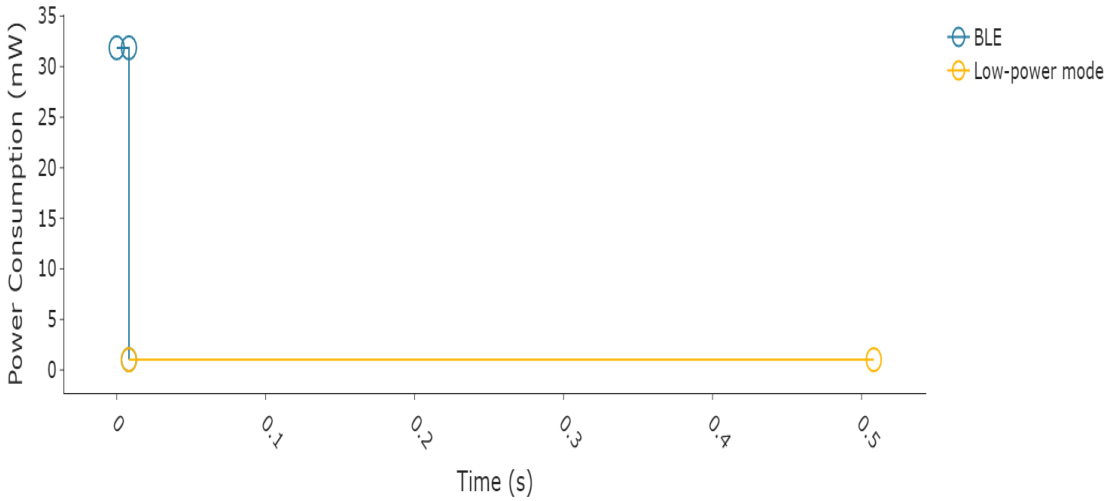


Figure 5.9: Temporal graph when transmitting 64 samples (each consisting of 4 float values) through the BLE stack and successive 0.5 s of Low-Power mode.

From these results, we can make some considerations about the effective feasibility of the architecture deployed, and in general of deep neural networks in embedded systems working with the Bluetooth LE communication protocol. Figure 5.9 shows a scenery where the inference is *not* performed directly on the MCU, but the latter is only in charge of sending the data through the BLE to the client application. In the contest of the Dalia dataset, to make an inference we need the transmission of 64 new samples (note that only for the first the network needs all the 256 elements brand new), after that the MCU is set to the Low-power mode for a short period of time.
On the other hand, in Figure 5.10 we can see the temporal graph when the inference *is* performed on the MCU. In this case, the total power consumption is given by the consumption of the network, the power consumption of sending a single prediction to the client (i.e., only a float value), and finally a small power consumption of the MCU when it is set on Low-power mode as in the previous case. It is easy to notice that the total energy consumption change with the inference time of the network deployed on the board. In particular, in Figure 5.11 is illustrated a visual representation of this phenomenon. The energy consumption of the network grows linearly with the inference time. Since the BLE consumes always a certain amount of energy when sending the same amount of data, performing the inference directly on the device is convenient only up to the intersection of the two straight lines.

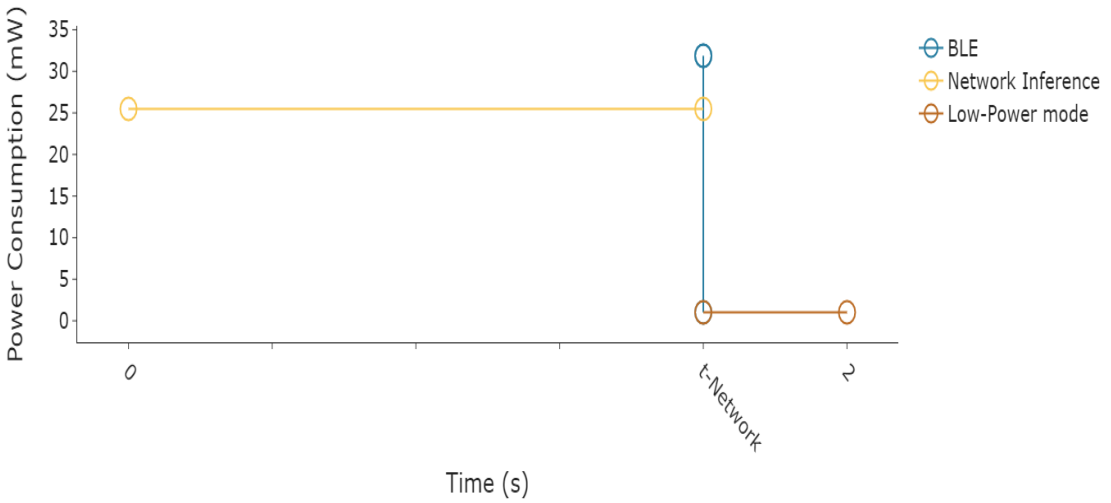


Figure 5.10: Temporal graph when performing inference on edge, transmission of the predicted value (i.e., one float) and a successive 0.5 s of Low-Power mode.

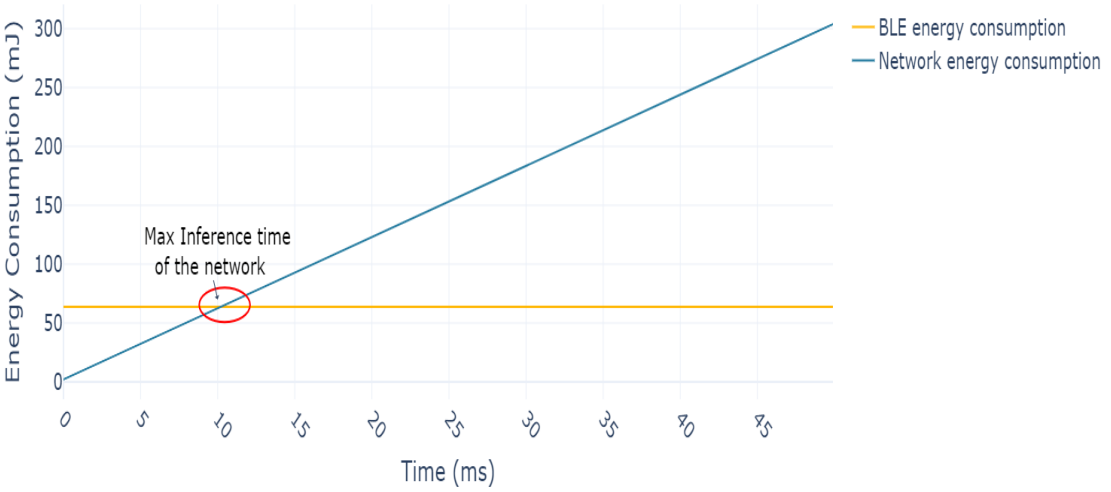


Figure 5.11: Graphic scheme to show the minimum inference time required for the network. The x-axis is cut to 0.045 (ms) for visualization's sake. The analysed energy consumption is based on the 2s time interval between two successive inferences.

In other words, to be convenient the inference time should be less than the time needed to send the data through the BLE times the ratio between the power consumption of the BLE and the power consumption of the inference. The BLE communication protocol has a data rate of $1\,MBps$, therefore the latency of transmission of a single sample is obtained by diving the amount of data by the data rate. Hence, to transfer the 4 float values (1 PPG value and 3 accelerometer values) takes around $0.128\,(ms)$. Considering how the data in segmented is the Dalia dataset, except for the first batch, we need 64 new elements to make an inference. Thus, it would take around $8.192\,(ms)$ to transfer the data. All things considered, the inference on edge results a convenient solution only when the inference time of the network is less than $10.242\,(ms)$. As said, the *Arch. 2* deployed on the MCU takes around $2\,(s)$ to make an inference, this implies that, with the current configuration, streaming all the data with the BLE would result in a more convenient solution.

Chapter 6

Conclusions

Deep Learning has proven to be very efficient in different fields. But the computational power and memory space required to run most of the networks are often too high to be integrated into embedded devices. Therefore, researchers are exploring several solutions to the challenge, among which there is the reduction of the networks' complexity and space requirements through quantization, pruning and ad hoc network designing.
The success of the integration of deep learning models into edge devices can bring many improvements in a variety of fields. One of them is surely the healthcare industry with the empowering of wearable devices to monitor patients and track both fitness and health data throughout time.
The aim of the work was the development of a Client-Server interaction based on the Bluetooth LE communication protocol, the development of a FSM (on the server-side) in charge of collecting data and transmitting it, and the integration of a TCN specifically optimized to fit into the MCU used as embedded device. Moreover, a dashboard has been integrated with the client-side application to visualize data and prediction on a second device, which in this contest was represented by a computer, but could easily be adapted to other devices, like smartphones, thanks to the portability of the Bluetooth LE technology and its presence in almost every device in use nowadays.
The TCN has been optimized to occupy $374.75KB$, and it consists of a total of $97K$ parameters and around $7M$ MAC operation. The performance of the network has been evaluated using the PPGDalia dataset, reaching a good absolute error of $2.364\,BPM$ compared to the golden HR values provided by the dataset, that were obtained from ECG signals. The time required for one inference on the STM32WB55 is about $2\,(s)$.
The analysis has shown that the average current consumed when computing the inference with the network on the STM32WB55 Nucleo board is $7.078\,(mA)$ and therefore a power consumption is $25.481\,(mW)$, while sending a window of input

data through the BLE stack consumes on average $8.852\,(mA)$ of current and $31.867\,(mW)$ of power. We derived a simple temporal graph to show that the energy consumption of the process varies according to the network inference time. In particular, taking into consideration the server-client communication through the BLE stack, we need an inference time of less than $10.242\,(ms)$ in order to have the inference on edge as the convenient choice.
Future works surely may include the further exploration of the optimization of the neural networks' architectures, especially from the point of view of inference time and energy efficiency.
Moreover, the framework might be used to sample new data and the creation of a personalized dataset. The dashboard would be used to verify if the data collection process is going well, reducing the error measurements and possible time wasted when analysing the data afterwards.
Finally, the Server-Client application is not strictly linked to the HR monitoring, by changing the kind of sensors used, it can be easily adapted to any other task like respiratory monitoring or human temperature monitoring.

Acknowledgements

The development of this work could not have been a reality without the guidance of my supervisors. I would like to sincerely thank Prof. Daniele Jahier Pagliari, Dr. Alessio Burrello and Dr. Matteo Risso for their continuous support and patient throughout the stages of experiments and writing of this thesis. Moreover, I am also grateful to them for providing me the equipment and the instrumentation needed to complete this project.

Bibliograpy

[1] D. Giansanti and G. Maccioni. «Development and testing of a wearable Integrated Thermometer sensor for skin contact thermography». In: *Medical Engineering Physics* 29.5 (2007), pp. 556–565. ISSN: 1350-4533. DOI: https://doi.org/10.1016/j.medengphy.2006.07.006. URL: https://www.sciencedirect.com/science/article/pii/S1350453306001482 (cit. on p. 1).

[2] Tobias Grosse-Puppendahl, Christian Holz, Gabe Cohn, Raphael Wimmer, Oskar Bechtold, Steve Hodges, Matthew S. Reynolds, and Joshua R. Smith. «Finding Common Ground: A Survey of Capacitive Sensing in Human-Computer Interaction». In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pp. 3293–3315. ISBN: 9781450346559. DOI: 10.1145/3025453.3025808. URL: https://doi.org/10.1145/3025453.3025808 (cit. on p. 1).

[3] Jiasi Chen and Xukan Ran. «Deep Learning With Edge Computing: A Review». In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674. DOI: 10.1109/JPROC.2019.2921977 (cit. on pp. 1, 2).

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge computing: Vision and challenges». In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on p. 1).

[5] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. DOI: 10.48550/ARXIV.1510.00149. URL: https://arxiv.org/abs/1510.00149 (cit. on p. 2).

[6] Babak Hassibi, David G Stork, and Gregory J Wolff. «Optimal brain surgeon and general network pruning». In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 293–299 (cit. on p. 2).

[7] Marian Verhelst and Bert Moons. «Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices». In: *IEEE Solid-State Circuits Magazine* 9.4 (2017), pp. 55–65 (cit. on p. 2).

[8] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*. 2016. DOI: 10.48550/ARXIV.1609.07061. URL: https://arxiv.org/abs/1609.07061 (cit. on p. 2).

[9] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. *A White Paper on Neural Network Quantization*. 2021. DOI: 10.48550/ARXIV.2106.08295. URL: https://arxiv.org/abs/2106.08295 (cit. on p. 2).

[10] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. *Pruning and Quantization for Deep Neural Network Acceleration: A Survey*. 2021. DOI: 10.48550/ARXIV.2101.09671. URL: https://arxiv.org/abs/2101.09671 (cit. on p. 2).

[11] Peter Mølgaard Sørensen, Bastian Epp, and Tobias May. «A depthwise separable convolutional neural network for keyword spotting on an embedded system». In: *EURASIP Journal on Audio, Speech, and Music Processing* 2020.1 (2020), pp. 1–14 (cit. on p. 3).

[12] Attila Reiss, Ina Indlekofer, Philip Schmidt, and Kristof Van Laerhoven. «Deep PPG: Large-Scale Heart Rate Estimation with Convolutional Neural Networks». In: *Sensors* 19.14 (2019). ISSN: 1424-8220. DOI: 10.3390/s19143079. URL: https://www.mdpi.com/1424-8220/19/14/3079 (cit. on pp. 4, 27, 28, 43, 44).

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. «Deep learning». In: *nature* 521.7553 (2015), pp. 436–444 (cit. on p. 5).

[14] Rukshan Pramoditha. *The Concept of Artificial Neurons (Perceptrons) in Neural Networks*. URL: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc (cit. on p. 5).

[15] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on pp. 6, 7).

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 (cit. on pp. 6–9, 12, 14).

[17] Sheraz Aslam, Herodotos Herodotou, Syed Muhammad Mohsin, Nadeem Javaid, Nouman Ashraf, and Shahzad Aslam. «A Survey on Deep Learning Methods for Power Load and Renewable Energy Forecasting in Smart Microgrids». In: *Renewable and Sustainable Energy Reviews* (Mar. 2021). DOI: 10.1016/j.rser.2021.110992 (cit. on p. 8).

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. 10, 12).

[19] Lukas Mosser, Olivier Dubrule, and Martin Blunt. «Stochastic Reconstruction of an Oolitic Limestone by Generative Adversarial Networks». In: *Transport in Porous Media* 125 (Oct. 2018). DOI: 10.1007/s11242-018-1039-9 (cit. on p. 11).

[20] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. «Understanding of a convolutional neural network». In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186 (cit. on p. 13).

[21] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2016. DOI: 10.48550/ARXIV.1603.07285. URL: https://arxiv.org/abs/1603.07285 (cit. on p. 13).

[22] Zhou and Chellappa. «Computation of optical flow using a neural network». In: *IEEE 1988 International Conference on Neural Networks*. 1988, 71–78 vol.2. DOI: 10.1109/ICNN.1988.23914 (cit. on p. 14).

[23] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. «Internet of Things applications: A systematic review». In: *Computer Networks* 148 (2019), pp. 241–261. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2018.12.008. URL: https://www.sciencedirect.com/science/article/pii/S1389128618305127 (cit. on p. 14).

[24] Zarlish Ashfaq, Rafia Mumtaz, Abdur Rafay, Syed Mohammad Hassan Zaidi, Hadia Saleem, Sadaf Mumtaz, Adnan Shahid, Eli De Poorter, and Ingrid Moerman. «Embedded AI-Based Digi-Healthcare». In: *Applied Sciences* 12.1 (2022), p. 519 (cit. on p. 14).

[25] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. «A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones». In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 8357–8371. DOI: 10.1109/JIOT.2019.2917066 (cit. on p. 14).

[26] M. Schlett. «Trends in embedded-microprocessor design». In: *Computer* 31.8 (1998), pp. 44–49. DOI: 10.1109/2.707616 (cit. on p. 14).

[27] Frédéric Gaillard. «Microprocessor (MPU) or Microcontroller (MCU)? What factors should you consider when selecting the right processing device for your next design». In: *URL http://ww1. microchip. com/downloads/en/DeviceDoc/MCU_vs_ MPU_Article. pdf* (2013) (cit. on p. 14).

[28] Jeevan F D'Souza, Andrew D Reed, and C Kelly Adams. «Selecting microcontrollers and development tools for undergraduate engineering capstone projects». In: *Computers in Education* 24.1 (2014) (cit. on p. 14).

[29] Jin-Yang Lai, Chiung-An Chen, Shih-Lun Chen, and Chun-Yu Su. «Implement 32-bit RISC-V Architecture Processor using Verilog HDL». In: *2021 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. 2021, pp. 1–2. DOI: 10.1109/ISPACS51563.2021.9651130 (cit. on p. 15).

[30] Ben Lutkevich. *Embedded System*. URL: https://www.techtarget.com/iotagenda/definition/embedded-system (cit. on p. 15).

[31] Frank Vahid and Tony D Givargis. *Embedded system design: a unified hardware/software introduction*. John Wiley & Sons, 2001 (cit. on pp. 15, 33).

[32] Marian Verhelst and Bert Moons. «Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices». In: *IEEE Solid-State Circuits Magazine* 9.4 (2017), pp. 55–65. DOI: 10.1109/MSSC.2017.2745818 (cit. on pp. 16, 17).

[33] Samuel Williams. «Roofline: An insightful visual performance model for floating-point programs and multicore». In: *ACM Communications* (2009) (cit. on p. 16).

[34] Ahmad Shawahna, Sadiq M. Sait, Aiman El-Maleh, and Irfan Ahmad. «FxP-QNet: A Post-Training Quantizer for the Design of Mixed Low-Precision DNNs With Dynamic Fixed-Point Representation». In: *IEEE Access* 10 (2022), pp. 30202–30231. DOI: 10.1109/ACCESS.2022.3157893 (cit. on pp. 17, 18, 20).

[35] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. DOI: 10.48550/ARXIV.2004.09602. URL: https://arxiv.org/abs/2004.09602 (cit. on pp. 18, 19).

[36] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. «Scalpel: Customizing dnn pruning to the underlying hardware parallelism». In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 548–560 (cit. on pp. 19, 21, 22).

[37] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. DOI: 10.48550/ARXIV.2103.13630. URL: https://arxiv.org/abs/2103.13630 (cit. on p. 20).

[38] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. *Structured Pruning of Deep Convolutional Neural Networks*. 2015. DOI: 10.48550/ARXIV.1512.08571. URL: https://arxiv.org/abs/1512.08571 (cit. on p. 21).

[39] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. «Proving the Lottery Ticket Hypothesis: Pruning is All You Need». In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 6682–6691. URL: https://proceedings.mlr.press/v119/malach20a.html (cit. on p. 21).

[40] Congcong Liu and Huaming Wu. «Channel pruning based on mean gradient for accelerating convolutional neural networks». In: *Signal Processing* 156 (2019), pp. 84–91 (cit. on p. 22).

[41] Tommaso Polonelli, Lukas Schulthess, Philipp Mayer, Michele Magno, and Luca Benini. «H-Watch: An Open, Connected Platform for AI-Enhanced COVID19 Infection Symptoms Monitoring and Contact Tracing». In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401362 (cit. on p. 22).

[42] Robert Avram et al. «Real-world heart rate norms in the Health eHeart study». In: *NPJ digital medicine* 2.1 (2019), pp. 1–10 (cit. on p. 24).

[43] Chenggang Yu, Zhenqiu Liu, Thomas McKenna, Andrew T Reisner, and Jaques Reifman. «A method for automatic identification of reliable heart rates calculated from ECG and PPG waveforms». In: *Journal of the American Medical Informatics Association* 13.3 (2006), pp. 309–320 (cit. on p. 24).

[44] Sanjeev Kumar et al. «A wristwatch-based wireless sensor platform for IoT health monitoring applications». In: *Sensors* 20.6 (2020), p. 1675 (cit. on p. 25).

[45] Shahid Ismail, Usman Akram, and Imran Siddiqi. «Heart rate tracking in photoplethysmography signals affected by motion artifacts: a review». In: *EURASIP Journal on Advances in Signal Processing* 2021.1 (2021), pp. 1–27 (cit. on p. 24).

[46] Dwaipayan Biswas et al. «CorNET: Deep Learning Framework for PPG-Based Heart Rate Estimation and Biometric Identification in Ambulant Environment». In: *IEEE Transactions on Biomedical Circuits and Systems* 13.2 (2019), pp. 282–291. DOI: 10.1109/TBCAS.2019.2892297 (cit. on pp. 25, 28).

[47] Zhilin Zhang, Zhouyue Pi, and Benyuan Liu. «TROIKA: A General Framework for Heart Rate Monitoring Using Wrist-Type Photoplethysmographic Signals During Intensive Physical Exercise». In: *IEEE Transactions on Biomedical Engineering* 62.2 (Feb. 2015), pp. 522–531. DOI: 10.1109/tbme.2014.2359372. URL: https://doi.org/10.1109%2Ftbme.2014.2359372 (cit. on p. 25).

[48] Zhilin Zhang. «Photoplethysmography-Based Heart Rate Monitoring in Physical Activities via Joint Sparse Spectrum Reconstruction». In: *IEEE Transactions on Biomedical Engineering* 62.8 (2015), pp. 1902–1910. DOI: 10.1109/TBME.2015.2406332 (cit. on p. 26).

[49] Zhilin Zhang. «Heart rate monitoring from wrist-type photoplethysmographic (PPG) signals during intensive physical exercise». In: *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE. 2014, pp. 698–702 (cit. on p. 26).

[50] Boreom Lee, Jonghee Han, Hyun Jae Baek, Jae Hyuk Shin, Kwang Suk Park, and Won Jin Yi. «Improved elimination of motion artifacts from a photoplethysmographic signal using a Kalman smoother with simultaneous accelerometry». In: *Physiological measurement* 31.12 (2010), p. 1585 (cit. on p. 26).

[51] Mahdi Boloursaz Mashhadi, Ehsan Asadi, Mohsen Eskandari, Shahrzad Kiani, and Farokh Marvasti. «Heart Rate Tracking using Wrist-Type Photoplethysmographic (PPG) Signals during Physical Exercise with Simultaneous Accelerometry». In: *IEEE Signal Processing Letters* 23.2 (2016), pp. 227–231. DOI: 10.1109/LSP.2015.2509868 (cit. on p. 26).

[52] Seyed Salehizadeh, Duy Dao, Jeffrey Bolkhovsky, Chae Ho Cho, Yitzhak Mendelson, and Ki Chon. «A Novel Time-Varying Spectral Filtering Algorithm for Reconstruction of Motion Artifact Corrupted Heart Rate Signals During Intense Physical Activities Using a Wearable Photoplethysmogram Sensor». In: *Sensors* 16 (Dec. 2015), p. 10. DOI: 10.3390/s16010010 (cit. on p. 26).

[53] Heewon Chung, Hooseok Lee, and Jinseok Lee. «Finite state machine framework for instantaneous heart rate validation using wearable photoplethysmography during intensive exercise». In: *IEEE journal of biomedical and health informatics* 23.4 (2018), pp. 1595–1606 (cit. on p. 27).

[54] Menglian Zhou and Nandakumar Selvaraj. «Heart rate monitoring using sparse spectral curve tracing». In: *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. IEEE. 2020, pp. 5347–5352 (cit. on p. 27).

[55] Nicholas Huang and Nandakumar Selvaraj. «Robust ppg-based ambulatory heart rate tracking algorithm». In: *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. IEEE. 2020, pp. 5929–5934 (cit. on p. 27).

[56] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. «Fast and accurate deep network learning by exponential linear units (elus)». In: *arXiv preprint arXiv:1511.07289* (2015) (cit. on p. 28).

[57] Heewon Chung, Hoon Ko, Hooseok Lee, and Jinseok Lee. «Deep Learning for Heart Rate Estimation From Reflectance Photoplethysmography With Acceleration Power Spectrum and Acceleration Intensity». In: *IEEE Access* 8 (2020), pp. 63390–63402. DOI: 10.1109/ACCESS.2020.2981956 (cit. on p. 28).

[58] Alessio Burrello, Daniele Jahier Pagliari, Matteo Risso, Simone Benatti, Enrico Macii, Luca Benini, and Massimo Poncino. «Q-ppg: energy-efficient ppg-based heart rate monitoring on wearable devices». In: *IEEE Transactions on Biomedical Circuits and Systems* 15.6 (2021), pp. 1196–1209 (cit. on pp. 29, 31, 47).

[59] Matteo Risso, Alessio Burrello, Daniele Jahier Pagliari, Simone Benatti, Enrico Macii, Luca Benini, and Massimo Pontino. «Robust and Energy-Efficient PPG-Based Heart-Rate Monitoring». In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401282 (cit. on pp. 29–31, 47).

[60] Alessio Burrello, Daniele Jahier Pagliari, Pierangelo Maria Rapa, Matilde Semilia, Matteo Risso, Tommaso Polonelli, Massimo Poncino, Luca Benini, and Simone Benatti. «Embedding temporal convolutional networks for energy-efficient PPG-based heart rate monitoring». In: *ACM Transactions on Computing for Healthcare (HEALTH)* 3.2 (2022), pp. 1–25 (cit. on pp. 30, 47–49).

[61] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. «Morphnet: Fast & simple resource-constrained structure learning of deep networks». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 1586–1595 (cit. on p. 30).

[62] RS. *Scheda di espansione Nucleo Pack STMicroelectronics, CPU ARM-Cortex-M0*. URL: https://it.rs-online.com/web/p/strumenti-di-sviluppo-per-microcontrollori/1830086 (cit. on p. 33).

[63] ETH-PBL. *H-Watch*. URL: https://github.com/ETH-PBL/H-Watch (cit. on p. 33).

[64] *STM32WB55xx - Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5.2 and 802.15.4 radio solution*. Rev. 11. STMicroelectronics. Apr. 2021 (cit. on p. 32).

[65] Surachai Panich. «A mobile robot with a inter-integrated circuit system». In: *2008 10th International Conference on Control, Automation, Robotics and Vision*. 2008, pp. 2010–2014. DOI: 10.1109/ICARCV.2008.4795839 (cit. on p. 33).

[66] Piyu Dhaker. «Introduction to SPI interface». In: *Analog Dialogue* 52.3 (2018), pp. 49–53 (cit. on p. 33).

[67] *MAX30101 - High-Sensitivity Pulse Oximeter and Heart-Rate Sensor for Wearable Health*. Rev. 3. Maxime Integrated. June 2020 (cit. on p. 34).

[68] *LSM6DSM - iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope*. Rev. 7. STMicroelectronics. Sept. 2017 (cit. on p. 34).

[69] STMicroelectronics. *Integrated Development Environment for STM32*. URL: https://www.st.com/en/development-tools/stm32cubeide.html (cit. on p. 34).

[70] STMicroelectronics. *STM32Cube initialization code generator*. URL: https://www.st.com/en/development-tools/stm32cubemx.html (cit. on p. 35).

[71] Jacopo Tosi, Fabrizio Taffoni, Marco Santacatterina, Roberto Sannino, and Domenico Formica. «Performance Evaluation of Bluetooth Low Energy: A Systematic Review». In: *Sensors* 17.12 (2017). ISSN: 1424-8220. DOI: 10.3390/s17122898. URL: https://www.mdpi.com/1424-8220/17/12/2898 (cit. on p. 36).

[72] STMicroelectronics. *STM32CubeProgrammer software for all STM32*. URL: https://www.st.com/en/development-tools/stm32cubeprog.html (cit. on p. 38).

[73] *AN5289 - Application note - Building wireless applications with STM32WB Series microcontrollers*. Rev. 6. STMicroelectronics. Dec. 2021 (cit. on pp. 39, 40).

[74] Bluetooth SIG. *Bluetooth*. URL: https://www.bluetooth.com/ (cit. on p. 39).

[75] *bleak Documentation - Release 0.14.3*. Rev. 6. Henrik Blidh. Apr. 2022. URL: https://github.com/hbldh/bleak (cit. on p. 41).

[76] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. «A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems». In: *CoRR* abs/1906.00367 (2019). arXiv: 1906.00367. URL: http://arxiv.org/abs/1906.00367 (cit. on p. 41).

[77] Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. «Asynchronous programming, analysis and testing with state machines». In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pp. 154–164 (cit. on p. 41).

[78] Python. *asyncio — Asynchronous I/O*. URL: `https://docs.python.org/3/library/asyncio.html` (cit. on p. 41).

[79] STMicroelectronics. *https://www.st.com/en/embedded-software/x-cube-ai.html*. URL: `https://www.st.com/en/embedded-software/x-cube-ai.html` (cit. on p. 42).

[80] *UM2526 - User manual - Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)*. Rev. 8. STMicroelectronics. Jan. 2022 (cit. on p. 43).

[81] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. «Temporal convolutional networks for action segmentation and detection». In: *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 156–165 (cit. on p. 47).

[82] Tensorflow. *Tensorflow*. URL: `https://www.tensorflow.org/` (cit. on p. 47).

[83] Marcello Zanghieri, Simone Benatti, Alessio Burrello, Victor Kartsch, Francesco Conti, and Luca Benini. «Robust Real-Time Embedded EMG Recognition Framework Using Temporal Convolutional Networks on a Multicore IoT Processor». In: *IEEE Transactions on Biomedical Circuits and Systems* 14.2 (2020), pp. 244–256. DOI: `10.1109/TBCAS.2019.2959160` (cit. on pp. 47, 48).

[84] Plotly. *Dash Enterprise*. URL: `https://plotly.com/dash/` (cit. on p. 49).

[85] Plotly. *Introduction to Dash* (cit. on pp. 49, 50).

[86] TensorFlow. *Quantization aware training comprehensive guide*. URL: `https://www.tensorflow.org/model_optimization/guide/quantization/training_comprehensive_guide.md#experiment_with_quantization` (cit. on p. 53).

[87] Jeffrey J Goldberger, Nils P Johnson, Haris Subacius, Jason Ng, and Philip Greenland. «Comparison of the physiologic and prognostic implications of the heart rate versus the RR interval». In: *Heart Rhythm* 11.11 (2014), pp. 1925–1933 (cit. on p. 55).

[88] Keysight. *34401A Digital Multimeter, 6½ Digit*. URL: `https://www.keysight.com/us/en/product/34401A/digital-multimeter-6-digit.html` (cit. on p. 57).

[89] *STM32WB55xx STM32WB35xx- Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5.2 and 802.15.4 radio solution - Datasheet production data*. Rev. 12. STMicroelectronics. Jan. 2022 (cit. on p. 57).

[90] *STM32 power mode examples*. Rev. 3. STMicroelectronics. Aug. 2019 (cit. on p. 57).