POLITECNICO DI TORINO

Master degree course in Ingegneria Elettronica
(Electronic Engineering)

Master Degree Thesis

Neural Architecture Search Techniques for the Optimized Deployment of Temporal Convolutional Networks at the Edge

Supervisors
Doc. Daniele Jahier Pagliari
Doc. Alessio Burrello

Candidate
Matteo Risso
matricola: 265419

Academic Year 2020-2021

Summary

For many years Recurrent Neural Networks (RNNs) achieved state-of-the-art (SoTA) results in time series analysis, but their large computational complexity makes them ill-suited to be deployed on microcontrollers and, in general, on other resource-constrained edge devices. A more viable alternative, towards the efficient deployment of time series related tasks is represented by Temporal Convolutional Networks (TCNs), a particular class of Convolutional Neural Networks (CNNs), achieving comparable results to SoTA RNNs architectures. TCNs offer many advantages from a computational standpoint, resulting in a more hardware-friendly alternative to RNNs. Nevertheless, the optimized deployment of a TCN on a microcontroller-based edge device still requires a careful and time-consuming hand-tuning of the model's hyper-parameters. This tedious process is necessary to achieve a good trade-off among inference accuracy and computational complexity (total number of operations and memory footprint). Nonetheless, due to the extremely large design-space of such architectures, usually this hand-tuning process leads to sub-optimal solutions. This thesis tackles the problem using Neural Architecture Search (NAS), i.e., the automatic tuning of hyper-parameters by means of an algorithm. In this work the usage of NAS algorithms is explored in the direction of searching for TCN architectures with hardware-friendly features, i.e., a small memory footprint and/or reduced number of Floating Point OPerations (FLOPs). In particular, two different low-complexity NAS approaches, i.e., MorphNet and PIT, are applied to a seed TCN, in order to train the network on its task and, jointly, optimize the target hyper-parameters that define the architecture, with respect to a specific metric (e.g., the FLOPs). MorphNet is an existing NAS approach present in literature, which is here combined with PIT, a novel light-weight NAS designed during this thesis work. The two methods are applied both independently on the seed network and also jointly exploring different setups. This allows to

explore the orthogonality of the two NASes, leading to a complete workflow for the optimization of TCNs. In particular, the proposed workflow allows to obtain both SoTA results on the considered time series analysis task and also to compress the seed architecture as much as 99.6%.

Acknowledgements

Anyone knowing me well is aware how little I am inclined to let me go publicly. Nevertheless, I am sure that spending few lines for all the people that were there for me in this long journey it is the least I could do.

To my supervisors, Daniele Jahier Pagliari and Alessio Burrello, for helping me with kindness and passion in whatever difficulties I encountered. Moreover, I want to infinitely thank you for opening me up a way to my future. I respect deeply your work and I hope to be worthy, too.

To my parents, Angelo and Magda, for supporting every choice I have made. For never held back a solid support. I thank you for always being there, even when I want to be left alone.

To my grandparents, Marta, Renzo, Rosanna and Piero. Even if I have sinned to prove it you are and always has been part of who I am.

To Dante, for never leave me alone (literally).

To Betty, for sharing with me every aspect of this long journey. You support me when I was overwhelmed, bringing levity where I put weight. Thank you for teaching me to take one step at a time. You made me feel at home, always.

To all the people unjustly neglected here, but that have contributed, directly or indirectly, to this amazing accomplishment.

Sincerely, thank you.

Table of Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence.

ANN Artificial Neural Network.

CNN Convolutional Neural Network.

CV Computer Vision.

DAG Direct Acyclic Graph.

DL Deep Learning.

DMA Direct Memory Access.

DNAS Differentiable Neural Architecture Search.

DNN Deep Neural Network.

ECG ElectroCardioGram.

FLOP Floating Point Operation.

FPU Floating Point Unit.

GAN Generative Adversarial Network.

GPU Graphic Processing Unit.

MA Motion Artifact.

MAE Mean Absolute Error.

MCU MicroController Unit.

ML Machine Learning.

MLP MultiLayer Perceptron.

MSE Mean Squared Error.

NAS Neural Architecture Search.

NLP Natural Language Processing.

NN Neural Network.

PPG PhotoPlethysmoGraphy.

PPG PhotoPlethysmoGram.

PULP Parallel Ultra Low Power.

RNN Recurrent Neural Network.

SGD Stochastic Gradient Descent.

SoC System on Chip.

TCN Temporal Convolutional Network.

Chapter 1

Introduction

Throughout history several scientific breakthroughs and innovations have changed the perception of the world where everybody of us live. Mainly this happens slowly, through several years, but sometimes this change is very abrupt. A very evocative concept, used by historians to denote this type of events is summarized by the two following words : *Copernican revolution*, which recalls the transition from an geocentric view of the universe to the well-known heliocentric one. Is certainly very difficult to understand when a *revolution* of this type is happening in real-time, but some philosophers [1] bravely indicates our days as revolutionary.

This announced revolution is strictly related to information technologies and to the very inflated concept of Artificial Intelligence (AI), which absolutely requires few words to be properly framed. This work is mainly related to Deep Learning (DL) algorithms, which are a subset of the wider discipline of Machine Learning, in turn usually seen as an autonomous branch of the aforementioned AI.

From a technical perspective Machine Learning techniques represent an effective way to realize systems that are able to learn from data and experiences, usually requiring a strong intervention in the form of tuning and data preparation of the human programming them [2]. DL pushes forward the concept of learning making it as autonomous as possible, which mainly means reducing the need of human intervention in tuning the algorithm and nearly removing the data preparation step, enabling therefore learning from raw data, as human and other living beings do.

In the last years DL algorithms have started to provide state of the art results in different areas *e.g.*, Computer Vision (CV), Natural Language

Processing (NLP) and time series analysis. For each different task various DL solutions exist, mainly in the form of different flavours of Artificial Neural Networks (ANNs), or simply Neural Networks (NNs), which are, as suggested by the name, brain-inspired Machine Learning models. In particular, Convolutional Neural Networks (CNNs) and its variants are well suited for CV problems, while Recurrent Neural Networks (RNNs) and its evolutions provide best results in NLP and when dealing with time series [3].

Day after day NNs are becoming more complex to achieve better performances, and this linked with the huge amount of data to be managed at training time, makes mandatory the exploitation of a huge computing power. The training phase of ANNs is today possible thanks to Graphic Processing Units (GPUs), and is almost confined to that type of hardware usually exploited on the cloud with high performing clusters. A different reasoning can be, instead, performed for the inference phase of NNs, which is the phase when the trained model deals with real-world data and tries to accomplish its task. Inference can always be performed in the cloud, but today a large effort is devoted by industries and researchers to bring inference from cloud to the edge [4], which means running trained DL architectures on resource constrained devices like MicroController Units (MCUs) and IoT-nodes. In this way, data that are collected by this type of devices can be directly processed in place, with several benefits. One of such benefits is an increased responsivity of the system. In fact, the latency required to perform computations directly on the device is much lower compared to the time requested to send data, wait for the elaboration on the cloud and finally receive results of the elaboration. Another potential advantage is an improvement in privacy, since data stored in the edge-device never leave it. Lastly, the energy consumption required for the inference may be lower compared to the one required for the wireless transmissions, resulting in a more efficient computation [2].

If edge-computing, from a side, presents significant advantages from the other it opens several challenges, first of all the need to embed large models in resource-constrained devices. An ANN model "as is" presents, usually, a too large memory footprint to be fitted on a general purpose MCU making it impossible to perform inference on the edge. For this reason various solutions and techniques have been proposed in literature [5], [6] [7], [8], in order to compress ANNs models, which mainly rely on their intrinsic high-level of redundancy. ANNs present redundancy at the level of data representation : often is possible to pass from a floating-point representation of data to a

simpler fixed-point representation with lower number of bits, in a process known as quantization [9]. Another source of redundancy is realized by the network itself because usually ANN are over-parametrized. Thanks to pruning techniques is possible to identify redundant (and not useful) elements and prune them [5], [6], [7]. Both these techniques, if applied in an intelligent way, are able to shrink very efficiently the networks, without tampering too much their performance, enabling DL on the edge. Another approach that allows ANN to be executed on low-resources hardware, is Neural Architecture Search (NAS) [10], [11], [12], [13]. NAS techniques aim is to find the best DL architecture, from the point of view of a certain metric (e.g., the size), to accomplish a certain task. The search performed by NAS algorithms is usually based on Machine Learning techniques. Is then possible to apply NAS to find architectures with a certain size constraint. Moreover a reduction in size is usually followed by a reduction in the number of operations (MACs) to be performed in a single iteration of the algorithm, which is translated in a lower latency and lower energy consumption.

This work deals with the problem of time series analysis on edge devices. The state of the art in time series is represented, today, by RNNs and its evolutions, which are usually not so well suited to be executed on general-purpose MCUs. For this reason we investigate here the use of TCNs [14], [15], a special class of CNN, for the task of time series analysis on edge. In particular, we investigate the optimization and implementation, for an embedded device, of an architecture based on TEMPONet, a TCN presented in [16], that gives excellent results with bio-signals. The main contributions of this work can be identified in a framework of NAS techniques for the joint optimization and compression of a seed ANN, like TEMPONet. In particular, the problem is tackled from two different sides :

(i) By means of the NAS approach proposed in [12], which tries to optimize, during the training, the number of output filters of each convolutional and fully-connected layer. The effectiveness of the method is shown and demonstrated for TCN architectures, which are not considered in [12].

(ii) Even if the literature includes plenty of NAS approaches that are general, and thus allow whatever network's parameters to be specified, like in [11], a specific, fast and light-weight method for learning dilations factors, the most characteristic hyper-parameter of TCNs, is still missing. Here a novel NAS algorithm is proposed, called Pruning In Time (PIT), that allows the learning of dilation factors.

Moreover the orthogonality of the two methods is also explored, in order to understand if the two approaches can be combined together to find even better architectures, with respect to to the standalone NAS algorithms. As a benchmark for the architectures and techniques analyzed and developed the Dalia dataset [17] is exploited. It represents the largest publicly available dataset of PhotoPlethysmoGram (PPG) signals, used with the aim of accurate heart-rate estimation in wearable embedded devices.

As target device, in the whole work, the GAP8 SoC is considered [18].

The rest of this thesis is structured as follows. Chapter 2 presents the required background on DL algorithms and the description of the task and the dataset used as benchmark. Chapter 3 proposes a summary of the main published works in the field of NAS. Chapter 4 represents the core section of the whole work, presenting the developed algorithms and techniques. Finally chapter 5 summarizes the obtained results on the target task.

Chapter 2

Background

2.1 Overview

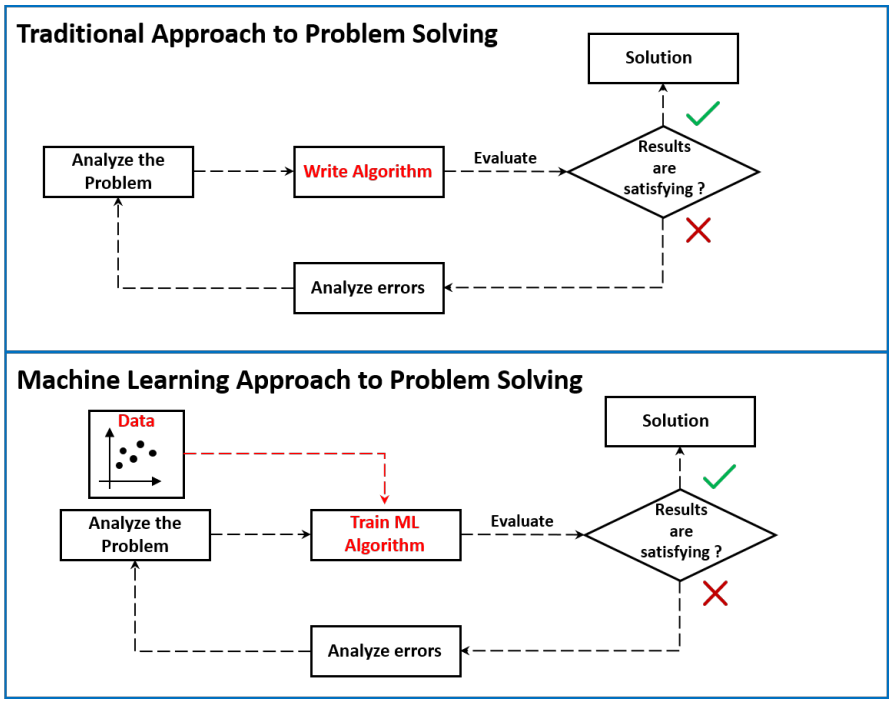

Figure 2.1: Comparison between the traditional paradigm to problem solving (uppermost scheme) and the Machine Learning one (lowermost scheme) [19]

One of the first problems that need to be faced when dealing with Machine

Learning is related to its own definition, because an univocal and generally accepted one does not exist. One good candidate might be :

Machine Learning is the science (and art) of programming computers so they can learn from data. -Aurèlien Gèron, 2019 [19].

A more technical one is presented in [20] :

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. -Tom Mitchell, 1997.

This second definition gives a great hint of what a Machine Learning algorithm is and how it works. It is highlighted that a Machine Learning system learns from experience, providing a new paradigm with respect to the traditional one, where the algorithm simply encodes some prior knowledge of the programmer. This comparison between traditional and Machine Learning approaches is summarized in figure 2.1.

Deep Learning, instead, as briefly mentioned in chapter 1 is a subset of Machine Learning, therefore the same considerations presented before hold, but DL is somehow an evolution of the plain Machine Learning approach, that trades more complicated (deep) models in exchange for a lesser need of domain expertise in their setup.

Before the advent of DL, the development and the diffusion of Machine Learning methods were hampered by a strong need of *feature engineering*, where the term *feature* denotes each type of information that is provided to the algorithm as an input. A general approach to feature engineering does not exist, thus each task needs to be studied and analyzed separately requiring a strong know-how and lot of time.

For example, if our Machine Learning model's aim is to classify different species of Iris flowers, possible features to be considered are the width or the length of the petals [21]. In the proposed example is quite easy to identify candidates features, but this is not always the case : usually a laborious trial-and-error work is required to identify the set of useful features that allows to accomplish the task in the best way. Another example, that shows this other side, might be a face-recognition task starting from photographs. A human being knows that faces generally include two eyes, one mouth and so on and we can think to use this knowledge as features, but it is extremely

difficult to describe simply what an eye (or a mouth) is in terms of raw image pixels.
This problem is solved in DL by means of the so called *representation learning* where the Machine Learning approach is used not only to learn the mapping between features and the output but also to learn representation of features starting from raw inputs (e.g., an image) [22].
Deep Learning algorithms are, as said, more complex than Machine Learning counterparts but, thanks to representation learning, they minimize the data preparation and feature engineering phase solving therefore the strongest drawback of Machine Learning models. In DL models the raw input data are transformed by means of stack of functions, called layers in the *DL jargon*, where each function is composed of many simpler functions. Each layer transforms the input data in representations with an increasing level of abstraction.

A plethora of different Machine Learning and DL algorithms exist and different ways of classifying them are possible. Probably the most common classification criteria is related to the characteristics of the learning process through which the algorithm is able to tune itself in order to accomplish its task. Three main learning approaches exist : *supervised learning*, *unsupervised learning* and *reinforcement learning*. The most common is the first one, which is also the only learning technique considered in this work.

In *supervised learning* the algorithm is fed with a training set that includes both the inputs and the desired outputs, usually denoted as *labels*. The labels are used to make a comparison between them and the effective output, the aim of the learning process is tuning the algorithm in a way that the difference between effective outputs and labels is minimized. Typical supervised learning tasks are *regression* and *classification*.

Unsupervised learning is characterized by unlabeled data, therefore the model tries to learn without a gold-reference. This type of approach is usually exploited in *clustering* and *data reduction* and *visualization*.

The last type of learning, here briefly presented is *reinforcement learning*, where the system is denoted as an *agent* which learns during it execution, having rewards and penalties depending on the actions performed.

The most common class of DL algorithms are Deep Neural Networks (DNNs) (or simply NNs). They are characterized by a stacked and linked modular architecture, usually trained with a supervised approach. Also in the contest of DNNs various classifications exist, which are mainly related to the type of layers involved and their connections. Usually each ANNs'

family has been developed for addressing a specific set of task. The main architectures are :

- **Multi Layer Perceptron (MLP)**, one of the first ANN model proposed. It is made of feed-forward stacked layer of densely connected elements. Usually they are exploited to realize simple classifiers.
- **Convolutional Neural Network (CNN)**, evolution of MLP models based on layers performing discrete convolution of input matrices with learnable filters. They represent the state of the art in CV tasks.
- **Recurrent Neural Network (RNN)**, presents loops and a memory of previous computations. They represent the state-of-art in NLP and sequences analysis tasks.

2.2 Neuron

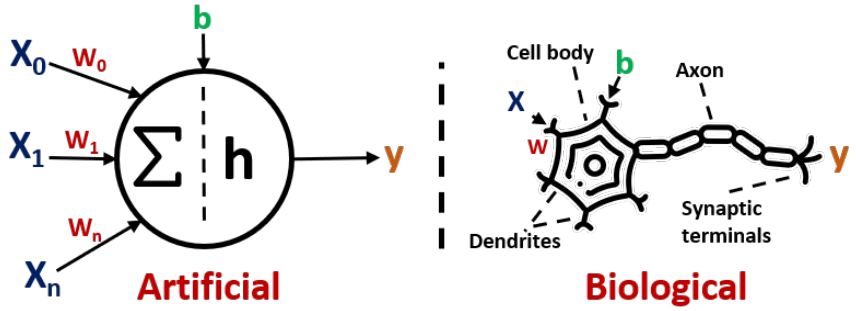

Figure 2.2: Artificial (left) and biological (right) neurons.

The basic functional unit of neural networks, from which the adjective *neural* is inherited is the *neuron*. The DL neuron, namely the artificial neuron, is a simple elementary unit that performs a weighted sum of its inputs followed by a non-linear function. The formula describing its behavior is :

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

Where x_i are the inputs of the neuron, multiplied to the respective weights w_i, b is a bias term and $h()$ denotes a non-linear function, usually called *activation function*, which is applied to the weighted sum.

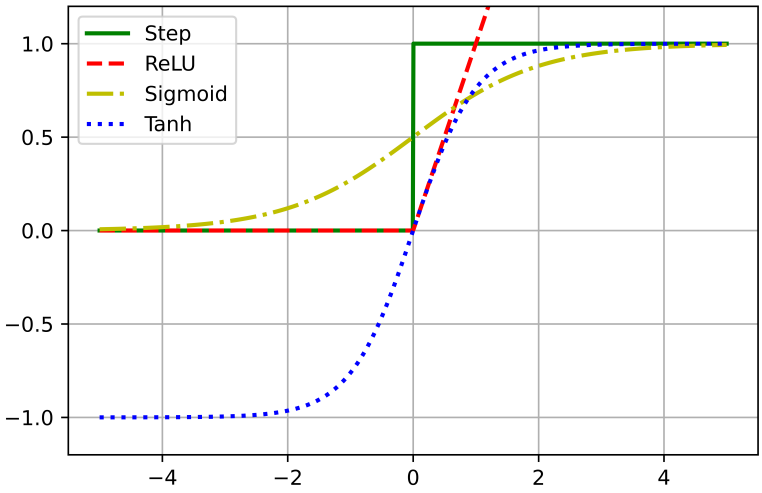

Figure 2.3: Most common activation functions.

2.2.1 Most Common Activation Functions

In figure 2.3 the most common activations function that can be found in NNs are represented [19]. Some brief notions about them are here provided :

- **Step Function** (ref : green curve-figure 2.3): represents the first activation function proposed in ANNs, in the context of the *perceptron* [23]. Its analytical form is :

$$h(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{2.2}$$

  This function binarize the output of the neuron, deciding if its output can be propagated or not, limiting then the information provided to other elements in the network. Moreover the step function is not derivable in the point $x = 0$, this complicates the training of the system through the powerful *backpropagation* algorithm, presented in 2.4, which requires almost-everywhere derivable functions.

- **Sigmoid Function** (ref : yellow curve-figure 2.3): provides an output in the interval (0,1) as the step function, but it is fully-derivable in its

domain. It is described by :

$$h(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Thanks to its smooth behavior it can be used with *backpropagation*, but it can suffer of the *vanishing* gradients problem, when x is too high or too small, having that the sigmoid is only sensitive to its input when it is near 0. Sigmoid units can be used as output units, to predict the confidence, in terms of probability, that a binary variable is 1 [22].

- **Hyperbolic Tangent Function (tanh)** (ref : blue curve-figure 2.3): presents S-shaped, smooth behavior like sigmoid, but differs from it about the output range, which ranges from -1 to 1. Its equation can be written as :

$$h(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

Usually *tanh* performs better than sigmoid function during training, providing a much more stable gradient and thus a faster convergence. This is mainly related to the fact that the hyperbolic-tangent function is more similar to an identity function [22], because $tanh(0) = 0$ while $sigmoid(0) = \frac{1}{2}$. This results in a neuron which is linear in the neighborhood of 0, providing a much more simpler training in that region.

- **Rectified Linear Unit (ReLU)** (ref : red curve-figure 2.3): represents today (with some variants) the "standard" activation for ANNs. Its equation is :

$$h(x) = max(0, x) \tag{2.5}$$

It is easy to compute and solve the saturation problems for large values of x, avoiding vanishing gradients. Moreover thanks to its linearity the training is much faster. The main drawback of ReLU is that it blocks learning from input examples that produces a negative output, for this reason some evolutions has been proposed that sometimes perform better. Some of this ReLU variants (not analyzed here) are **Leaky ReLU** and **Exponential Linear Unit (ELU)**.

2.2.2 Artifical and Biological Neurons

Although the *neuron* term is very catchy and attractive, the biological analogy with brain cells is very thin. In figure 2.2 an artificial and a biological neuron are schematized.

Biological neurons are made of several parts and are much more complex, both in terms of structure and function, with respect to to artificial ones. The main parts of a "living"-neuron are a *cell body* which contains the nucleus and the cell's essential elements which are somehow common to every human cell. What characterize neurons is the fact that they build a densely interconnected structure : they catch biological signals, as inputs, from other neurons thanks to *dendrites* which are branching extensions connected to the cell body. Outputs travel through a very long extension called *axon*, that splits at its end in many *synaptic terminals* or *synapses* which are connected to the dendrites of other neurons. Electrical signals that reach synapses stimulate the release of chemical signals indicated as *neurotransmitters*. This are next collected by dendrites of other cells, which would fire (i.e., produce and propagate another signal) if the amount neurotransmitters is sufficient [19].

Artificial neurons somehow resemble to the biological counterparts, mainly in the sense that they form a densely connected structure of similar units collecting inputs of other units and broadcasting outputs to some others. Also the concept of activation function is bio-inspired, mimicking mainly the activation mechanism implemented with neurotransmitters, but at the same time is also misleading with the biological analogy, because real-neurons seem to implement an S-shaped activation function, which is not the best performing choice in ANNs, where ReLU generally is better.

This was only an extremely simple (and simplistic) overview of what neurons really are, but it is important to know that neurons of Neural Networks are only a remotely inspired model of what a real neuron is.

2.3 Layers of Neurons

The single neuron presented in section 2.2 is only capable to perform linear combinations (followed by a non-linear function) of the input values, thus it can be used only for very simple binary classification or regression tasks. Taking more neurons of this type, juxtaposing them and broadcasting inputs to every neuron a so-called layer of neurons is realized. This structure is able

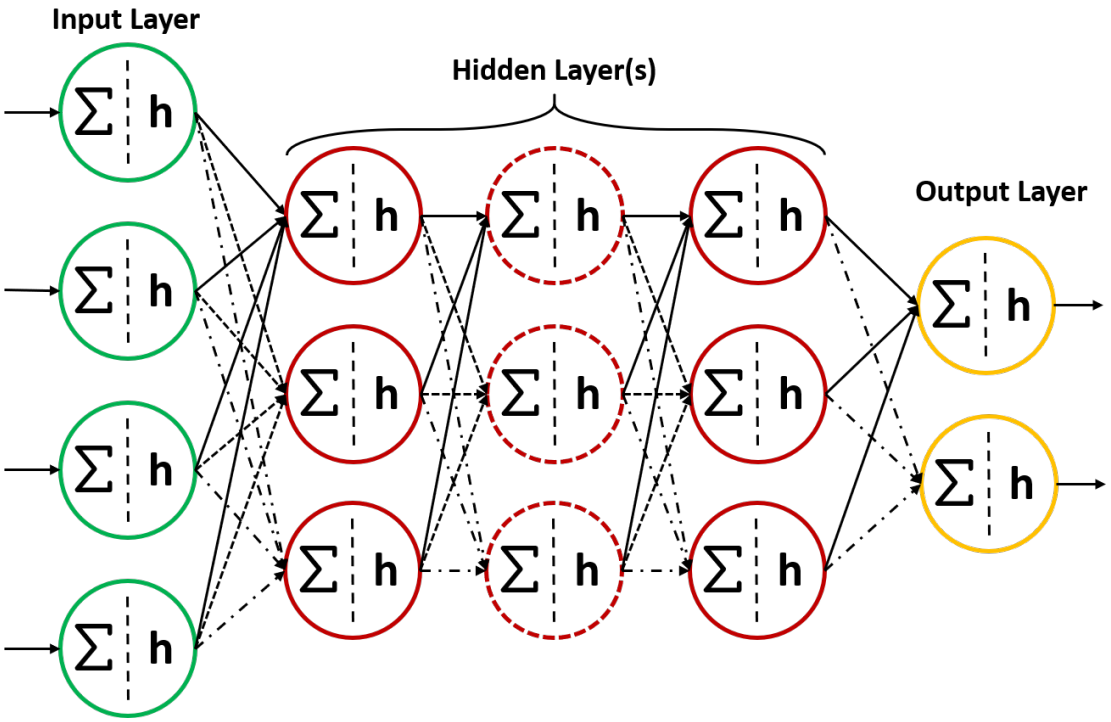

Figure 2.4: Fully Connected layers of neurons (MLP).

to perform tasks like multi-output classification. A single layer of neurons presents several problems, in particular it fails in solving some classical problems like the *XOR* classification problem [24]. This and other problems are solved by stacking multiple layers of this type, in an architecture which is known as MLP [19].

The simplest MLP is composed by three layers : *input*, *hidden* and *output* layers. An example of MLP is prsented in figure 2.4.

The input layer collects input data that are used to perform computations inside the network. Usually an additional default-input is provided, which takes the value of 1 and is used to take into account the bias term.

The hidden layer is the one that follows the input layer. It produces an intermediate output which is not the final one. It is the "core" of the network, its purpose is learning useful relations used to extract features from the inputs, which are then fed to the output layer.

The output layer provides results of the computation. It characterizes the specific task to be performed : classification or regression. In regression tasks a single neuron realizes the output layer and usually the activation

function is not present. When dealing with classification the number of neurons indicates the number classes to predict, when a single neuron is present the task is indicated as a binary-classification task.

NNs with a number of hidden layers greater than one (e.g., example of figure 2.4) are denoted as a Deep NN (DNN). Having the input and output layers fixed by the task to be accomplished, the design of an ANN reduces to the choice of the number of hidden layers (i.e., the *depth*) and the number of neurons (i.e., the *width*) present in every layer. A general rule to choose depth and width of NNs does not exist and usually obtaining an optimal result requires many trial-and-error steps. Nevertheless some heuristics exist, that allow to identify a range of candidates depth and width. In particular it is important to avoid the two situations of *underfitting* and *overfitting*. In case of underfitting the network is too small and is not able to learn patterns and extract features from the training examples. Increasing depth and/or width might improve a lot performances. Overfitting represents the opposite situation, the network is very large and has the capability to fit perfectly the training set. In this situations the network is unable to generalize and performs very poorly on examples not seen during training. This is symptomatic of an oversized network, reducing depth and/or width can solve the problem.

2.4 Training of DNNs

The most crucial step in all Machine Learning (and DL) algorithms is certainly related to the training phase : a not properly trained Machine Learning model is a poor performing model. Depending on the type of algorithm several training techniques are possible. In the field of DNNs, due to the extreme complexity of models, a *closed-form solution* is completely impossible. Moreover due to the presence of non-linearity (activation functions) the problem to be solved, i.e., the optimization of neurons' weights and biases, becomes *non-convex* [22]. The training process is essentially a non-convex optimization problem where the function to be optimized is the so called *cost* or *loss* function $\mathcal{L}$, which could be a generic function that measures somehow the distance between the expected output of the network and the actual one. The set of parameters to be optimized (weights and biases) is usually denoted by the greek letter $\boldsymbol{\theta}$.

The loss function characterizes intimately the task to be solved, then the

choice of the proper loss is of primary importance when designing a Machine Learning algorithm.

Due to the non-convexity of the loss function is impossible to guarantee convergence to an absolute minimum, for this reason ANNs are trained by means of iterative *gradient-based* techniques that drives the cost function to low values, without ensuring that is the lowest possible point.

2.4.1 Gradient-Based Learning

Nearly all DL models are trained by means of algorithms based on Stochastic Gradient Descent (SGD). SGD represents an evolution, which is well suited when dealing with complex models and large amount of data, of the generic optimization algorithm denoted as *Gradient Descent*.

The starting point of every gradient-descent algorithm is the evaluation of a function to be minimized, in our case is the loss $\mathcal{L}(\boldsymbol{\theta})$. Then the gradient of $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ is evaluated : $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$. This quantity is used in the update rule of $\boldsymbol{\theta}$ as follows :

$$\boldsymbol{\theta}^{new} = \boldsymbol{\theta}^{old} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \tag{2.6}$$

Where η is known as the *learning rate*, which represents the most important parameter of the training process. The learning rate controls the rate of update of model's parameters : if it is too low the algorithm will require a very large number of iteration to converge; if it is too high the algorithm diverges.

The plain gradient-descent algorithm, also identified as batch gradient descent, requires the usage of the whole training set at every iteration, thus when it is composed of a very large number of instances the training process is dramatically slowed down. SGD instead picks a random instance of the dataset at every iteration, in this way the algorithm becomes much faster, but the optimization process is much less straightforward : the loss does not decrease smoothly but erratically goes up and down, decreasing only on average [19]. This is of course a drawback but at the same time helps the convergence to the global minimum because is possible to jump out from local minima. As it often happens the best choice lies in the middle, usually SGD is used sampling not a single instance but a certain number of instances that form together a *mini-batch*. The size of the mini-batch is another free parameter in the hands of the designer.

The most intensive part of SGD is certainly the evaluation of all the gradients, therefore the way for an efficient training pass through an efficient evaluation of the derivatives. In [25] the *backpropagation* algorithm is presented, which is the other cornerstone, along with SGD, of ANNs training. Backpropagation evaluates all the derivatives of $\mathcal{L}(\boldsymbol{\theta})$ with respect to to $\boldsymbol{\theta}$ passing only two times through the network. The first pass is the *forward pass* : the mini-batch is passed through the network and the loss evaluated. The second pass is the *backward pass* : starting from the output of the network all the gradients are evaluated by means of the well-known *chain-rule* of calculus. This action is performed for every layer having gradients that flow from the output to the input. At the end weights and biases are updated with the rule of equation 2.6.

2.4.2 Most Common Loss Functions

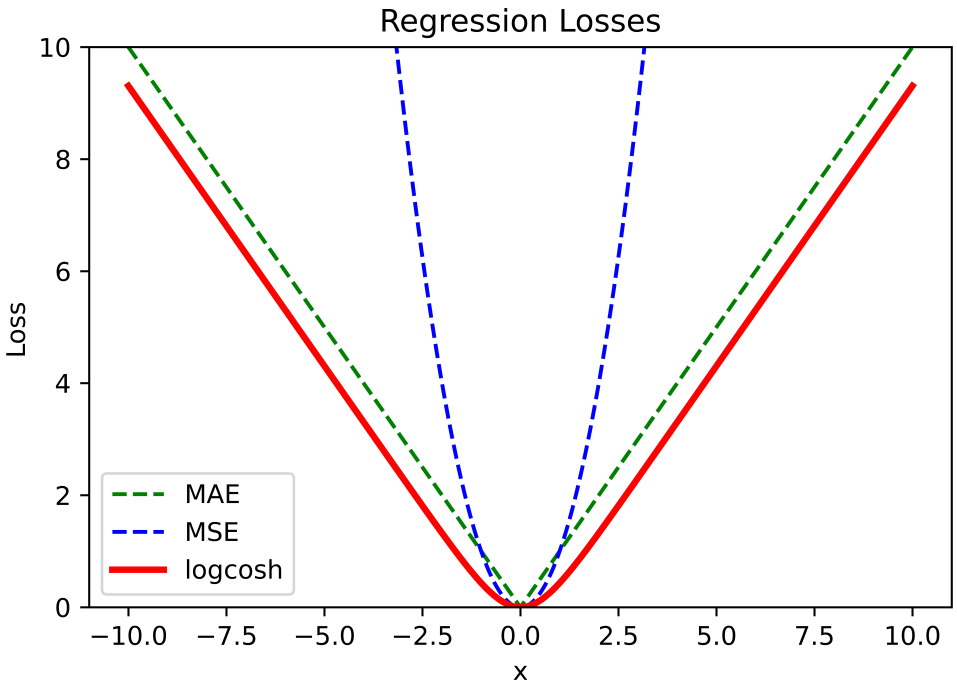

Figure 2.5: Comparison between some of the most common loss functions employed in regression tasks.

Depending on the task several loss functions can be employed, in particular

here we concentrate mainly on regression, since later on we will benchmark our NAS methods on a regression task. Some of the most common are :

- *Mean Squared Error (MSE)* : loss function commonly employed in regression tasks, it minimizes the squared differences between the estimated and existing target values. Its analytical form is : $\mathcal{L} = \frac{1}{N}\sum_{i=0}^{N}(y_i - \tilde{y_i})^2$, where $\tilde{y}$ is the predicted value. It is easily differentiable in its whole domain but it is very susceptible to outliers.

- *MAE* : loss function commonly employed in regression tasks, it minimizes the absolute differences between the estimated and existing target values. Its analytical form is : $\mathcal{L} = \sum_{i=0}^{N}|y_i - \tilde{y_i}|$, where $\tilde{y}$ is the predicted value. It is not differentiable in its minimum, then some problems of convergence might arise, furthermore its gradient is the same throughout, which means that even for small loss values the gradient will be large. Then it is not optimal for training based on SGD. It is very robust to outliers.

- *logcosh loss* : it is a compromise between MSE and MAE. Far away from the minimum it acts as MAE, instead near it the behavior of the function is smoother resembling MSE. Its analytical form is : $\mathcal{L} = \sum_{i=0}^{N} log(cosh(y_i - \tilde{y_i}))$.

In figure 2.5 the analyzed regression losses are reported.

For the sake of completeness the most common loss function used in classification tasks is here reported : *Cross-Entropy Loss*, which computes the cross-entropy between labels and predictions. In the case of binary classification, i.e., only two classes are present, the loss is : $\mathcal{L} = -(ylog(p) + (1-y)log(1-p))$, where y is the label and p is the output probability evaluated by the network. In the case of multi-class classification with M the number of classes the loss becomes : $\mathcal{L} = -\sum_{c=1}^{M} y_c log(p_c)$.

2.4.3 Regularization Techniques

In section 2.4.1 the loss $\mathcal{L}$ takes into account only the part related to the error between expected and actual outputs. In many cases is useful to add another term to the loss $\mathcal{L}$, which is called *regularization* loss. This additional term usually is a soft constraint on the parameters values. Possible reasons behind regularization are : improving performance on the test set, i.e., improving generalization; encoding a prior knowledge; promoting one family of solutions

over others in the space of possible solutions, e.g., promoting solutions that present, in absolute value, smaller weights.

The most common regularization techniques are the ones based on norm penalties [22], in particular we recall :

- L^2*-Regularization*, also known as *ridge* or *Tikhonov* regularization. The term added to the loss $\mathcal{L}(\boldsymbol{\theta})$ is the squared L^2 norm of $\boldsymbol{\theta}$:

$$\mathcal{L}_{ridge} = \lambda \|\boldsymbol{\theta}\|_2^2 = \lambda \sum \theta_i^2 \tag{2.7}$$

  The resultant effect of ridge regularization is driving weights to the possible smallest values [19].

- L^1*-Regularization*, also known as *LASSO(Least Absolute Shrinkage and Selection Operator Regression)* regularization. The term added to the loss $\mathcal{L}(\boldsymbol{\theta})$ is the L^1 norm of $\boldsymbol{\theta}$:

$$\mathcal{L}_{LASSO} = \lambda \|\boldsymbol{\theta}\|_1 = \lambda \sum |\theta_i| \tag{2.8}$$

  The resultant effect of LASSO regularization is that it tends to drive to zero weights of least importance [19]. Then model trained with penalty will present *sparse* weights. For this reason regularizer based on LASSO are known as *sparsifying* regularizers and are at the base of many pruning techniques [26], [27], [28].

In both equations 2.7 and 2.8 the term λ is known as *regularization strength* and it controls the amount of penalty introduced in the total loss, thus how strong is the effect of the regularization.

2.5 Popular DNNs Architectures

DNNs can be categorized with respect to peculiarities of the underline architecture. One of the defining characteristics of a network is the flow followed by data during inference. Relying on this, it is possible to divide DNNs in two families : *feed-forward neural networks* and *recurrent neural networks*. The first ones are characterized by the absence of feedback connections, meaning that data flow in a single direction, from the input to the output. In contrast, the second ones present feedback loops, which allow the network to have memory of previous computations. In the following only feed-forward

neural networks are considered, since the benchmark application on which we evaluate our NAS methods is based on a network of that type.

The most known (and studied) feed-forward neural network architecture is, without any doubt, represented by CNN and its variants. Here a brief review of the main characteristics of CNNs and its uni-dimensional variants TCNs is provided.

2.5.1 Convolutional Neural Network (CNN)

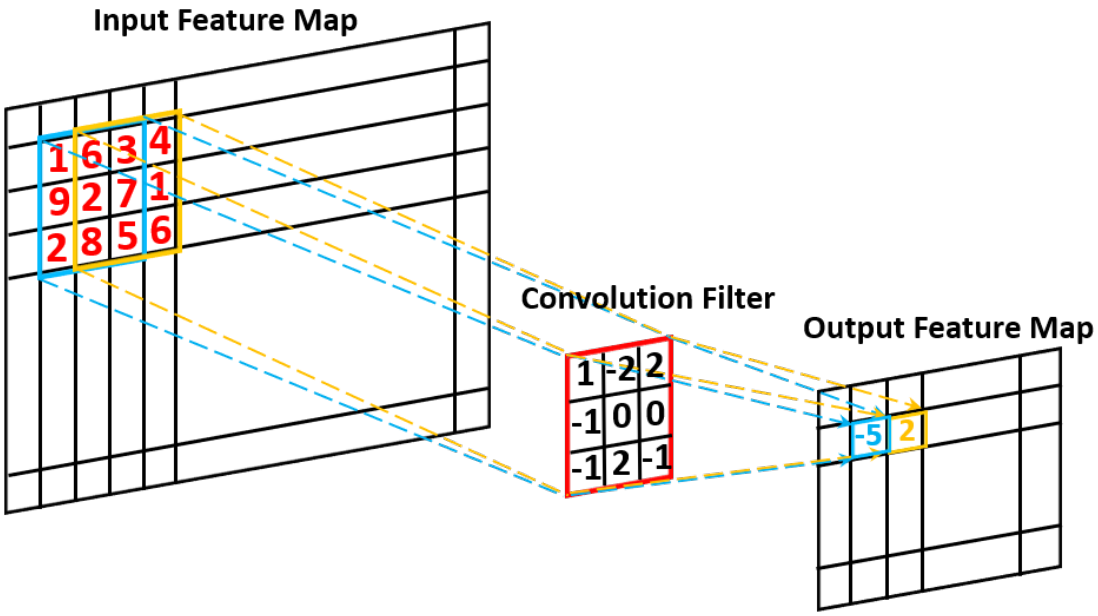

Figure 2.6: Example of 2D Convolution.

Convolutional Neural Networks (CNNs) are specialized network developed to handle multi-dimensional data like images. CNNs represents today the de-facto standard in CV tasks [29] such as object detection or face recognition. As the name suggest, the core of CNNs is represented by a layer that performs a *convolution* operation. Others commonly employed layers are *pooling* and *normalization*. The last layers of CNNs are usually a certain number of MLPs, also identified as *fully-connected* or *dense* layers. They are used to combine features extracted from the upper part of the network and perform the final classification or regression.

Convolutional Layer

In figure 2.6 an example of a 2D convolution operation is reported. The input of a convolutional layer is usually denoted as *input feature map* (or *ifmap*) and the output is the *output feature map* (or *ofmap*). Here, without loss of generality, ifmap and ofmap are assumed to be squared, which is the most common situation.

A point (or pixel) in the ofmap is obtained moving rigidly a filter, called the convolution filter, across portions of the ifmap, computing the sum of the elements of the Hadamard product between the filter and the corresponding window in the ifmap. The filters are made of weights learned during the training phase of the network. Mathematically this operation can be written as :

$$y_{(m,n)} = \sum_{i=0}^{k} \sum_{j=0}^{k} K_{(i,j)} \cdot x_{(m+i-i(s-1),n+j+j(s-1)} \tag{2.9}$$

Where $y_{(m,n)}$ is an output pixel in position (m, n), x denotes the ifmap and K is the convolutional filter (sometimes identified also as the *kernel*) of dimension $k \times k$. Instead s denotes the *stride* parameter, an integer that measures how far is moved the window across the ifmap between two successive steps of the convolution. The dimension of the output feature map can be simply evaluated with the relation :

$$d = \frac{t + p - k}{s} + 1 \tag{2.10}$$

Where t is the dimension of the input feature map and p the eventual amount of zero-padding. Usually each output $y_{(m,n)}$ is then passed through an activation function like ReLU (2.2.1). The operation presented in equation 2.9 describe the case of a single bi-dimensional input feature map, but it is easily extendable to the n-dimensional case, presented in equation 2.11.

$$y_{(m,n)}^{(c_{out})} = \sum_{c_{in}=0}^{C} \sum_{i=0}^{k} \sum_{j=0}^{k} K_{(i,j)}^{(c_{in},c_{out})} \cdot x_{(m+i-i(s-1),n+j+j(s-1)}^{(c_{in})} \tag{2.11}$$

In equation 2.11, the points of the output map y of a certain c_{out} output channel are obtained convolving all the c_{in} input channels with the different filters $K^{(c_{in},c_{out})}$ and computing their sum. Multi-dimensional convolution is usually employed in CNNs, where inputs with multiple channels are considered.

Pooling Layer

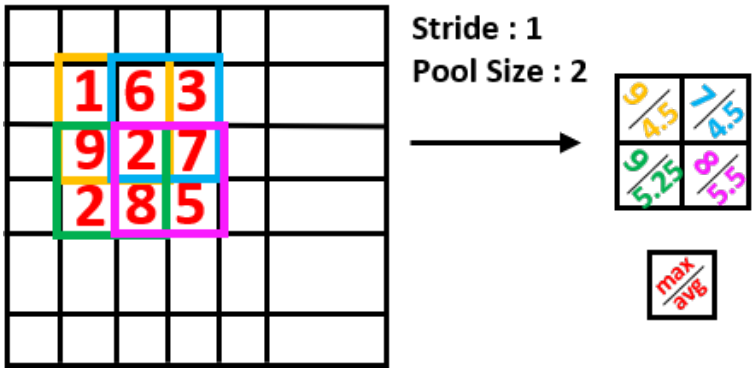

Figure 2.7: Example of Max/Average Pooling.

Besides convolutional layers, usually CNNs employ another common layer called the *pooling layer*, which usually follows the convolutional one. Pooling is in general a *sub-sampling* operation used to discard unimportant features. Moreover it helps to make representation approximately invariant to small translations of the input [22]. This layer is simply made of a sliding window moved across the input map, then it does not requires any trainable parameters. A Pooling operation is fully specified by the window's size, i.e., the *pool size*, the stride s and the implemented technique. Various pooling techniques exist, the most common are *average* and *max* pooling. In the case of average pooling the output of the pool window is the arithmetic mean of the inputs. Instead with max pooling the output is the largest of the inputs. In figure 2.7 an example of both approaches is presented. The dimension of the output feature map can be calculated again with equation 2.10.

Normalization Layer

Another common building block of modern CNNs is realized by normalization layers. Normalization layers are usually placed after the activation function that follows the convolution or in between them. This operation is used in order to speed-up the training phase and improving the generalization of the network mitigating overfitting. Literature is plenty of proposed normalization techniques, but without any doubt the most commonly employed is *Batch Normalization* [30]. Batch normalization is used to fix mean and variance of the data flowing in the network stabilizing their distribution. The

transformation applied to each point in every channel c of the input feature map x is presented in equation 2.12.

$$\hat{y}_i^{(c)} = \gamma^{(c)} \frac{x_i^{(c)} - \mu_B^{(c)}}{\sqrt{\sigma_B^{(c)^2} + \epsilon}} + \beta^{(c)} \tag{2.12}$$

The normalization is applied on batches B due to the impracticality to normalize on the entire training set in one-shot. For each batch B and channel c the corresponding mean ($\mu_B^{(c)}$) and variance ($\sigma_B^{(c)^2}$) are evaluated and used to perform the normalization. The ϵ parameter is a small number employed to avoid divisions by zero. Moreover, the normalized data are also processed with an affine transformation by means of the γ and β parameters, which are trainable.

2.5.2 Temporal Convolutional Network (TCN)

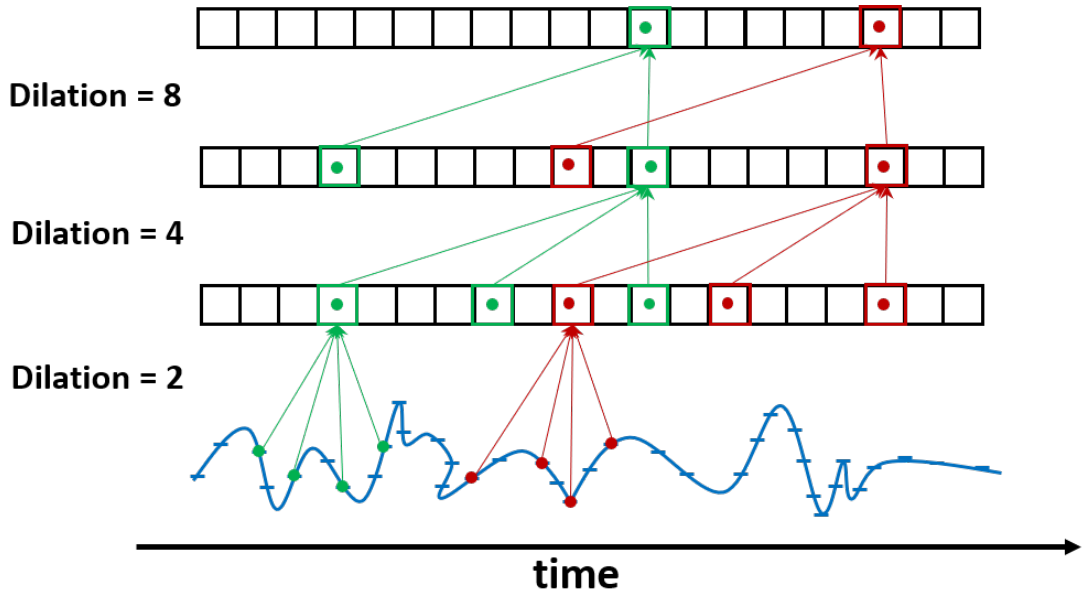

Figure 2.8: Example of 1D Dilated Convolution.

For many years the scene in DNN has been dominated by CNNs, thanks to the extraordinary results obtained with multi-dimensional data. On the other side, RNNs were considered for a long time the best performing DNN architectures for processing uni-dimensional time sequences (or series) of

data. Recently, starting from the work presented in [14], researchers have begun to investigate the usage of convolutional architectures to perform classical RNNs tasks like word/char-level language modeling and time series analysis. In particular *Temporal Convolutional Networks* (TCNs) have been shown to provide comparable results to state-of-the-art architectures on that tasks. TCNs offers many advantages from the computational standpoint with respect to RNNs: more data reuse opportunities, higher arithmetic intensity and smaller memory footprint [14].

Temporal convolution is a variant of the plain convolution presented in equation 2.9, whose peculiarity is in the use of *causality* and *dilation*. Causality forces the padding to be applied only on left side of the sequence; therefore, the outputs $\mathbf{y}_t$ will present the same length of the inputs $\mathbf{x}_{\tilde{t}}$ and are only functions of inputs with $\tilde{t} \leq t$. In other words, output samples respect time causality, not taking information from the future. Dilation is a fixed gap d inserted between input samples processed by the convolution. It is used to increase the receptive field of the operation on the time axis without requiring more parameters. The function implemented in a temporal convolutional layer is :

$$\mathbf{y}_t = \sum_{i=0}^{k-1} \sum_{l=0}^{C_{in}-1} \mathbf{x}_{t-d\,i}^{l} \cdot \mathbf{W}_{i}^{l,m} \tag{2.13}$$

Where $\mathbf{x}$ and $\mathbf{y}$ are the ifmap and ofmap, t and m the output time-step and channel respectively, $\mathbf{K}$ is the matrix of filter weights with k filter size, C_{in} the number of input channels and d the dilation factor. In figure 2.8 an example of dilated convolution is provided.

2.6 PPG-based Heart-Rate Estimation

This section provides a brief overview of PPG signals, which are part of the Dalia dataset [17], used as benchmark for this thesis work. PhotoPlethysmoGraphy (PPG) is an optical monitoring technique commonly employed to monitor the cardiovascular system. PPG is of high interest as an alternative to traditional ECG, since it is non-invasive and less expensive. A PPG sensor, represented in figure 2.10, consists of one or more LEDs, usually in the green spectrum, that continuously emit light to the skin and a photodiode that measures variations of light intensity caused blood flowing in veins. Variations of the blood flow are directly related to heart rate, thus by

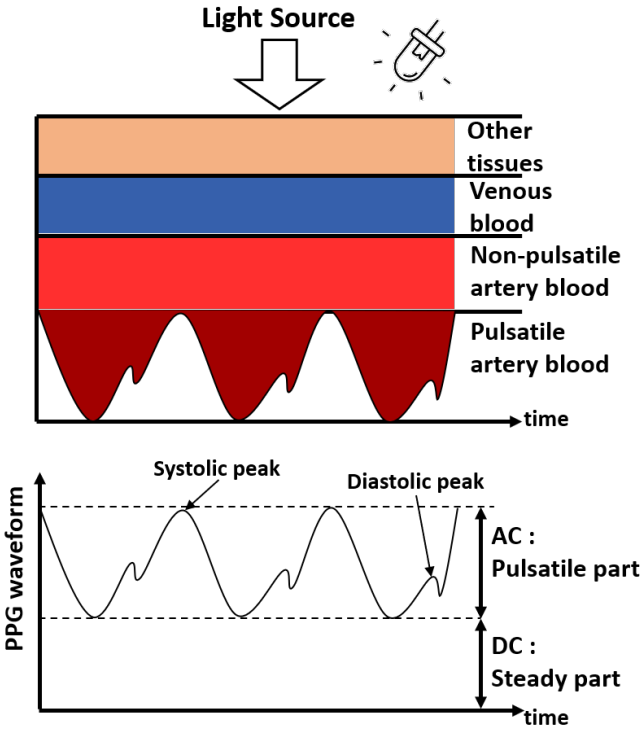

Figure 2.9: PPG waveform.

analyzing the PPG waveform it is possible to extrapolate information about it. Figure 2.9 reports a typical PPG signal, which is composed of two parts:

- DC component: this component is due to the optical reflection of tissues and to the volume of arterial and venous blood.
- AC component: this component represents the blood's variation of volume due to the heart activity. It is the useful part of the PPG signal from which is possible to extract the heart-rate information.

Moreover, in PPG signals it is possible to identify two phases, which are specific of the heart activity : the *systolic* and *diastolic* phases. During the systolic phase the signal increases in magnitude until it reaches a peak (systolic peak), then it follows the descending diastolic phase that ends with a secondary peak (diastolic peak). The interval between two consecutive systolic peaks represent an heartbeat. A major limitation of PPG-based HR estimation is due to Motion Artifacts (MAs). These are artifacts caused by movements of the user's arm, which result in ambient light leaking in

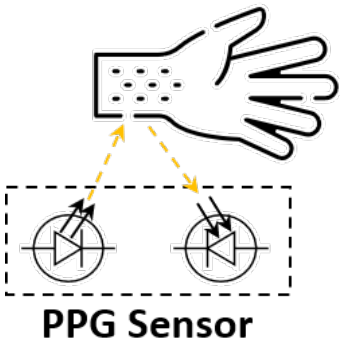

Figure 2.10: PPG Sensor.

the gap between the wrist and the photodiode or variations of the sensor pressure on the skin. Moreover, blood flow can vary considerably depending on the type of physical activity, contributing to a less precise light absorption measurement, and hence HR estimate [17]. In order to mitigate the effect of MAs several approaches has been proposed, many of which are based on a *sensor fusion* approach that combines the PPG sensor with a tri-axial accelerometer. Some of these approaches rely on classical algorithm [31], [32] requiring lots of hand-tuning, while some other are based on DL [17], [33], [34].

2.6.1 Dalia Dataset

The Dalia [17] dataset includes, along with PPG sensor data, also data collected from a 3D accelerometer, which are used together to produce a reliable heart-rate estimation, mitigating the effect of MAs. The heart-rate ground truth value is obtained with an R-peak detector [35] applied on ElectroCardioGram (ECG) signals collected simultaneously with the rest of the data. Acceleration and PPG signals are collected from a wrist-worn device (Empatica E4 [36]), while ECG data are collected with a chest-worn device (RespiBAN Professional [37]).

Data are collected from 15 subjects of different gender (seven male and eight female) and age (30.60±9.59 years). Each subject is denoted throughout this thesis with a capitol S followed by a specific number between 1 and 15. A total of 37.5 hours of recording are available, where the subjects perform different activities such as cycling, driving, walking, etc. Before feeding the

network the raw data are pre-processed. In particular, data are sampled at 32 Hz and organized by means of a sliding window that see 8 s of history. Each window overlaps with the previous one of 6 s.

Chapter 3

Related Works

In chapter 2 a brief overview of DNNs architectures has been provided. What should be clear is that the design space is extremely vast. In fact the number of *hyper-parameters* that can be tuned and modified is extremely high, e.g., the number and type of layers, the specific parameters of layers (filter size, number of channels, dilations, ...) and so on. Some intuitions of what works better in specific situations exist, but general design rules do not, thus building ANNs is somehow an art. Moreover the common trend in DNNs is having more and more complex and deeper models. For this reason, the number of possible combinations of hyper-parameters explodes. For this reason a new trend in NNs' research is Neural Architecture Search, which is a general term that identify the usage of algorithmic techniques techniques to automate the design of DNNs. NAS tools are designed to explore efficiently the design space optimizing jointly the accuracy of the model and eventually other metrics like the memory footprint (i.e., the number of parameters), the number of FLOPs [12] or the latency [11]. NAS outputs architectures are Pareto-points in the considered complexity-accuracy space, where the complexity is the metric to be optimized. In particular, finding size-optimized architectures is extremely attractive in order to deploy them on edge devices [38], [39].

In literature, different approaches to NAS have been proposed. The forerunners are techniques based on reinforcement learning like *NAS-RL* [40] and *MetaQNN* [41] or evolutionary algorithms [42]. These methods provide state-of-the-art results on the considered datasets, but they require an enormous amount of GPU hours, thus being prohibitive to apply on a large scale. Therefore successive NAS works have moved in the direction of reducing

the amount of resources required to perform the search. Early NAS algorithms train each candidate architecture from scratch; a first intuition that allows to speed-up a lot the search phase is the one proposed in *ENAS* [43], where weights are inherited from previous searching phases. Weight sharing requires a constrained search space, which becomes smaller, but this allows to run the NAS in a reasonable time. Nevertheless, this approach still requires multiple models to be trained and reinforcement learning is notoriously not so fast. Therefore, recent research has focused on techniques that allow a single model to be trained. Differentiable Neural Architecture Search (DNAS) solutions, first time presented in *DARTS* [10], allow to do that. In

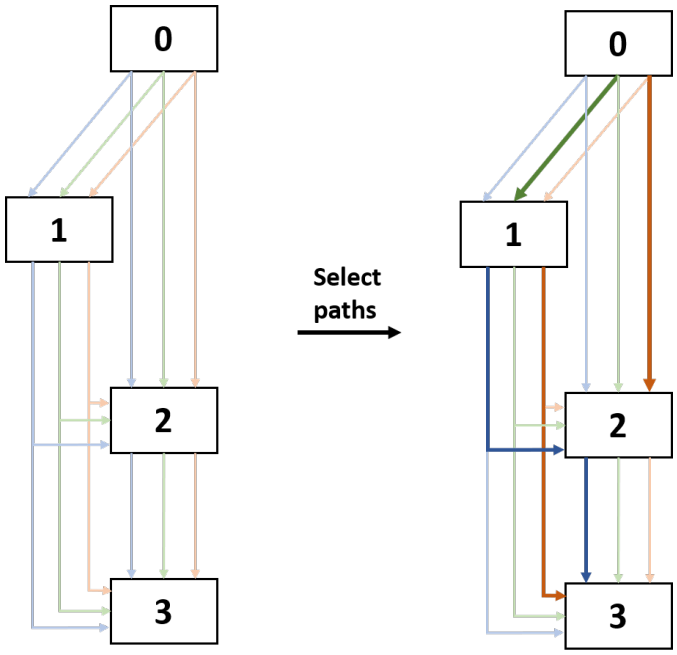

Figure 3.1: Example of supernet and path selection.

DNAS the problem is modeled as the optimization of a *supernet*, that embeds different implementations of each layer. In figure 3.1 an example of supernet, represented as a Direct Acyclic Graph (DAG) is provided. Each edge of the DAG represents a candidate layer and is associated with a trainable weight that is learned during training, alternating their optimization with the one of "normal" network parameters. Edges are collected in sum nodes. At the end of the training and searching phase the learned architecture will be the one that is composed by the edges with the highest weights. This solution is much more efficient than previous ones, but the memory requirements and training

time is still very high, due to the complex architecture of the supernet and mostly it does not scale well with the supernet size. These problems are tackled by another DNAS method, *ProxylessNAS* [11], by training a single path of the DAG per batch. This allows to limit the memory occupation of the whole supernet, but the search space is not fully explored at the same time.

Lastly, an emerging set of DNAS techniques is based on the *Dmasking-NAS* approach [12], [13], where the search space is limited to a single *seed* architecture. This method is used to search and optimize hyper-parameters characterizing specific layers, e.g., number of output channels/neurons in convolutional/dense layers. DmaskingNAS methods, as the name suggest, are based on differentiable trainable masks which are used to tweak layers' parameters to explore different architectures during the search. In detail, for each layer l and for each hyper-parameter h that enters in the learning process a set of additional weights $\gamma^{(l,h)}$ is inserted. The $\gamma_i^{(l,h)}$ weights are then combined with normal ones, in such a way that when a $\gamma_i^{(l,h)}$ passes from the value 1 to 0 the architecture changes. The main effort of this methods is devoted to making these discrete masks differentiable.

MorphNet [12] is a DmaskingNAS method that searches the optimal number of output channels of convolutional layer. It exploit as masks weights the multiplicative terms of batch normalization layer, which are then driven to 0 in order to yield a simplified architecture, with an approach similar to weight pruning. MorphNet uses specific regularizers to optimize specific metrics such as size, FLOPs and latency. Further details about MorphNet are provided in chapter 4 where the algorithm is applied to TCNs.

FBNetV2 [13] also searches for the optimal number of output channels of convolutional layer, but its method is easily extendable to other hyper-parameters such as the filter size. Its approach is based on mutually exclusive masks. For example, if a certain convolutional layer presents N output channels, then it is possible to build a specific number of masks with l leading *1s* and *N-l* trailing *0s*, that correspond to variants of the layer with *N-l* output filters, with l = *1, 2, ..., N* . Masks are then multiplied with a *Gumbel-Softmax* weight [44], summed together and multiplied with the output of convolution.

The search space in DMaskingNAS algorithms is smaller than DNAS ones, but it is wide enough to find good solutions, and the time overhead with respect to plain training of the network is negligible. Moreover this approach can be used to further optimize a seed network, already well performing on

its task. In particular, it is possible to use it to enable inference on edge devices compressing well performing, but too large, seed architectures.

Chapter 4

NAS Techniques for Edge-TCNs Optimization

4.1 Motivations and Objectives

For many years DL has been confined to the cloud, whereas models were extremely complex and resource-hungry. Therefore, the only way to effectively exploit them in real-world application leans on high-performance computing platforms with highly-parallel hardware such GPUs.

The world of Mobile and IoT applications is, without any doubt, one of the fields that could benefit more from powerful DNNs. Today, lots of different DL applications effectively "run" on this type of devices, jointly identified as *edge* devices. Albeit, the execution of DL algorithms is still mostly confined to the cloud. The main reason for this is that edge devices usually do not provide an amount of HW resources, especially in terms of memory, that effectively allows it to run complex models such ANNs.
This is the main reason why the computational part is demanded to high-performance servers : the edge device acts only as an interface that collects and sends data to be processed and then waits for the elaboration result.

This cloud-based inference computing paradigm presents several problems, such :

- High Latency : the inference is not real-time and includes a time overhead related to the wireless communication with the cloud. This is especially problematic for applications that could benefit from real-time inference, like face-recognition or Heart-Rate tracking.

- Privacy and Data Integrity : the data collected at the edge and sent to a server might be sensitive for the privacy. Moreover, the client has no mean to control if its data are collected or manipulated by a malicious third party.

- Energy: the data pipeline collect-send-wait-receive is not efficient also from an energy perspective, which is of primary importance in edge devices, usually powered with a battery. Indeed, doing every computation at the edge is usually more efficient mainly because it does not involve all the "wasted" energy in the transmission of data and all the consumed energy is devoted to useful computations. Two main reasons lie behind this fact. First, the energy required for transmitting and receiving does not scale with technology in the same way as compute energy. Second, the wireless links are often long range, thus they require a great amount of energy.

For this reasons lot of effort is devoted to an alternative computing paradigm for DL, which brings inference from the cloud to the edge. In particular lot of effort is devoted to the compression of DNNs with negligible drops in accuracy, so that these models can be deployed on memory- and energy-constrained edge devices.

This thesis work deals with edge-inference optimization of time series. The problem is tackled by means of TCNs, which represent an HW-friendly alternative to RNNs. TCNs represent a quite novel DL architecture, in fact few works are present in literature and even less in the direction of their optimization. For this reason, this work proposes two NAS techniques applied specifically to TCNs.
The first one represents an application of an existing NAS, MorphNet [12] briefly described in chapter 3 and more deeply presented in section 4.2. MorphNet was originally developed for CNNs and here is applied to TCNs for the first time.
The second one is the novel NAS algorithm PIT, developed in this thesis and presented in 4.3, which allows to optimize the dilation factor parameters of temporal convolutional layers. Both methods are tailored in a way that allows to compress networks enabling inference-on-the-edge.

The proposed NAS techniques are tested on a real IoT task that needs real-time computations and the models obtained are tested on a commercial edge platform. The task taken into consideration is PPG-based Heart-Rate Monitoring on wrist-worn devices, such as the Apple Watch [45] or the Fitbit

Charge 4 [46].

4.2 TimePPG : Optimized TCNs for PPG-based Heart Rate Monitoring

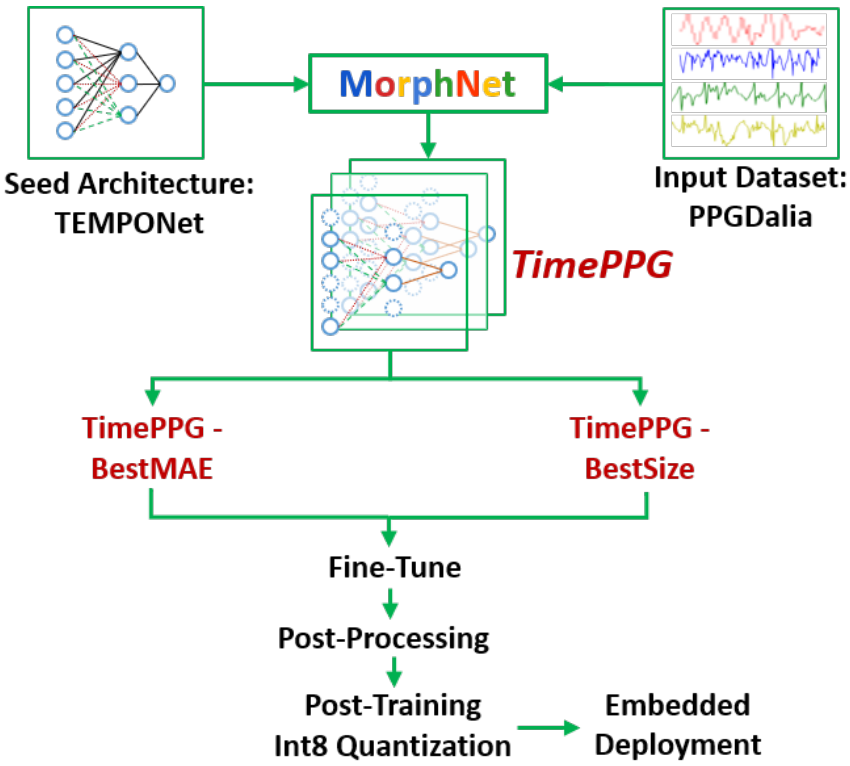

Figure 4.1: Proposed NAS and deployment flow.

Figure 4.1 summarizes the *TimePPG* flow, i.e., the steps performed to optimize the pre-existing TCN TEMPONet [16] by means of the MorphNet [12] NAS technique on the Dalia [17] dataset. The picture highlights two specific architectures that MorphNet found : *TimePPG-BestMAE* and *TimePPG-BestSize*. The former represents the Pareto point in the Size vs MAE [1] space that achieves the lowest MAE. The latter represents the Pareto point with lowest size.

[1]MAE is used as metric to measure network performances. The analytical expression of MAE is the same presented in section 2.4.2

Figure 4.1 also shows the *Fine-Tune* and *Post-Processing* step, which are not part of the NAS algorithms but allow to outperform other DL algorithms in terms of performance on the Dalia dataset. These two additional improvements are briefly analyzed in section 4.4.

Finally the *Post-Training Int8 Quantization* is an additional step necessary to deploy the models on an edge device such as GAP8 MCU.

4.2.1 TEMPONet

The base architecture used in this work is *TEMPONet*, a TCN which shows impressive results in bio-signals analysis tasks, like EMG-based gesture recognition [16]. TEMPONet presents a modular structure composed of three Convolutional Blocks, where each block is made of :

- two temporal convolutional layers with kernel size 3×1, variable dilation and causal padding.
- one convolutional layer with kernel size 5×1, variable stride followed by an Average Pooling layer with stride and pool size 2.

The three blocks are characterized, respectively, by stride s = 1, 2, 4 and by dilation d = 2, 4, 8. The strided convolution of the three blocks raises the number of channels to 32, 64 and 128 respectively, while each pooling reduces the sequence length. The three convolutional blocks are followed by two Fully Connected (FC) layers with dropout (to improve generalization [47]) and a SoftMax operation. All layers include ReLU non-linearity as activation function and are equipped with Batch-Normalization [30].

To adapt TEMPONet to Heart-Rate monitoring, it is necessary to slightly modify the original architecture, in particular the input and output layers. The input needs to be adapted to the target dataset Dalia, which is made of raw sensor data generated by a triaxial accelerometer and a PPG sensor. An example of these four raw signals is reported in the leftmost part of figure 4.2. The data are sampled at 32 Hz and the inputs for the network are obtained by means of a sliding window that see 8 s of history. The overlap between two successive windows is 6 s, thus at each step the window movement is 2 s as shown in the rightmost part of figure 4.2. The resulting four time series present a number of samples equal to 256, thus the input needs to collect a 256×4 matrix.
The original output layer made of a classifier with softmax activation function

is replaced by a single neuron, without any activation function, to perform regression. The modified version of TEMPONet is depicted and summarized in figure 4.3.

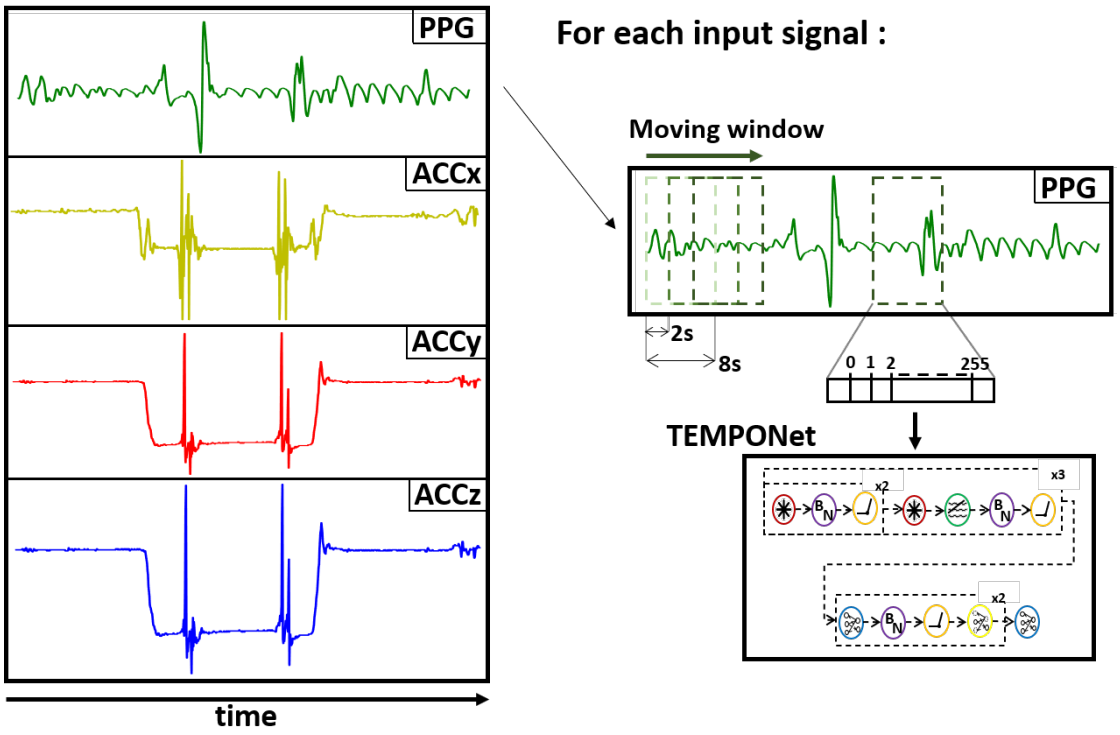

Figure 4.2: Raw input signals and sliding window sampling.

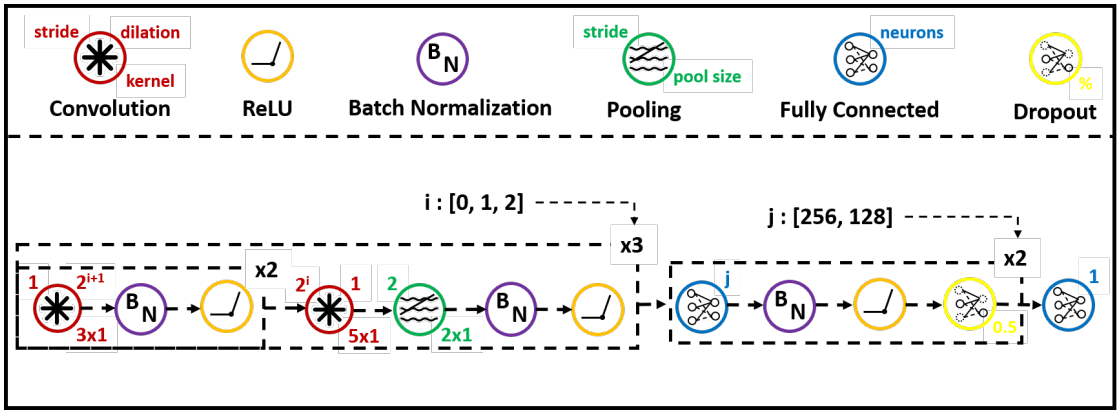

Figure 4.3: Modified TEMPONet architecture.

4.2.2 MorphNet Details

Algorithm 1 MorphNet Algorithm

1: **procedure** MorphNet($\mathcal{F}, \mathcal{L}, \mathcal{G}, \lambda, \omega, \zeta, \boldsymbol{\theta}, th$)

2: ▷ $\mathcal{F}$ is a cost metric, e.g., number of FLOPs, model size
3: ▷ $\mathcal{L}$ is a generic loss function used to train the model
4: ▷ $\mathcal{G}$ is the regularization loss
5: ▷ λ is the regularization strength
6: ▷ ω is the width multiplier
7: ▷ ζ is the upper bound for the constraint $\mathcal{F}$
8: ▷ $\boldsymbol{\theta}$ is the set of weights to be optimized
9: ▷ th is a threshold parameter used to prune channels

10: ▷ Execution :

11: Train the network : $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \{\mathcal{L}(\boldsymbol{\theta}) + \lambda \mathcal{G}(\boldsymbol{\theta})\}$
12: Find new number of output channels $O^{'}_{1:M}$ induced by $\boldsymbol{\theta}^*$ and th
13: Find largest ω such that $\mathcal{F}(\omega \cdot O^{'}_{1:M}) \leq \zeta$
14: **goto** *Line 11* until desired, setting $O^{o}_{1:M} = O^{'}_{1:M}$
15: **return** $\omega \cdot O^{'}_{1:M}$
16: **end procedure**

MorphNet [12], as mentioned in chapter 3, is a NAS approach that can be categorized as a DMaskingNAS technique. This algorithm, starting from a seed network, in a single training is able to optimize the number of output channels of each convolutional layer. Other layers are kept unchanged.

The pseudo-code of the algorithm is presented in 1 and a Python implementation based on the Keras deep learning framework is open-sourced at [48]. This NAS is composed of two successive steps repeated iteratively on a seed network denoted as $O^{o}_{1:M}$, where M convolutional layers are present :

- *Pruning Step* (*Lines 11-12*) : the loss function $\mathcal{L}$ is augmented with a sparsifying regularizer $\mathcal{G}$, used to induce sparsity in the network. Moreover, the regularizer is such that it puts more cost on neurons that contribute more to a certain metric $\mathcal{F}$ such size, FLOPs or latency. The regularizer is controlled by the strength parameter λ. After the training

output channels are pruned by means of *th* threshold.

- *Expansion Step* (*Line 13*) : the pruning step shrinks the network, usually affecting performance. This expansion step is based on a uniform *width multiplier* [49], which allows to recover accuracy. The width multiplier technique consists in simply finding the largest ω that allows to respect the $\mathcal{F}(\omega \cdot O'_{1:M}) \leq \zeta$ constraint.

In this work we employ as $\mathcal{F}$ measure the model size and the FLOPs. We consider a certain convolutional layer L, with I_L input channels, O_L output channels, (x_{in}, y_{in}) input spatial dimensions, (x_{out}, y_{out}) output spatial dimensions and (k_x, k_y) filter dimensions. The $\mathcal{F}$ expression for FLOPs will be :

$$\mathcal{F}_{FLOPs} = 2x_{out}y_{out}k_xk_y \cdot I_LO_L \tag{4.1}$$

For the model size, instead :

$$\mathcal{F}_{size} = k_xk_y \cdot I_LO_L \tag{4.2}$$

TEMPONet has been tested and optimized with both the metrics.

The complete regularizer $\mathcal{G}$ embeds the cost function $\mathcal{F}$ and a certain regularization strategy. If batch-normalization layers are present in the DNNs the trainable parameter γ of such layer is used to mask entire output channels of convolutional layers. Sparsification of γ parameters is promoted by means of L_1-Regularization (2.4.3). An alternative employed regularization technique, which adapts well to architectures without batch normalization layers, is the so-called group LASSO regularization [50]. Figure 4.4 reports a comparison between the plain Lasso (upper-most) and the group-Lasso (lower-most) along with its mathematical expression. In group-Lasso groups g_i of weights are identified, which together form a set G, such that $\bigcup_i g_i \equiv G \equiv \boldsymbol{\theta_{conv}}$, with $\boldsymbol{\theta}_{conv}$ denoting the fraction of total weight $\boldsymbol{\theta}$ related to convolutional layers. The plain Lasso regularization leads to a sparse (or unstructured) pruning, instead group-Lasso pruning is structured. MorphNet forms the g_i groups with the weights related to each output channels, thus a structured pruning of output channels will be performed during training.

TEMPONet besides convolutional layers presents also three large fully-connected layers, which contribute about the 25% to the size, thus also optimizing this layers along with convolutional one would be extremely valuable. Unfortunately, the publicly available MorphNet code, at the time of starting this thesis work, only optimized convolutional layers. Fortunately,

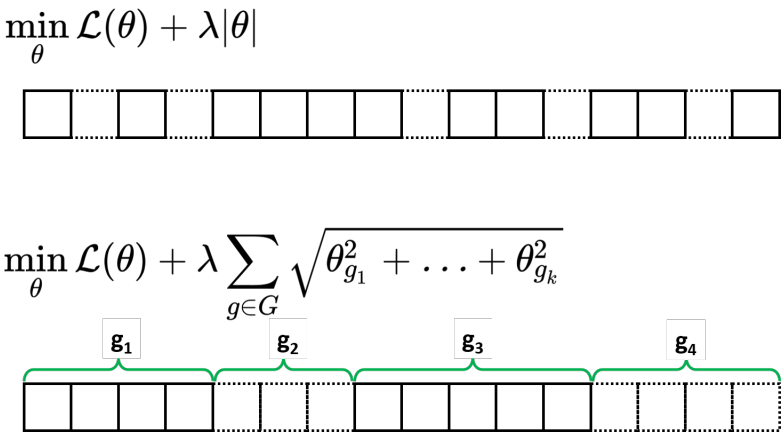

Figure 4.4: Plain Lasso regularizer vs Group Lasso regularizer.

fully connected layers can be easily seen as particular cases of convolutional ones : a fully connected layer with N_{in} input and N_{out} output neurons is equivalent to a convolutional layer with N_{in} input and N_{out} output channels with a kernel size equal to 1. Transforming FC layers in convolutional ones in this way allows MorphNet to optimize them too.

4.3 PIT : Pruning In Time

Most of the developed NAS tools address hyper-parameters optimization of CNNs. Despite the fact thatTCNs share most of their hyper-parameters with CNNs (e.g., filter sizes, number of output channels, etc.) the key peculiarity of those networks, i.e., the dilation factor, is not covered by any NAS. For example, there is no way to optimize dilation factors using MorphNet. As detailed in section 2.5.2, dilation allows to enlarge the receptive field of the network without any size-overhead. Therefore, finds their optimal value is extremely valuable in making DNNs more hardware-friendly.

Pruning In Time (PIT) is a novel light-weight DMaskingNAS algorithm that tries to optimize dilation factor of TCNs considering jointly accuracy and complexity. As MorphNet does, PIT starts from a seed network to be optimized and tackles the optimization problem as a *structured pruning of weights*, performing the search in a single training run.

The proposed method is depicted in figure 4.5 : PIT takes as input the target dataset that defines the task to be accomplished and a seed TCN with maximally-sized filters without dilation (i.e., every dilation factors is equal

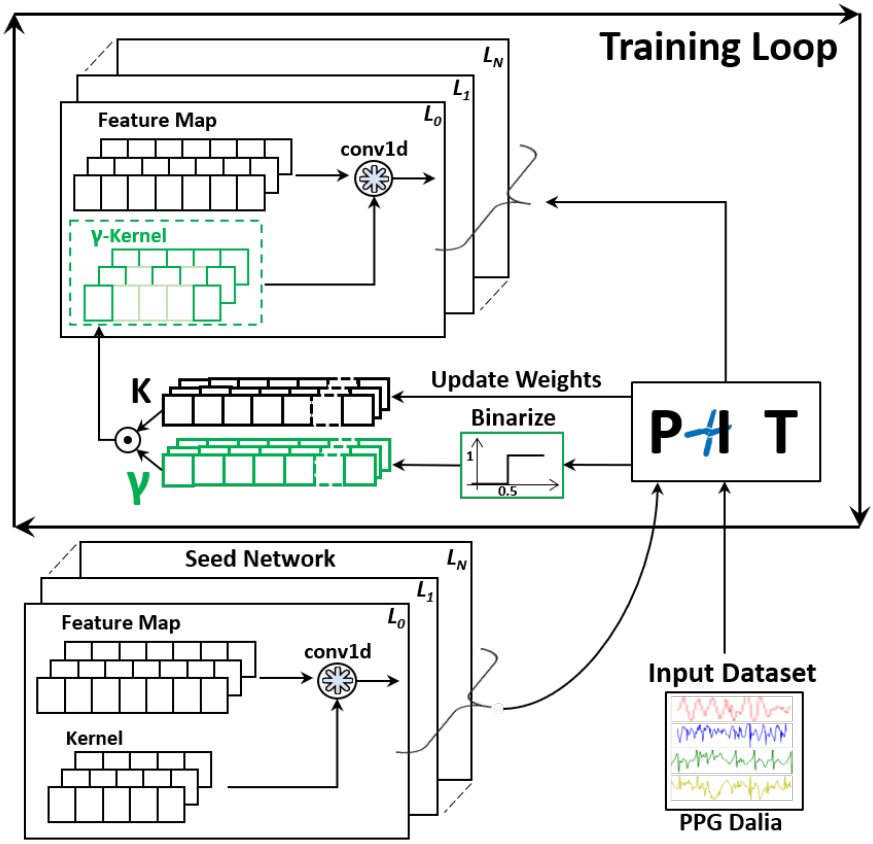

Figure 4.5: Training flow of the proposed Pruning In Time (PIT) NAS tool.

to 1). For each convolutional layer a set of γ parameters is added, which are then combined in a differentiable manner, in order to mask weights of the kernel and realize different dilations factors. Further details are provided in section 4.3.1 and 4.3.2. In section 4.3.3 the regularization strategy is presented, while section 4.3.4 explain the PIT's training procedure.

4.3.1 Dilation-Aware Mask

Each convolutional layer in a TCN is characterized by a certain receptive field rf, which is defined as $rf = d \cdot (k-1)$, where d is the dilation factor and k the kernel size. PIT limits the allowed dilation-search space to regular values, which are better from an edge-deployment perspective. Regular dilation is the only variant supported supported by current inference libraries for

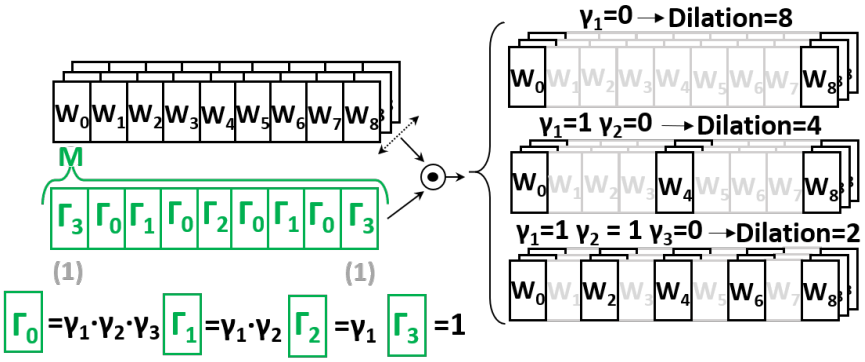

Figure 4.6: Combination of γ parameters with each other and point-wise multiplication to convolutional kernels to form different dilation patterns. Example for $rf_{max} = 9$.

MCUs [51], [52], and enable more regular memory access patterns along with better low-level optimizations. Moreover, PIT further restricts the search space only to dilations which are powers of 2. This choice enables a simpler formalization of the dilation-learning problem.

In PIT, every temporal convolutional layer is characterized by its maximum receptive field rf_{max}. This value is used to define the length $L = \lfloor \log_2(rf_{\max} - 1) \rfloor + 1$ of a *binary vector* $\boldsymbol{\gamma}$ associated to each temporal convolution. The parameters γ_1 to γ_{L-1} are trainable and control dilation. Instead, the first element γ_0 is always equal to 1 and it is used to make the notation simpler and more consistent.

Figure 4.6 shows how the elements of $\boldsymbol{\gamma}$ are combined together in a way that the resulting search space is restricted to regular power-of-two dilation patterns : the γ_i are multiplied together forming the new set of parameters Γ_i.
The i^{th} element of Γ_i is simply :

$$\Gamma_i = \prod_{k=0}^{L-1-i} \boldsymbol{\gamma_k} \tag{4.3}$$

As shown in the green part of Figure 4.6, Γ_i are used as multiplicative factors for all the kernels of a certain layer, and their combination forms the overall mask vector $\boldsymbol{M}$.

The rightmost part of figure 4.6 shows an example of all the possible outcomes (except the trivial one where no weight is masked, i.e., $d = 1$) of

the masking operation. When $rf_{max} = 9 \Rightarrow L = 4$, four possible dilation patterns are possible :

(i) $\gamma_1 = 0 \Rightarrow d = 8$

(ii) $\gamma_1 = 1, \gamma_2 = 0 \Rightarrow d = 4$

(iii) $\gamma_1 = 1, \gamma_2 = 1, \gamma_3 = 0 \Rightarrow d = 2$

(iv) $\gamma_i = 1, \forall i \Rightarrow d = 1$.

Therefore, starting from the $d = 1$ case, which holds when $\Gamma_0 = \gamma_0 \cdot \gamma_1 \cdots \gamma_{L-1} = 1$, $d = 2$ is obtained removing the $\gamma_i = 1$ condition on the last factor of Γ_1, then $\Gamma_1 = \gamma_0 \cdot \gamma_1 \cdots \gamma_{L-2} = 1$. Larger values of d are obtained with the same procedure considering the next γ_i and Γ_i until the always true $\Gamma_{L-1} = \gamma_0 = 1$ is reached. It is therefore evident that, thanks to this formulation, all possible combinations of zero-values in the γ_i array correspond to a valid (regular) dilation pattern, so that the resulting network can be directly deployed on an edge device.

In PIT the equation 2.13 describing dilated convolution is modified as follows :

$$\mathbf{y}_t = \sum_{i=0}^{rf_{max}-1} \sum_{l=0}^{C_{in}-1} \mathbf{x}_{t-i}^{l} \cdot (\mathbf{M}_i \odot \mathbf{W}_i^{l,m}) \tag{4.4}$$

Where $\odot$ is the Hadamard product and all the other symbols maintain the same meaning defined before.

4.3.2 Making Dilation Differentiable

Equation 4.4 describes the behavior of the temporal convolutional layer in PIT when the forward pass is evaluated. In order to include $\boldsymbol{\gamma}$ parameters in the loss function and train the whole architecture as a DNAS algorithm it is essential to make the operation 4.4 completely differentiable. This requires working in two directions :

(i) Binarization of $\boldsymbol{\gamma}$: the different masks $\mathbf{M}_i$ are binary masks, thus the float trainable version of $\boldsymbol{\gamma}$, denoted as $\hat{\boldsymbol{\gamma}}$, needs to be binarized in the forward pass.

(ii) Encoding of dilation in $\mathbf{M}$: each binary mask is built combining the Γ_i factors. In order to train and search between different architectures by

means of $\boldsymbol{\gamma}$ trainable parameters it is necessary to allow the gradient flow during the backward pass, thus it is necessary to express $\mathbf{M}$ as a differentiable function of $\boldsymbol{\gamma}$.

Binarization of $\hat{\boldsymbol{\gamma}}$

In the forward pass, before evaluating the temporal convolution, the binarization of $\hat{\boldsymbol{\gamma}}$ is performed following the approach used in Binary-Connect [53]. The adopted binarization function is an Heaviside step functions with threshold δ :

$$\mathcal{H}(\hat{\gamma}_i - \delta) = \begin{cases} 1, & for\ \hat{\gamma}_i \geq \delta \\ 0, & for\ \hat{\gamma}_i < \delta \end{cases} \tag{4.5}$$

In all the experiments the value $\delta = 0.5$ is used. In the backward pass, when the derivatives are evaluated, the gradient of equation 4.5, which is zero almost everywhere, is treated with a straight-through estimator [53] : the Heaviside function is replaced with an identity function in order to allow the gradients to be propagated.

Differentiable Dilation Mask M

In order to build the $\mathbf{M}$ masks in a completely differentiable manner starting from the respective $\boldsymbol{\gamma}$ vector, the following tensor transformation is used :

$$\boldsymbol{M} = \prod_{columns} \{[(\boldsymbol{\gamma} \cdot \mathbb{1}_{1\times L}) \odot \boldsymbol{T} + (\mathbb{1}_{L\times L} - \boldsymbol{T})] \cdot \boldsymbol{K}\} \tag{4.6}$$

Where $\prod_{columns}$ indicates a product that runs along all elements in each column of the final matrix, $\mathbb{1}_{i\times j}$ is a matrix of 1s of size $i \times j$. $\boldsymbol{T}$ and $\boldsymbol{K}$ are two constant matrices of 0s and 1s. In particular, $\boldsymbol{T}$ is an upper triangular matrix with inverted columns, whereas $\boldsymbol{K}$ can be generated procedurally for any value of $rf_{\max}$, by repeating a pattern of 0s and 1s. Algorithm 2 details how $\boldsymbol{K}$ is generated. Figure 4.7 shows an example of these two matrices and of the entire transformation, for the case $rf_{max} = 9$. Also here γ_0 has been directly replaced with its constant value 1.

4.3.3 PIT Regularizer

To promote sparsity, the loss function of PIT is augmented with an L_1-Regularization term on $\boldsymbol{\gamma}$ vectors. As MorphNet does, it is possible to

Algorithm 2 K matrix generation

procedure K-GEN(L)
    Define *seed-matrix* **as** $\mathbb{1}_{1\times 1}$
    $\boldsymbol{K} \leftarrow$ *seed-matrix*
    for $i \leftarrow 1, \ldots,$ L **do**
        Add row of 0_s to $\boldsymbol{K}$
        Add column of $(2^i - 1)$ leading 0_s and one trailing 1 to $\boldsymbol{K}$
        Add row of 0_s to *seed-matrix* and append it to $\boldsymbol{K}$
        seed-matrix $\leftarrow \boldsymbol{K}$
    end for
end procedure

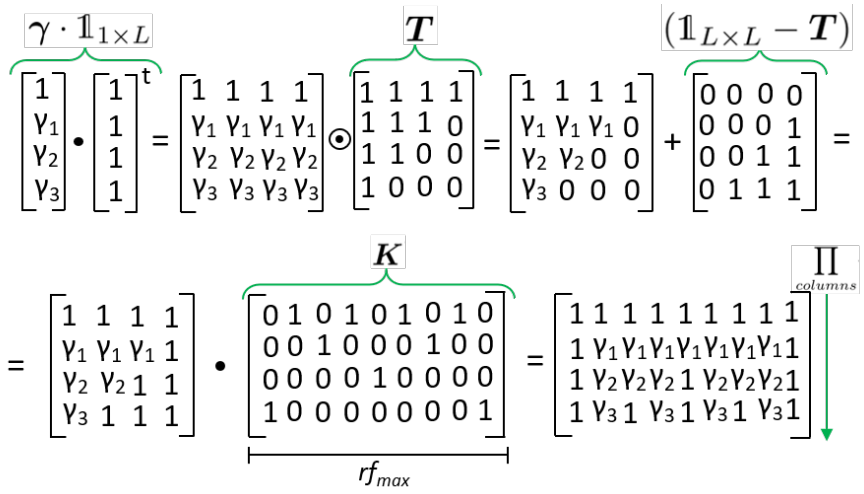

Figure 4.7: Generation of the **M** mask vector with differentiable operations. Example for $rf_{max} = 9$.

multiply the regularization term with a factor that takes into account the measure of a certain metric to be optimized. Here we consider as target metric the model size, as the final aim is having small hardware-friendly architecture, but the method is easily extendable to other target metrics such as FLOPs or latency. The specific form of the PIT-regularizer, that promotes small networks, is the following :

$$\mathcal{L}_R^{size}(\boldsymbol{\gamma}) = \lambda \sum_{l=1}^{layers} C_{in}^{(l)} \cdot C_{out}^{(l)} \sum_{i=1}^{L-1} \text{round}\left(\frac{rf_{\max} - 1}{2^{L-i}}\right) |\hat{\gamma}_i^{(l)}| \tag{4.7}$$

Where λ controls the strength of the regularization, l denotes the l^{th} layer. $C_{in}^{(l)}$ and $C_{out}^{(l)}$ are the number of input/output channels in the l^{th} layer. The last term, round $\left(\frac{rf_{\max}-1}{2^{L-i}}\right)$ is the number of non-masked kernel time-slices added by each non-zero $\gamma_i^{(l)}$.

The total loss function employed by PIT during the NAS phase is :

$$\mathcal{L}_{PIT}(\boldsymbol{W}, \boldsymbol{\gamma}) = \mathcal{L}^{perf}(\boldsymbol{W}) + \mathcal{L}_R^{size}(\boldsymbol{\gamma}) \tag{4.8}$$

Where $\mathcal{L}^{perf}(\boldsymbol{W})$ denotes the loss term related to the TCN's performance, which depends on the specific task considered (e.g., *LogCosh* in the Dalia benchmark).

4.3.4 Training

Algorithm 3 summarizes PIT's training procedure, which is composed of three different loops : *Warmup*, *Pruning* and *Fine-Tuning*.

The first phase is the warmup, it is characterized by at least $Steps_{wu}$ iterations of the training loop, with all the $\boldsymbol{\gamma}$ vectors frozen to 1 and $\mathcal{L}_{PIT}(\boldsymbol{W}, \boldsymbol{\gamma}) \equiv \mathcal{L}^{perf}(\boldsymbol{W})$.

The next step is the pruning loop, which starts from the pre-trained state reached during warmup. Here both the weights $\mathbf{W}$ and the $\boldsymbol{\gamma}$ vectors are updated with the complete loss of equation 4.8. This training phase does not present a predefined number of epochs but it continues until convergence by means of an early-stop mechanism, which stops the loop when the loss does not increase anymore on the validation set.

The NAS algorithm ends with a fine-tuning, made of $Steps_{ft}$ steps. All the $\boldsymbol{\gamma}$ vectors are frozen at the values learned during the previous loop, thus a specific architecture in the search space is considered. As warmup, fine-tuning uses only the $\mathcal{L}^{perf}(\boldsymbol{W})$ loss component. This last step is necessary to adapt all weights to the specific architecture found, significantly improving the final performances of the learned network.

4.4 Further Improvements

Figure 4.8 presents the effect of two improvements, the *post-processing* and *fine-tuning*, performed on the TimePPG architectures produced by MorphNet starting from TEMPONet. As said, these two additional steps are not part of the MorphNet algorithm or other NASes, but they allow to obtain DNNs that outperform other DL approaches on the Dalia dataset and for this reason are here presented.

The *post-processing* is applied to the output of the TCN, which produces an Heart-Rate estimation for every time-window considered at the input. This step is motivated by the fact that fully data-drive models such as DNNs sometimes make errors when inputs deviate from the distributions seen during the training phase. The resultant effect is that some Heart-Rate predictions are not compatible with human physiology, in particular too steep variations. For this reasons it is useful to smooth the overall prediction with a filtering technique. Specifically, a limit is imposed on the maximum relative Heart-Rate variation over time. The adopted filtering approach is very simple : the latest prediction is compared with the arithmetic mean

Algorithm 3 PIT - Pruning in Time

procedure PIT($\mathcal{L}_{perf}, \mathcal{L}_{PIT}, \boldsymbol{\theta}, \boldsymbol{\gamma}, Steps_{wu}, Steps_{ft}$)

▷ $\mathcal{L}_{perf}$ is the loss function used to train the model
▷ $\mathcal{L}_{PIT}$ is the sum of $\mathcal{L}_{perf}$ and the regularization loss
▷ $\boldsymbol{\theta}$ is the set of weights to be optimized
▷ $\boldsymbol{\gamma}$ is the set of masking parameters used to learn dilations
▷ $Steps_{wu}$ is the number of training iterations executed during warm-up
▷ $Steps_{ft}$ is the number of training iterations executed during fine-tuning

▷ Execution :

for $i \leftarrow 1, \ldots, Steps_{\text{wu}}$ **do** ▷ Warmup Loop
    Update $\boldsymbol{\theta}$ based on $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{perf}(\boldsymbol{\theta})$
end for
while not converged **do** ▷ Pruning Loop
    Update $\boldsymbol{\theta}$ and $\boldsymbol{\gamma}$ based on $\nabla_{\boldsymbol{\theta},\boldsymbol{\gamma}} \mathcal{L}_{PIT}(\boldsymbol{\theta}, \boldsymbol{\gamma})$
end while
for $i \leftarrow 1, \ldots, Steps_{\text{ft}}$ **do** ▷ Fine-Tuning Loop
    Update $\boldsymbol{\theta}$ based on $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{perf}(\boldsymbol{\theta})$
end for
end procedure

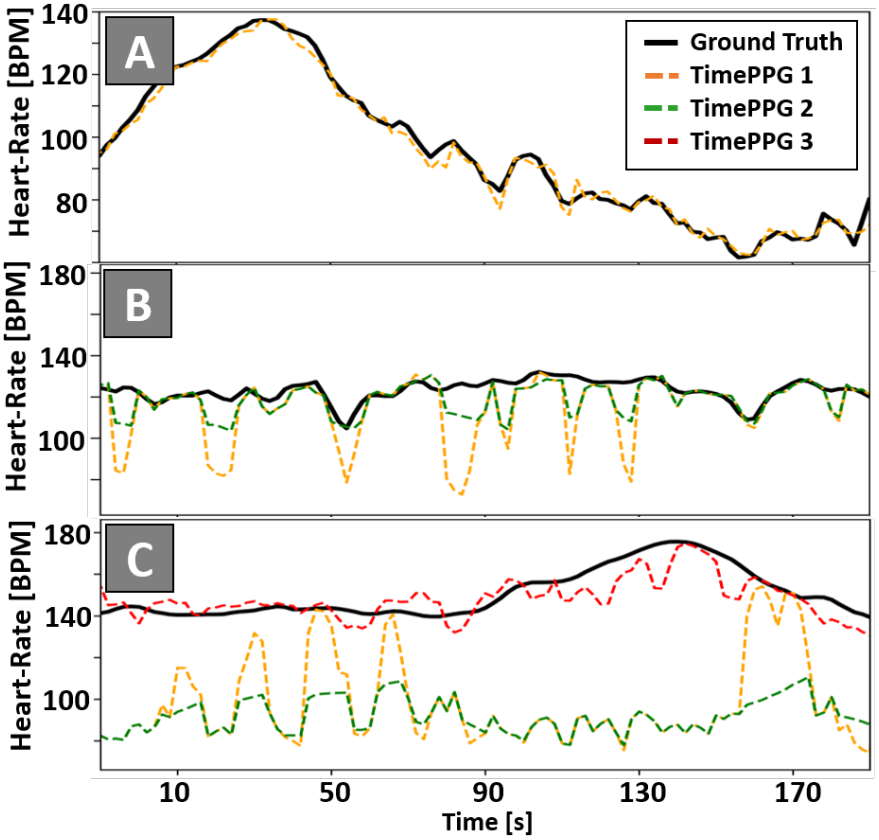

Figure 4.8: In subfigure A, the prediction of TimePPG on 200 seconds of subject 7. In B and C, traces from subj. 8 and 5, showing the benefits of the postprocessing and of fine-tuning. TimePPG 1 is the plain TCN, TimePPG 2 includes the post-processing and TimePPG 3 integrates both the post-processing and the fine-tuning.

$\overline{HR}_N$ of the previous N predictions. When the difference between them is larger than a certain threshold δ, the actual prediction is clipped to $\overline{HR}_N \pm \delta$. Good values for N and δ are found to be 10 and 10% of the mean. The effect of post-processing can be appreciated in Figure 4.8-B.

In the DL jargon *fine-tuning* denotes an additional, usually fast, training executed after the main one. The Dalia dataset presents an outlier subject (S5), characterized by higher mean values of the Heart-Rate with respect to the other fourteen subjects. For this reason TimePPG TCNs fails in predicting the Heart-Rate of this subject, affecting the overall MAE. In

figure 4.8-C it is shown how the plain TCN (TimePPG 1) and the TCN with post-processing (TimePPG 2) fail in predicting Heart-Rate values greater than 140 BPM. This problem affects overall performances of the TCN and is a side effect of data scarcity. By means of fine-tuning it is possible to demonstrate this assumption and show how the performance would improve if more data were available.

Specifically, fine-tune training is performed with a low learning rate, freezing the weights of the first convolutional block, on the initial portion of data, the 25%, relative to the specific subject used as test set, then the remaining 75% is used to test the fine-tuned network. The proposed approach is hardly feasible on the field because it would require a new training for each specific subject wearing the wrist-watch. The main limitations behind a new in-field training are twofold. From a side, the necessity of specific training hardware (i.e., GPUs). From the other, a means to collect ground-truth target (i.e., an ECG apparatus). Therefore, as said before, this last fine-tuning step is only an expedient to demonstrate how our model would benefit of more data.

Chapter 5

Experimental Results

In this chapter the effectiveness of the two methods proposed in chapter 4 is tested. In section 5.1 the experimental setup is presented, then sections 5.2 and 5.3 describe the architectures and results found respectively with TimePPG and PIT frameworks, presenting a state-of-the-art comparison and the embedded deployment on a commercial MCU. Finally, section 5.4 explores the orthogonality of the two NASes.

5.1 Experimental Setup

As anticipated, the two inputs of both the NASes are the TEMPONet [16] TCN, used as seed architecture and the Dalia [17] dataset. The following training details are common to both the methods:

- The obtained models are validated by means of the cross-validation scheme proposed in [17].
- TCNs are trained with Adam [54] as optimizer, using 1e-3 as learning rate, batch size equal to 128 and 500 epochs, with an early-stop with patience equal to 50.
- All the developed code is written in Python 3.6 and TensorFlow 1.14 [55].
- All the training has been performed on a single *NVIDIA-GTX 1080Ti* GPU.

5.1.1 GAP8 Hardware Platform

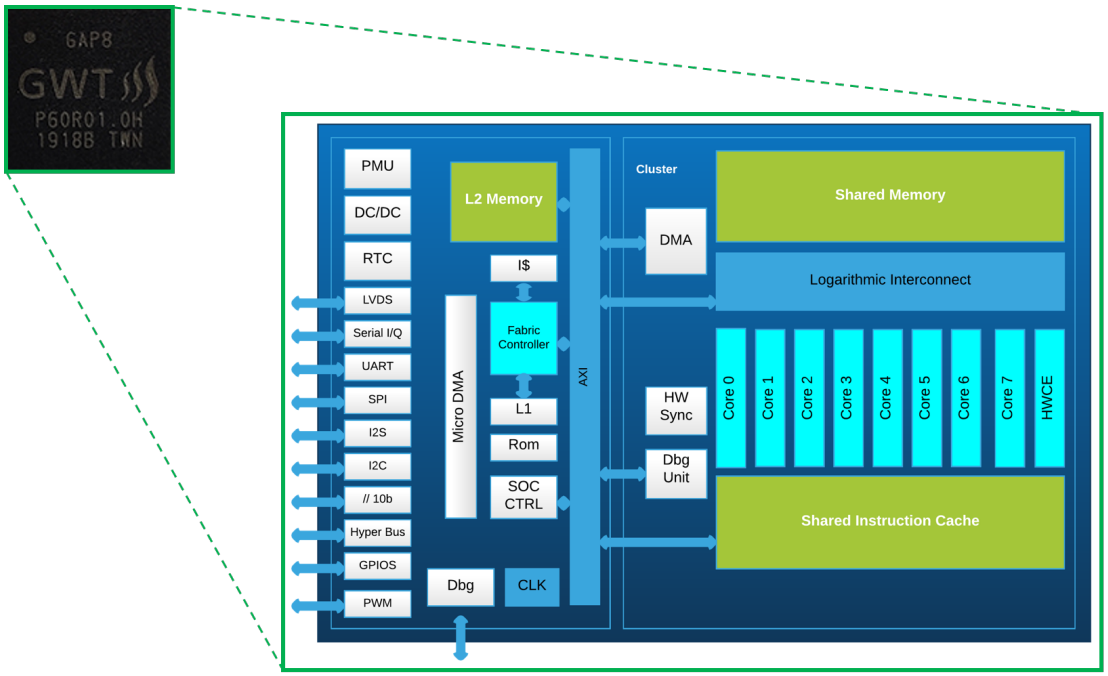

Figure 5.1: GAP8 Chip and Layout [52].

The obtained models have been deployed on a real commercial edge-computing platform : the Green-Waves Technologies' GAP8 SoC [18], depicted in figure 5.1. This allows to test and measure the effective latency and energy consumption of the architectures found by the NAS tools.

GAP8 is based on the Parallel Ultra Low Power (PULP) platform [56], an open-source hardware platform for digital architectures based on the RISC-V Instruction Set. It includes a cluster of 8 computing cores with enhanced digital signal processing and Machine Learning (ML) features. These 8 cores are controlled and activated by another computational unit denoted as the *Fabric Controller*, which manages communication and orchestrates computing. The memory hierarchy of GAP8 includes two levels : 64 kB of L1 single clock-cycle memory and 512 kB L2 memory. An additional L3 level can be added off-chip. Moreover, two Direct Memory Access (DMA) controllers are present, which allows to move data efficiently.

GAP8 does not include a Floating Point Unit (FPU), thus the models have to be deployed with a quantized-int8 format. GreenWaves' proposes a

proprietary neural network deployment flow, called NN-Tool, which has been used to deploy the found TCNs. All the presented data has been obtained with a core frequency of 100 MHz.

5.2 TimePPG Design Space Exploration Results

Figure 5.2 shows models obtained with MorphNet, starting from TEMPONet as seed network, in the MAE vs Network Size and MAE vs FLOPs spaces.

The values of MAE reported refer to *TimePPG 1*, i.e., the plain output of the network without post-processing and fine-tuning. Only models characterized by a MAE lower than 7 BPM are shown. The different points of figure 5.2

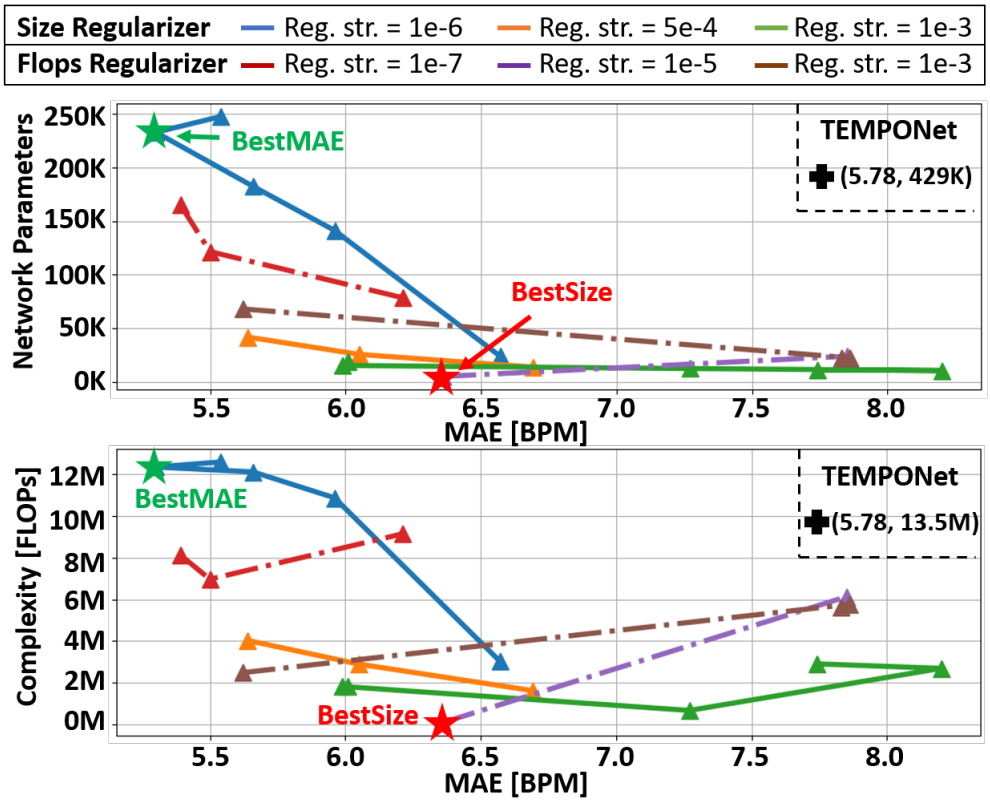

Figure 5.2: TimePPG Pareto charts in the MAE vs. size and MAE vs. FLOPs spaces

are obtained exploiting both the size and FLOPs regularizers, tweaking the

regularization strength and the pruning threshold with a grid-search scheme. In particular, for each one of the values of regularization strength indicated in figure 5.2, five values of threshold are used; namely th = [1.0e-2, 2.5e-2, 5.0e-2, 7.5e-2, 1.0e-1].

Figure 5.2 also reports the baseline values of MAE, size and FLOPs of TEMPONet, which achieves 5.78 BPM MAE at the cost of 429k parameters and 13.5 MFLOPs. The seed TEMPONet represented in figure 5.2 is not to-scale and is reported only for comparison. The same point, if represented to-scale, would be much more higher in the right part of the plot. This shows clearly how the original seed network is Pareto dominated by our results.

The explored design space spans from 5k to 230k network parameters and from 0.1M to 12M FLOPs, hence producing a large variety of networks with different peculiarities. Among the large number of different architectures produced, two Pareto-optimal points are highlighted : BestSize and BestMAE, depicted in figure 5.2 respectively with red and green stars. The former is the TCN that achieves the smallest size, which is obtained by means of the FLOPs regularizer with a regularization strength of 1.0e-5 and a threshold of 1.0e-2. BestSize network presents only 5.09k parameters (84× compression) and 86,658 FLOPs (151× compression) at the cost of an increased MAE of 6.29 BPM (0.51 increase), where relative results in parentheses are with respect to the baseline TEMPONet. Instead, BestMAE represents the other end of the Pareto-chart, i.e., the network with lowest MAE, obtained with the size regularizer and a strength-threshold combination of 1.0e-6 and 2.5e-2. BestMAE network presents an optimal MAE of 5.3 BPM (0.48 decrease) with 230k parameters (1.87× compression) and 12.3 MFLOPs (1.1× compression), again with respect to TEMPONet.

5.2.1 State-of-the-Art Comparison

Table 5.1 shows a comparison between TimePPG-BestMAE TCN and other state of the art methods on the Dalia dataset. The results are detailed for every subject S included in the dataset. TimePPG-BestMAE results are detailed both in raw form and with the improvements proposes in 4.4, i.e., post-processing and fine-tuning. The table reports both DL and classical algorithms.

The only DL algorithm detailed in 5.1 is the CNN proposed in [17] (in table STFT+CNN), which achieves a mean MAE of 7.65 BPM. Hence, TimePPG-BestMAE even without post-processing and fine-tuning outperforms this

approach. Moreover, the CNN presented in [17] is 260× larger than our TCN. This makes it nearly impossible to deploy on a real embedded device, which is, conversely, extremely feasible by means of our network composed of 230k parameters.
In literature, another DL approach can be found, which is based on Generative Adversarial Networks (GANs) and proposed in [57] (not presented in table 5.1 because the paper does not report the results on individual subjects), which achieves a mean MAE of 8.3 BPM. Therefore, also the GAN approach is outperformed by TimePPG-BestMAE.

All the other rows of 5.1 refer to classical methods, based on adaptive filtering and peak tracking. Schack2017 [58] and SpaMaPlus [59] are originally proposed for another PPG dataset, IEEE Training [31], which is much smaller than Dalia. Instead, CurToSS [60] and TAPIR [32] are tailored to Dalia and significantly outperform the two other methods detailed above. This is indicative of one of the main drawback of this type of methods : the strong correlation between the algorithm hyper-parameters and the underlying dataset, which means, that these algorithms are very prone to overfitting. TimePPG-BestMAE outperforms both Schack2017 and SpaMaPlus, and, when combined with post processing achieves comparable results with respect to TAPIR (with a slightly higher MAE of 0.31 BPM) and outperforms CurToSS (with a slightly higher MAE of −0.16 BPM).

The TCN fails in outperforming TAPIR and CurToSS, mainly due to the presence of S5, which is an outlier with an average hear-rate much higher than other subjects. The MAE of S5 is the highest and it affects a lot the mean MAE used in the comparison. This behavior is observable also in the other DL algorithm based on CNNs.

Thanks to fine-tuning, it is possible to mimic a wider training dataset, alleviating the outlier problem, whose MAE decreases from 14.68 BPM to 4.88 BPM. The fine-tuned version of TimePPG-BestMAE effectively outperforms all the other state-of-the-art algorithms that address PPG-based heart-rate monitoring on the Dalia dataset.

Table 5.1: Comparison of TimePPG-BestMAE with state-of-the-art PPG based heart rate monitoring algorithms. The p-value reported is computed with non-parametric Mann-Whitney statistic.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Mean	p-value
Schack2017 [58]	33.1	27.8	18.5	28.8	12.6	8.7	20.65	21.8	22.3	12.6	21.1	22.8	27.7	12.1	16.4	20.5	< 0.01
SpaMaPlus [59]	8.86	9.67	6.40	14.10	24.06	11.34	6.31	11.25	16.04	6.17	15.15	12.03	8.50	7.76	8.29	11.06	< 0.01
STFT+CNN [17]	7.73	6.74	4.03	5.90	18.51	12.88	3.91	10.87	8.79	4.03	9.22	9.35	4.29	4.37	4.17	7.65	< 0.01
TAPIR [32]	4.50	4.50	3.20	6.00	5.00	3.40	2.80	6.30	8.00	2.90	5.10	4.70	3.10	5.00	4.10	4.57	< 0.01
CurToSS [60]	5.40	4.30	3.00	8.00	**2.20**	**2.80**	3.30	8.50	12.60	3.60	**3.60**	6.10	3.00	5.50	3.70	5.04	0.02
TimePPG-BestMAE	4.51	3.37	2.33	5.25	14.68	4.76	2.37	8.04	8.75	3.30	5.19	8.08	**2.29**	3.02	**3.49**	5.30	n.a.
+ Post-Processing	4.01	3.16	**2.27**	4.62	14.96	4.28	2.58	6.02	7.61	**2.89**	4.79	6.95	2.54	**3.01**	3.56	4.88	n.a.
+ Fine Tuning	**3.17**	**2.74**	3.13	**4.25**	4.88	3.7	**2.48**	**5.19**	**7.00**	3.47	3.67	**3.91**	2.85	3.55	3.6	**3.84**	n.a.

5.2.2 Embedded Deployment

Table 5.2: Deployment of TimePPG solutions on GAP8 SoC and comparison with original TEMPONet without dilation and hand-tuned (h.-t.) dilation.

	# Weights	MAE [BPM]	Latency [ms]	Energy [mJ]
TEMPONet dil=1	939k	**5.08**	112.6	29.5
TEMPONet dil=h.-t.	423k	5.31	58.8	15.4
TimePPG-BestSize	**5k**	6.36	**1.62**	**0.53**
TimePPG-BestMAE	232k	5.3	20.2	4.24

Table 5.2 reports a comparison between the deployment results of the two best TimePPG architectures (i.e., BestSize and BestMAE) and the two baseline TCNs (i.e., TEMPONet without dilation and with hand-tuned dilation). As anticipated, the networks are deployed on the 8-core cluster of GAP8. For this deployment, all TCNs are quantized to INT8 data format.

Both TimePPG's variants are notable of mention:

- TimePPG-BestSize presents an higher MAE (+19.8%) compared to the seed TEMPONet, but it reduces size (-98.8%), latency (-97.2%) and energy (-95.65%).
- TimePPG-BestMAE improves all the metrics compared to the seed network, in fact it reduces MAE (-0.2%), size (-45.2%), latency (-82.1%) and energy (-72.5%).

5.3 PIT Design Space Exploration Results

Starting from TEMPONet, the PIT NAS has been applied to explore the design space, with respect to the dilation hyper-parameter of each temporal convolutional layer of the seed network. Figure 5.3 shows the Pareto points (red points), in the MAE vs size plane, obtained during the exploration, by tweaking the regularization strength and the warmup duration. Moreover, for comparison the graph also reports in black the two baseline architectures, i.e., TEMPONet without dilation (dil=1) and the original TEMPONet with hand-tuned dilation factors. The search space is quite huge, spanning from 400k to 900k parameters with $\sim 10^4$ candidates architectures. Table 5.3 reports

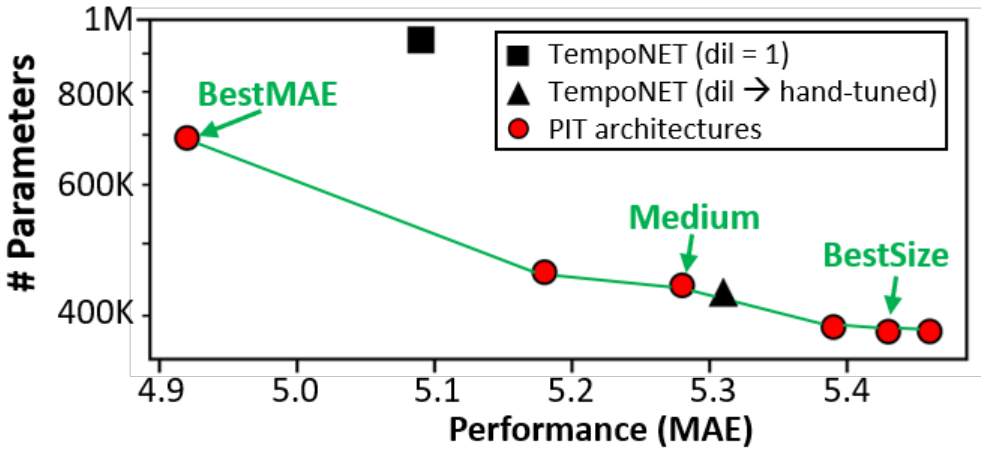

Figure 5.3: PIT Pareto chart in the MAE vs. size space.

Table 5.3: Dilation factors obtained for the different temporal convolutional layers of TEMPONet.

	PIT Dilations
TEMPONet dil=h.-t.	(2, 2, 1, 4, 4, 8, 8)
PIT-BestSize	(2, 4, 4, 8, 8, 16, 16)
PIT-Medium	(1, 2, 4, 2, 1, 8, 16)
PIT-BestMAE	(1, 1, 1, 1, 1, 1, 16)

the dilation factors obtained by PIT for three different architectures, i.e., the largest (PIT-BestMAE) the smallest (PIT-BestSize) and the closest in terms of parameters (PIT-Medium) to the original hand-tuned TEMPONet.

Figure 5.3 shows how PIT is able to compress and improve in performance the seed network (TEMPONet with dil=1, black square in figure), in particular the top-performing TCN is characterized by a MAE reduction of 0.16 BPM and a 1.35× compression. Instead, considering the original version of TEMPONet, with dilations tuned by experts, it is possible to see how this network is part of the Pareto-frontier, demonstrating the good quality of PIT in finding optimal architectures.

Table 5.4: Comparison between ProxylessNAS and PIT, with Dalia as dataset and TEMPONet as seed architecture.

	ProxylessNAS		PIT	
	# Weights	MAE [BPM]	# Weights	MAE [BPM]
BestSize	381k	5.43	381k	5.43
Medium	517k	5.21	440k	5.28
BestMAE	731k	5.15	694k	4.92

5.3.1 State-of-the-Art Comparison

In order to prove and understand the effectiveness of PIT, a comparison with another state-of-the-art NAS algorithm is here proposed. Specifically, the ProxylessNAS [11] has been selected for comparison, which is based on the supernet idea briefly presented in Chapter 3. The supergraph of ProxylessNAS is adapted to target all the same dilation factors that PIT supports.

Table 5.4 reports a comparison between the three BestSize, Medium and BestMAE architectures found by ProxylessNAS and PIT. It is possible to notice that both methods converge to the same small network. Instead, in the large case, PIT is able to find an architecture which is both more accurate (MAE : 4.92 BPM vs 5.15 BPM) and more compressed (size : 694k vs 731k parameters) than ProxylessNAS.

Figure 5.4 shows another comparison between PIT and ProxylessNAS based on the effective training time of the two NASes. The comparison is performed with the same setup and training details detailed in section 5.1. From the figure it is possible to see how ProxylessNAS is much more time consuming with respect to PIT, which is up to 10.4× faster. Moreover, PIT is also compared with a simple training of TEMPONet, without any NAS algorithm in the loop, showing the extremely low overhead of the proposed method. PIT results to be only 1.3×-2.3× slower than a plain training. This huge difference between the methods is motivated by the fact that ProxylessNAS trains only one arc of the supernet for each iteration of the training loop. PIT, instead, thanks to the DMaskingNAS approach trains all weights and all mask parameters together, with a considerable speed-up.

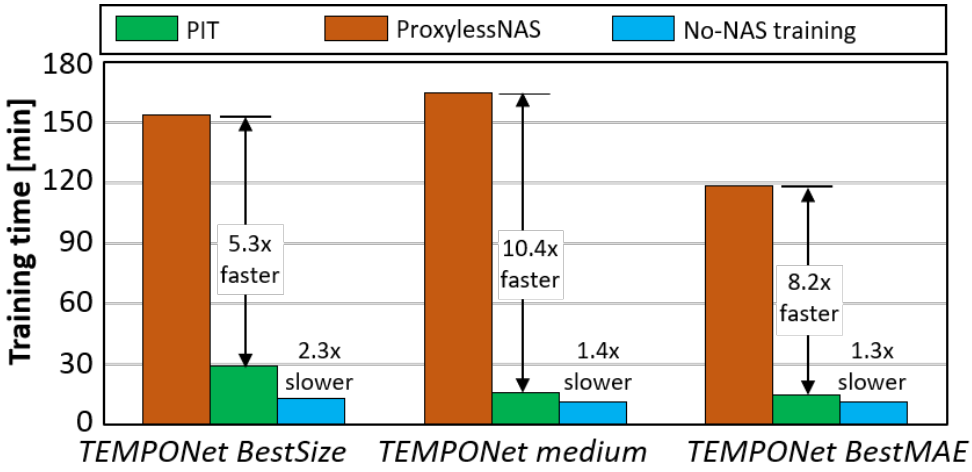

Figure 5.4: Comparison of training time between PIT, ProxylessNAS and a plain training on TEMPONet.

Table 5.5: Deployment of PIT solutions on GAP8 SoC and comparison with original TEMPONet without dilation and hand-tuned (h.-t.) dilation.

	# Weights	MAE [BPM]	Latency [ms]	Energy [mJ]
TEMPONet dil=1	939k	5.08	112.6	29.5
TEMPONet dil=h.-t.	423k	5.31	58.8	15.4
PIT-BestSize	**381k**	5.43	**54.8**	**14.4**
PIT-Medium	440k	5.28	59.8	15.7
PIT-BestMAE	693k	**4.92**	86.3	22.6

5.3.2 Embedded Deployment

Table 5.5 reports the deployment results of the two baseline TCNs (i.e., TEMPONet without dilation and with hand-tuned dilation) and the three identified PIT architectures (i.e., BestSize, Medium and BestMAE). Also in this case, the networks are deployed on the 8-core cluster of GAP8 and quantized to INT8 data format.

All the three PIT's variants are notable of mention:

- PIT-BestSize presents an higher MAE (+6.9%) compared to the seed TEMPONet, but it reduces size (-59.4%), latency (-51.3%) and energy (-51.2%).
- PIT-Medium is nearly equivalent to hand-tuned TEMPONet, with

respect to seed architecture (with dil=1) achieves higher MAE (+3.9%) and reduced size (-53.1%), latency (-46.9%) and energy (-46.8%).

- PIT-BestMAE improves all the metrics compared to the seed network, in fact it reduces MAE (-3.1%), size (-26.1%), latency (-23.4%) and energy (-23.4%).

5.4 MorphNet and PIT Orthogonality Exploration

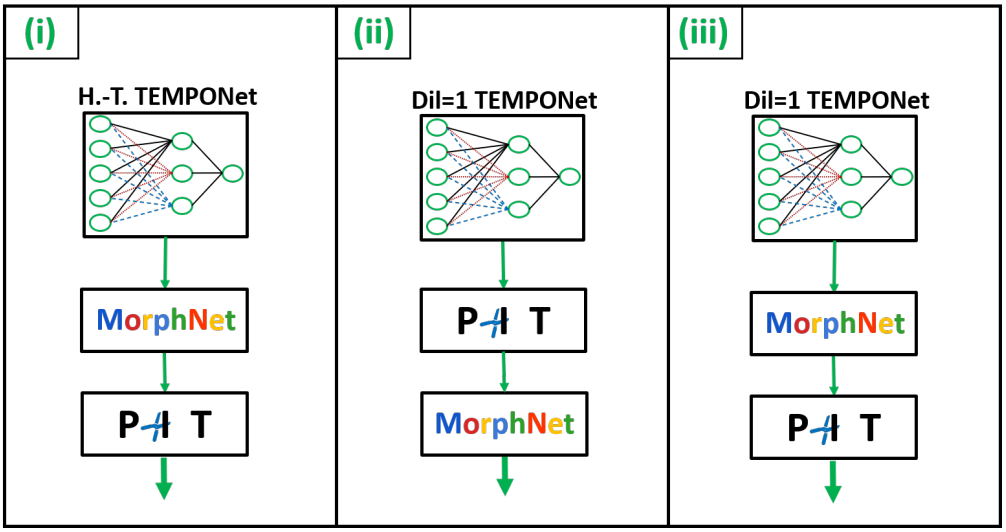

Figure 5.5: Three possible combinations of MorphNet and PIT algorithms.

In this last section the orthogonality of the two NASes, namely MorphNet (MN) and PIT, is tested in order to optimize both the number of channels and the dilations of each temporal convolution. Three combinations are possible :

(i) Apply MorphNet on hand-engineered TEMPONet, as done in Section 5.2, obtaining a family of architectures. Then, use MorphNet's output models as seeds for PIT.

(ii) Apply PIT on TEMPONet without dilations, as did in Section 5.3, obtaining a family of TCNs. Then use PIT's output architectures as seeds for MorphNet.

(iii) Apply MorphNet on TEMPONet *without dilations*, obtaining a family of networks. Then use MorphNet's output TCNs as seeds for PIT.

Figure 5.5 summarizes the previous three approaches.

5.4.1 H.-T. TEMPONet → MorphNet → PIT

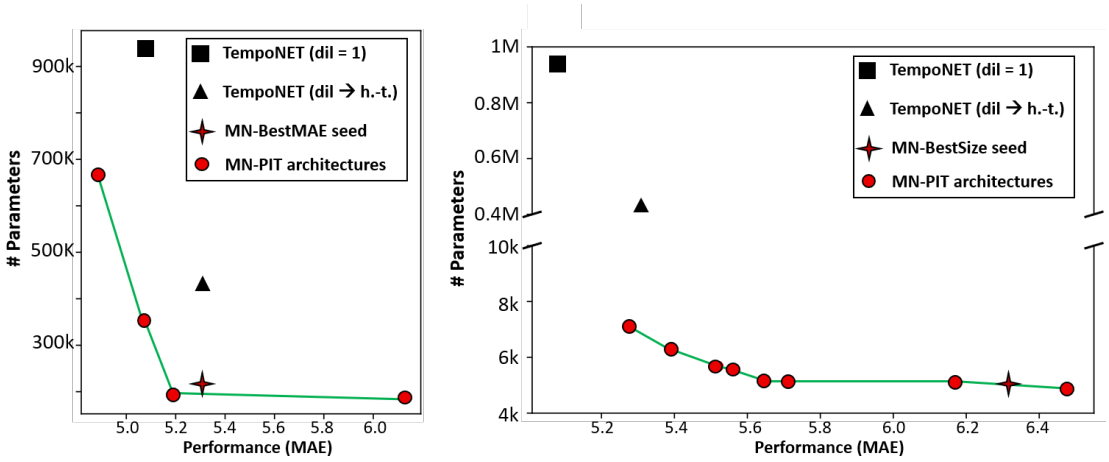

Figure 5.6: Pareto charts in the Performance vs. Number of Parameters space obtained using MN-BestMAE (left) and MN-BestSize (right) as seed networks for PIT.

Figure 5.6 shows two Pareto frontiers in the Performance vs. Number of Parameters space obtained applying the PIT workflow on the two size- and MAE-optimal architectures obtained with MorphNet. These two architectures are the ones presented in Section 5.2 as TimePPG-BestMAE and TimePPG-BestSize; here, they are denoted as MN-BestMAE and MN-BestSize and used as seeds for PIT.

The left part of figure 5.6 reports the Pareto architectures obtained using MN-BestMAE as seed. The obtained TCNs improve both the original TEMPONet variants without and with hand-tuned dilations, finding a new architecture (left-most point in the plot) that achieves the lowest MAE of all experiments, i.e., 4.88 BPM of MAE with 668k parameters. Therefore, this particular network, with respect to TEMPONet without dilations, achieves a compression of 28.9% with a MAE improvements of 3.9% . Moreover, MN-BestMAE is also improved; the minimum of the Pareto front improves MAE of 2.1% with 18.3% compression.

The right part of figure 5.6 reports the Pareto-optimal points obtained using MN-BestSize as seed. With this set-up a plethora of different extremely compressed TCNs are obtained. Notable of mention is the left-most point of the plot, which represents a network, that compared to hand-tuned TEMPONet, contains 98.4% less parameters along with a slightly better MAE (5.28 BPM). The seed network MN-BestSize is, in this case part of the Pareto frontier.

5.4.2 Dil=1 TEMPONet → PIT → MorphNet

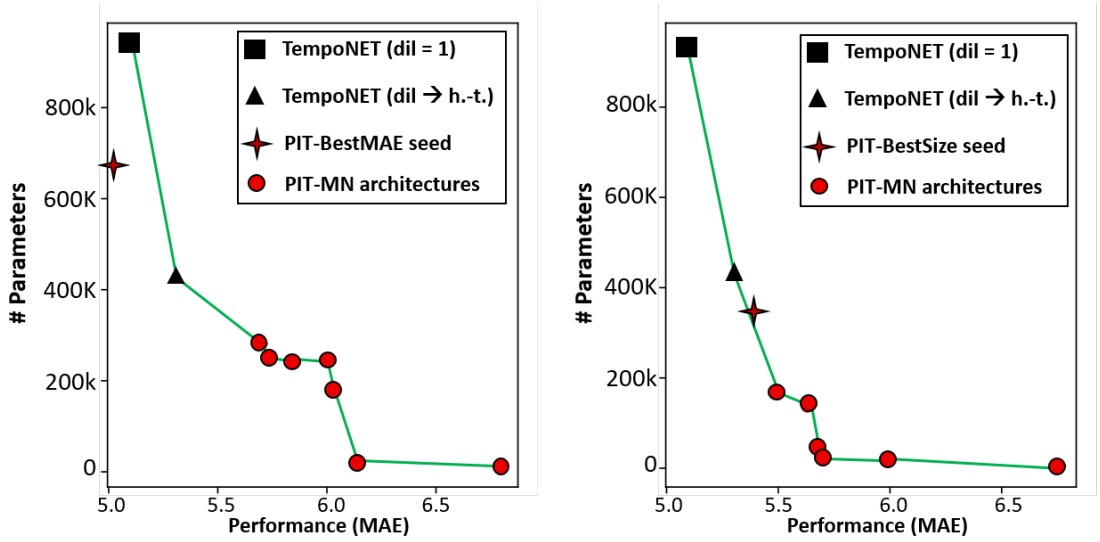

Figure 5.7: Pareto charts in the Performance vs. Number of Parameters space obtained using PIT-BestMAE (left) and PIT-BestSize (right) as seed networks for MorphNet.

Figure 5.7 shows the two Pareto frontiers in the Performance vs. Number of Parameters space obtained applying MorphNet on the size- and MAE-optimal architectures obtained with PIT. These two architectures are the ones presented in 5.3 as PIT-BestMAE and PIT-BestSize, here used as seeds architectures for MorphNet.

The left part of figure 5.7 reports the Pareto architectures obtained using PIT-BestMAE as seed. The obtained architectures fail in improving both the original TEMPONet variants without and with hand-tuned dilations. Moreover, also the seed networks is not improved.

The right part of figure 5.7 reports the Pareto-optimal points obtained using PIT-BestSize as seed. The seed network PIT-BestSize and the two TEMPONet flavors are part of the Pareto frontier, thus none of the found architectures are really capable of improving the seed ones.

This poor results finds a possible explanation in the fact that PIT operates a more refined research in the design landscape with respect to MorphNet. Therefore, the resultant search space obtained starting from PIT architectures as seeds for MorphNet is more constrained, leading to sub-optimal results.

5.4.3 Dil=1 TEMPONet → MorphNet → PIT

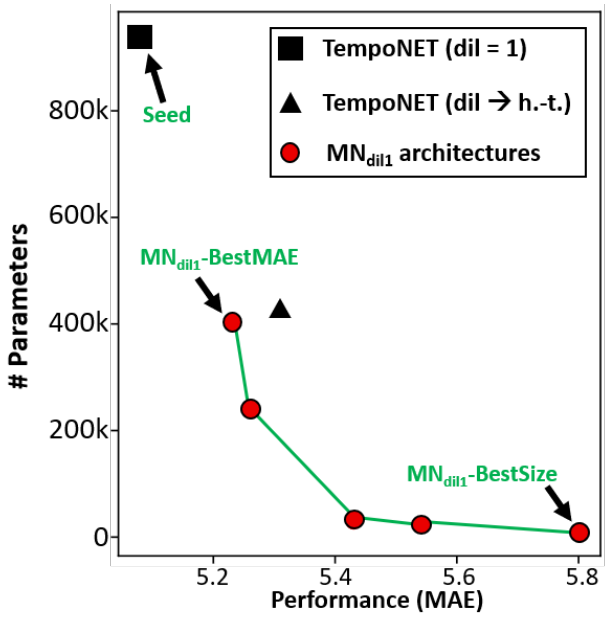

Figure 5.8: Pareto charts in the Performance vs. Number of Parameters space obtained with MorphNet using as seed TEMPONet with all dilation factors fixed to 1.

Figure 5.8 depicts the Pareto front obtained using the TEMPONet variants with all dilations fixed to 1 as seed of MorphNet. In the plot the two architectures $MN_{dil1}-BestMAE$ and $MN_{dil1}-BestSize$ are identified. Where the subscript *dil1* specify that the seed architectures are obtained applying MorphNet on the $dil = 1$ TEMPONet flavor.

$MN_{dil1}-BestMAE$ and $MN_{dil1}-BestSize$ are then used as seed networks for PIT. The obtained Pareto fronts are reported in figure 5.9.

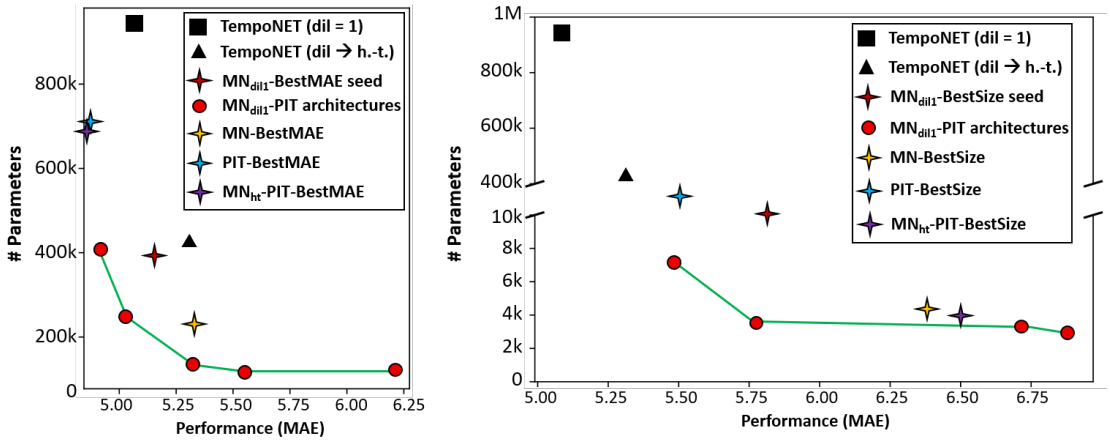

Figure 5.9: Pareto charts in the Performance vs. Number of Parameters space obtained using PIT with $MN_{dil1}-BestMAE$ (left) and $MN_{dil1}-BestSize$ (right) as seed networks.

The left part of figure 5.6 reports the Pareto architectures obtained using $MN_{dil1}-BestMAE$ as seed. The obtained TCNs improve both the original TEMPONet variants without and with hand-tuned dilations, finding a new architecture (left-most point in the plot) that achieves a MAE of 4.91 BPM with 405k parameters. Therefore, this particular network, with respect to TEMPONet without dilations, achieves a compression equal to 55.7% with a MAE improvements of 3.3% . Moreover, $MN_{dil1}-BestMAE$ is also improved; the second left-most Pareto-point improves MAE of 3.8% with 38.5% compression.

The right part of figure 5.6 reports the Pareto-optimal points obtained using $MN_{dil1}-BestSize$ as seed. With this set-up an ensemble of different, extremely compressed, TCNs are obtained. Notable of mention is the second left-most point of the Pareto frontier, which represents a network, that compared to the seed $MN_{dil1}-BestSize$, presents 68.7% less parameters with a slightly better MAE.

Moreover, figure 5.9 reports the others *BestMAE* (left) and *BestSize* (right) architectures obtained with methods presented in sections 5.2, 5.3 and 5.4.1. The results obtained in Section 5.4.2 are not reported in that they do not provide improvements. The two plots highlight how the approach described in this section is the best one in finding Pareto-optimal architectures, exploiting the -now demonstrated- orthogonality of the two NAS algorithms : MorphNet

and PIT.

5.4.4 Embedded Deployment

Table 5.6: Deployment of MN_{dil1}-PIT solutions on GAP8 SoC and comparison with original TEMPONet without dilation and hand-tuned (h.-t.) dilation.

	# Weights	MAE [BPM]	Latency [ms]	Energy [mJ]
TEMPONet dil=1	939k	5.08	112.6	29.5
TEMPONet dil=h.-t.	423k	5.31	58.8	15.4
MN_{dil1}-PIT-BestSize	**3k**	5.77	**0.88**	**0.23**
MN_{dil1}-PIT-BestMAE	405k	**4.91**	36.5	9.57

Table 5.6 reports the deployment results of the two baseline TCNs (i.e., TEMPONet without dilation and with hand-tuned dilation) and the BestSize and BestMAE architectures obtained with the flow presented in the previous section 5.4.3 (TEMPONet dil=1 is the seed network). The deployment is performed with the same set-up of section 5.2.2 and section 5.3.2.

Both variants provides interesting results:

- MN_{dil1}-PIT-BestSize presents an higher MAE (+13.6%) compared to the seed TEMPONet, but it reduces extremely size (-99.7%), latency (-99.2%) and energy (-99.1%).
- MN_{dil1}-PIT-BestMAE improves all the metrics compared to the seed network, in fact it reduces MAE (-3.3%), size (-56.9%), latency (-67.6%) and energy (-67.5%).

Chapter 6

Conclusions and Future Works

Deep Learning algorithms are among the most intriguing and promising technologies proposed in recent years and are today applied on a plethora of different tasks. For many years the common trend in DL has been having more and more complex models able to solve problems of increasing difficulties, requiring an huge amount of resources to effectively execute them. DL models "as-they-are" are confined to high-performance computing platforms, making it impossible to deploy them on embedded devices with constrained memory and computational power. A new research perspective was born in parallel to this continuous increase of models' complexity, targeting the efficient execution of DNNs on edge devices. The primary way to enable the so-called *inference-on-the-edge* is represented by somehow shrinking the original, huge DL architectures.

This work proposes a workflow that allows to quickly and efficiently compress DNNs, starting from a large one. In particular, resorting to NAS techniques. Two different NAS algorithms are here used and presented : MorphNet and PIT. While the former is present in literature and originally developed for CNNs and here applied to TCNs, the second one is a novel NAS approach that targets a specific hyper-parameter of such TCNs, the dilation factor. The effectiveness of both methods, used both in isolation and jointly, is demonstrated on a bio-signals processing task, highlighting the obtained compressed architectures, which are then deployed on a real edge device, namely the GAP-8 SoC. In all the tested solutions state-of-the-art architectures are found, that effectively compress the seed network up to

99.6%.

Future works include the extension of PIT to consider other hyper-parameters such as the kernel-size, in order to target the DNNs' compression problem from multiple directions, enlarging the search space of the proposed NAS algorithm.

Bibliography

[1] Luciano Floridi. *The fourth revolution: How the infosphere is reshaping human reality*. OUP Oxford, 2014 (cit. on p. 1).

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. «Efficient processing of deep neural networks: A tutorial and survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329 (cit. on pp. 1, 2).

[3] Zachary C Lipton, John Berkowitz, and Charles Elkan. «A critical review of recurrent neural networks for sequence learning». In: *arXiv preprint arXiv:1506.00019* (2015) (cit. on p. 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. 2).

[5] Yann LeCun, John Denker, and Sara Solla. «Optimal brain damage». In: *Advances in neural information processing systems* 2 (1989), pp. 598–605 (cit. on pp. 2, 3).

[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 pp. 2, 3).

[7] Song Han, Huizi Mao, and William J Dally. «Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding». In: *arXiv preprint arXiv:1510.00149* (2015) (cit. on pp. 2, 3).

[8] 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).

[9] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Quantized neural networks: Training neural networks with low precision weights and activations». In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898 (cit. on p. 3).

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. «Darts: Differentiable architecture search». In: *arXiv preprint arXiv:1806.09055* (2018) (cit. on pp. 3, 28).

[11] Han Cai, Ligeng Zhu, and Song Han. «Proxylessnas: Direct neural architecture search on target task and hardware». In: *arXiv preprint arXiv:1812.00332* (2018) (cit. on pp. 3, 27, 29, 57).

[12] 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 pp. 3, 27, 29, 32, 33, 36).

[13] Alvin Wan et al. «Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 12965–12974 (cit. on pp. 3, 29).

[14] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. «An empirical evaluation of generic convolutional and recurrent networks for sequence modeling». In: *arXiv preprint arXiv:1803.01271* (2018) (cit. on pp. 3, 22).

[15] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. «Temporal convolutional networks: A unified approach to action segmentation». In: *European Conference on Computer Vision*. Springer. 2016, pp. 47–54 (cit. on p. 3).

[16] 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 (2019), pp. 244–256 (cit. on pp. 3, 33, 34, 49).

[17] 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), p. 3079 (cit. on pp. 4, 22, 24, 33, 49, 52–54).

[18] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. «GAP-8: A RISC-V SoC for AI at the Edge of the IoT». In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–4 (cit. on pp. 4, 50).

[19] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019 (cit. on pp. 5, 6, 9, 11, 12, 14, 17).

[20] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997 (cit. on p. 6).

[21] R.A. Fisher. *Iris Data Set*. 1936. URL: `https://archive.ics.uci.edu/ml/datasets/iris` (cit. on p. 6).

[22] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. Vol. 1. MIT press Massachusetts, USA: 2017 (cit. on pp. 7, 10, 13, 17, 20).

[23] 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 p. 9).

[24] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017 (cit. on p. 12).

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985 (cit. on p. 15).

[26] Vadim Lebedev and Victor Lempitsky. «Fast convnets using group-wise brain damage». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2554–2564 (cit. on p. 17).

[27] Rodolphe Jenatton, Jean-Yves Audibert, and Francis Bach. «Structured variable selection with sparsity-inducing norms». In: *The Journal of Machine Learning Research* 12 (2011), pp. 2777–2824 (cit. on p. 17).

[28] 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 p. 17).

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Communications of the ACM* 60.6 (2017), pp. 84–90 (cit. on p. 18).

[30] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on pp. 20, 34).

[31] 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 (2014), pp. 522–531 (cit. on pp. 24, 53).

[32] 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 pp. 24, 53, 54).

[33] 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 (cit. on p. 24).

[34] Xiangmao Chang, Gangkai Li, Linlin Tu, Guoliang Xing, and Tian Hao. «DeepHeart: Accurate Heart Rate Estimation from PPG Signals Based on Deep Learning». In: *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE. 2019, pp. 371–379 (cit. on p. 24).

[35] Pat Hamilton. «Open source ECG analysis». In: *Computers in cardiology*. IEEE. 2002, pp. 101–104 (cit. on p. 24).

[36] Empatica. *Empatica E4 wristband*. URL: `https://www.empatica.com/en-eu/research/e4/` (cit. on p. 24).

[37] RespiBAN. *RespiBAN Professional*. URL: `https://biosignalsplux.com/products/wearables/respiban-pro.html` (cit. on p. 24).

[38] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. «Once-for-all: Train one network and specialize it for efficient deployment». In: *arXiv preprint arXiv:1908.09791* (2019) (cit. on p. 27).

[39] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. «Mcunet: Tiny deep learning on iot devices». In: *arXiv preprint arXiv:2007.10319* (2020) (cit. on p. 27).

[40] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. «Designing neural network architectures using reinforcement learning». In: *arXiv preprint arXiv:1611.02167* (2016) (cit. on p. 27).

[41] Barret Zoph and Quoc V Le. «Neural architecture search with reinforcement learning». In: *arXiv preprint arXiv:1611.01578* (2016) (cit. on p. 27).

[42] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. «Large-scale evolution of image classifiers». In: *arXiv preprint arXiv:1703.01041* (2017) (cit. on p. 27).

[43] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. «Efficient neural architecture search via parameter sharing». In: *arXiv preprint arXiv:1802.03268* (2018) (cit. on p. 28).

[44] Eric Jang, Shixiang Gu, and Ben Poole. «Categorical reparameterization with gumbel-softmax». In: *arXiv preprint arXiv:1611.01144* (2016) (cit. on p. 29).

[45] Apple. *Apple Watch Series*. URL: `https://www.apple.com/lae/watch/` (cit. on p. 32).

[46] Fitbit. *Fitbit Charge 4*. URL: `https://www.fitbit.com/global/us/products/trackers/charge4` (cit. on p. 33).

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 34).

[48] Google-Research. *MorphNet*. URL: `https://github.com/google-research/morph-net` (cit. on p. 36).

[49] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «Mobilenets: Efficient convolutional neural networks for mobile vision applications». In: *arXiv preprint arXiv:1704.04861* (2017) (cit. on p. 37).

[50] Ming Yuan and Yi Lin. «Model selection and estimation in regression with grouped variables». In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1 (2006), pp. 49–67 (cit. on p. 37).

[51] ST Microelectornics. *X-CUBE-AI*. 2017. URL: https://www.st.com/en/embedded-software/x-cube-ai.html (cit. on p. 40).

[52] GreenWaves Technologies. *GAP8 NNTool*. 2019. URL: https://greenwaves-technologies.com/manuals/ (cit. on pp. 40, 50).

[53] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1». In: *arXiv preprint arXiv:1602.02830* (2016) (cit. on p. 42).

[54] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 49).

[55] Martin Abadi et al. «Tensorflow: A system for large-scale machine learning». In: *12th symposium on operating systems design and implementation*. 2016, pp. 265–283 (cit. on p. 49).

[56] Davide Rossi et al. «PULP: A parallel ultra low power platform for next generation IoT applications». In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE. 2015, pp. 1–39 (cit. on p. 50).

[57] Pritam Sarkar and Ali Etemad. «CardioGAN: Attentive Generative Adversarial Network with Dual Discriminators for Synthesis of ECG from PPG». In: *arXiv preprint arXiv:2010.00104* (2020) (cit. on p. 53).

[58] Tim Schäck, Michael Muma, and Abdelhak M Zoubir. «Computationally efficient heart rate estimation during physical exercise using photoplethysmographic signals». In: *2017 25th European Signal Processing Conference (EUSIPCO)*. IEEE. 2017, pp. 2478–2481 (cit. on pp. 53, 54).

[59] Seyed Salehizadeh, Duy Dao, Jeffrey Bolkhovsky, Chae Cho, Yitzhak Mendelson, and Ki H 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* 16.1 (2016), p. 10 (cit. on pp. 53, 54).

[60] 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 pp. 53, 54).