Pruning in Time (PIT): A Lightweight Network Architecture Optimizer for Temporal Convolutional Networks

Published Version:
Pruning in Time (PIT): A Lightweight Network Architecture Optimizer for Temporal Convolutional Networks / Risso M.; Burrello A.; Pagliari D.J.; Conti F.; Lamberti L.; Macli E.; Benini L.; Poncino M.. - ELETTRONICO. - 2021:(2021), pp. 1015-1020. (Intervento presentato al convegno 58th ACM/IEEE Design Automation Conference, DAC 2021 tenutosi a San Francisco, CA nel 2021) [10.1109/DAC18074.2021.9586187].

Availability:
This version is available at: https://hdl.handle.net/11585/847051 since: 2022-01-23

Published:

DOI: http://doi.org/10.1109/DAC18074.2021.9586187

Pruning In Time (PIT): A Lightweight Network Architecture Optimizer for Temporal Convolutional Networks

Matteo Risso[*], Alessio Burrello[†], Daniele Jahier Pagliari[*], Francesco Conti[†],
Lorenzo Lamberti[†], Enrico Macii[*], Luca Benini[†], Massimo Poncino[*]
[*]Politecnico di Torino, Turin, Italy. *Email: name.surname@polito.it*
[†]University of Bologna, Bologna, Italy. *Email: name.surname@unibo.it*

***Abstract*—Temporal Convolutional Networks (TCNs) are promising Deep Learning models for time-series processing tasks. One key feature of TCNs is time-dilated convolution, whose optimization requires extensive experimentation. We propose an automatic dilation optimizer, which tackles the problem as a weight pruning on the time-axis, and learns dilation factors together with weights, in a single training. Our method reduces the model size and inference latency on a real SoC hardware target by up to 7.4× and 3×, respectively with no accuracy drop compared to a network without dilation. It also yields a rich set of Pareto-optimal TCNs starting from a single model, outperforming hand-designed solutions in both size and accuracy.**

***Index Terms*—Neural Architecture Search, Temporal Convolutional Networks, Edge Computing, Deep Learning**

I. Introduction

Deep learning (DL) models achieve state-of-the-art performance in many time-series processing tasks, such as biosignals analysis [1], [2], predictive maintenance [3] and sound classification [4]. Recurrent Neural Networks (RNNs) have long been considered the de facto standard for these tasks [5], but recently, Temporal Convolutional Networks (TCNs) – a subclass of Convolutional Neural Networks (CNNs) dedicated to time series – have been shown to provide comparable accuracy, whilst offering several advantages from a computational standpoint: smaller memory footprint, more data reuse opportunities and higher arithmetic intensity [6].

The deployment of DL models for direct inference on Internet of Things (IoT) edge devices is an emerging paradigm for the aforementioned time-series analysis tasks, as it provides several benefits compared to a cloud-centric approach. Edge computing reduces the pressure on the network, hence improving scalability, and guarantees predictable response latency, especially in presence of unreliable/unstable connectivity. Moreover, it may also improve energy efficiency and privacy, by avoiding the highly-consuming wireless transmission of large amounts of raw (and possibly sensitive) data to the cloud [7]. However, running DL inference at the edge implies deploying models on cost-constrained devices such as microcontrollers (MCUs), with extremely limited memory and low operating frequencies.

In this context, TCNs are especially interesting due to their hardware-friendly properties. Nonetheless, their deployment at the edge still requires a careful tuning of all hyper-parameters, in order to meet the accuracy requirements of the target application at the minimum cost in terms of number of parameters or operations. For standard CNNs, popular in computer vision, this tuning is increasingly performed with automatic tools, in a process known as Neural Architecture Search (NAS). Recent years have seen the appearance of a plethora of different NAS approaches, based on as many different search strategies, some specifically targeting edge devices [8]–[12].

While TCNs have most of their hyper-parameters in common with standard CNNs (e.g. the filter sizes, the channels in each layer, etc.) there is one fundamental peculiarity of these networks that requires an orthogonal approach, i.e., *time dilation*. As explained in Sec. II-A, dilation allows enlarging the receptive field of a TCN on the time axis without increasing the number of parameters. Tuning dilation parameters, therefore, offers a clear pathway to make TCN topologies smaller and more hardware-friendly.

In this work, we propose Pruning In Time (PIT), a novel and light-weight automated method to simultaneously optimize the dilation of all layers in a TCN, producing Pareto-optimal solutions in terms of accuracy and complexity within a *single* training run. This is achieved modeling the problem as a *structured weight pruning along the time axis*: starting from maximally-sized filters with no dilation, we concurrently train the model weights and increase the dilation by pruning regularly-spaced weights slices on the time dimension. To the best of our knowledge, PIT is the first architecture optimizer that seamlessly targets dilation as the main optimization knob; state-of-the-art tools (e.g. [12]) require explicitly listing all possible parameter combinations for every layer. With experiments on two different time-series processing tasks, we show that our method can find dilation factors that reduce the model size and inference latency on the GAP8 System-on-Chip (SOC) [13] by up to 7.4× and 3× respectively, with no accuracy drop compared to a network without dilation. Moreover, it can produce a rich set of Pareto-optimal TCNs starting from a single seed, including solutions that outperform hand-designed models, reducing the number of parameters by up to 54% without accuracy drop.

II. Background and Related Works

A. Temporal Convolutional Networks

TCNs share with 1-dimensional (1D) CNNs the main building blocks, i.e. convolutional, pooling and linear layers. However, to improve the processing of time series, TCN convolutions include two new properties. *Causality* forces the padding to be applied only on left side of the sequence; therefore, the outputs $\mathbf{y}_t$ are only functions of inputs $\mathbf{x}_{\tilde{t}}$ with $\tilde{t} \leq t$, so that output samples are not obtained taking information from the future. *Dilation* increases the receptive field of 1D convolutions without increasing the filters sizes, by applying a fixed step d between the input samples processed by each filter. Therefore, a 1D-convolution in a TCN is expressed by the following function:

$$\mathbf{y}_t^m = \text{Conv}\left(\mathbf{x}\right) = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} \mathbf{x}_{t-d\,i}^{l} \cdot \mathbf{W}_i^{l,m} \tag{1}$$

where t is the time index, $\mathbf{W}$ the filter weights, C_{in} the number of input channels, m the output channel, and K the filter size.

B. Neural Architecture Search

Neural Architecture Search (NAS) is increasingly used to automatically design deep neural networks, avoiding the hand tuning of hyper-parameters, e.g. layer channels, filter dimensions, etc. NAS automatically explores a large design space of possible hyper-parameters settings, co-optimizing the accuracy of the resulting network with its cost, measured as the memory footprint [10], the number of Floating Point Operations (FLOPs) or even the latency [12]. NAS outputs are sets of Pareto-optimal architectures in the complexity-accuracy space, from which users can select the best solution given their problem constraints. Due to this capability, the exploration of NAS solutions tuned for extreme edge devices has recently started getting traction [12], [14], [15].

We can categorize NAS methods in three main groups. Early approaches used reinforcement learning or evolutionary algorithms to improve the structure of the network at the cost of thousands of training runs (and GPU hours), thus being often prohibitive for real-world tasks [8]. More recent research has focused on *differentiable* NAS solutions (DNAS) [16], that model the problem as the optimization of a so-called *supernet*, which includes different alternative implementations of each layer. These methods jointly train a set of binary variables that determines a selection of layer implementations from the supernet (called a *path*) and the weights of that path. Although more efficient than the previous group, memory occupation and training time remain critical for these algorithms, since the supernet includes all possible combinations of layers implementations. ProxylessNAS [12] tackles the memory problem by training a single path of the supernet per batch, thus limiting the memory occupation of the whole model, but not exploring the full solution space at the same time.

Lastly, an emerging set of DNAS methods is based on so-called DMaskingNAS [9], [10], which aims at reducing the training time by limiting the search space to a single *seed* network. DMaskingNAS approaches add trainable parameters ("masks") to the seed network to tweak architectural features such as the number of channels or the filter dimensions. Specifically, a vector of $\gamma_i^{(l)}$ is defined per each layer l, and combined with the normal weights of the model, in such a way that setting each of the $\gamma_i^{(l)}$ to zero has an effect on the architecture (e.g. eliminating a channel or reducing the filter size). Then, the non-zero $\gamma_i^{(l}$ are minimized during training with an approach similar to *weight pruning*, thus reducing the network size. For instance, MorphNet [10] uses the already present multiplicative terms in batch normalization (BN) layers as $\gamma_i^{(l)}$, to zero-out entire channels. While the search space of DMaskingNAS algorithms is slightly more restricted than DNAS, it is still large enough to produce high-quality solutions; and at the same time, it can be explored at similar cost to a *single* network training.

In our work, we propose the first DMaskingNAS solution tuned specifically at exploiting the main architectural feature of TCNs – namely, *dilation* – to find Pareto-optimal alternatives in terms of accuracy/model size, to ease deployment on memory-starved IoT devices.

III. Pruning In Time

Despite the large number of NAS methods recently proposed in literature, most of them have focused on optimizing hyper-parameters that are relevant for standard CNNs, such as the number of channels in each layer and the kernel sizes [9], [10], [12], with particular focus on 2D CNNs for computer vision. To the best of our knowledge, no existing NAS has explicitly targeted the automatic optimization of dilation factors in 1D CNNs (d in Eq. 1). Nonetheless, dilation is fundamental for the accuracy of these networks, as detailed in [6], since it controls the relation between their receptive field in time (i.e., the range of samples covered by each convolution step) and their filter sizes (which determine the model complexity). Setting dilation factors by hand requires an empirical and time consuming trial and error process.

In this paper, we make up for this lack by proposing *Pruning In Time* (PIT), a light-weight and efficient DMaskingNAS method that learns the optimal set of dilation factors for a TCN given as seed network. The functionality of PIT is shown in Fig. 1: our algorithm starts from a seed network with maximally-sized filters and dilation $d = 1$ in all layers, and concurrently learns the TCN weights and the optimal dilations for each layer, based on a cost metric. Specifically, in this work we consider model size as the target metric, but the method is easily extendable to other types of optimizations (e.g., FLOPs reduction). The optimization of dilation factors is obtained by modeling the problem as a *weight pruning in time*, as explained in the following, hence the name of our approach.

A. Making Dilation Differentiable

In order to build a DNAS algorithm for dilated convolutions, the primary issue to be solved is how to embed the different dilation factors in a (differentiable) component of the

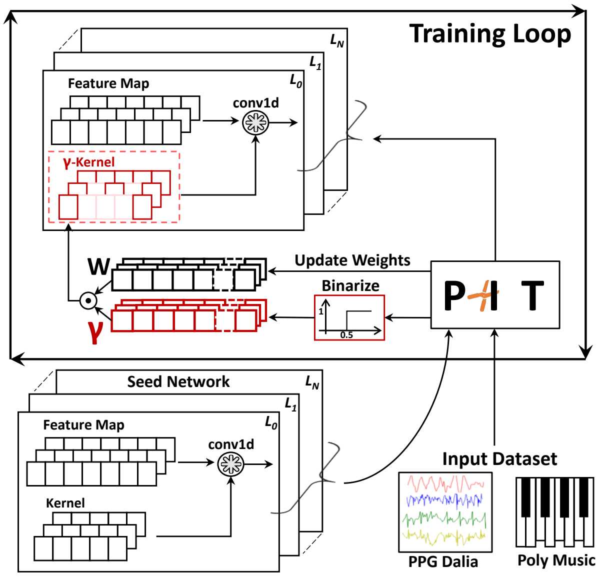

Fig. 1. Training flow of the proposed Pruning In Time architecture optimizer.

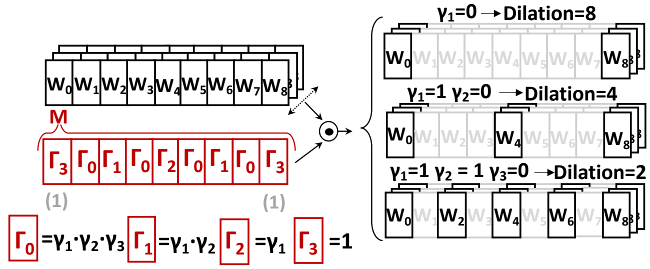
W0 W1 W2 W3 W4 W5 W6 W7 W8
M
Γ3 Γ0 Γ1 Γ0 Γ2 Γ0 Γ1 Γ0 Γ3
(1)
(1)
Γ0 =γ1·γ2·γ3 Γ1 =γ1·γ2 Γ2 =γ1 Γ3 =1
γ1=0 → Dilation=8
γ1=1 γ2=0 → Dilation=4
γ1=1 γ2 = 1 γ3=0 → Dilation=2

Fig. 2. Combination of $\boldsymbol{\gamma}$ elements with each other and with convolution filter weights to form different dilation patterns. Example for $rf_{\max} = 9$.

loss function, so that they can be optimized during training. Similarly to other DMaskingNAS techniques, our approach is built upon additional trainable parameters, which we call $\boldsymbol{\gamma}$.

Each temporal convolution in a TCN is characterized by a certain kernel size k and a certain dilation factor d. Together, these two parameters define the receptive field rf of the layer: $rf = (k-1)\cdot d$. Notice that, in this formulation, the dilation is *regular*, i.e., the number of skipped time-steps is constant in each convolutional layer. PIT limits the search space to such solutions, which are the only ones supported by current-generation inference libraries for MCUs [17], [18], and generally enable better low-level optimizations and more regular memory access patterns. In particular, we focus on dilation factors d that are expressed as powers of 2. These maintain a high degree of freedom in enlarging or reducing the receptive field, while restricting the solution space and enabling the simple formalization that we treat in the following.

For every temporal convolution, PIT starts by defining a vector of *binary parameters* $\boldsymbol{\gamma}$, containing $L = \lfloor \log_2(rf_{\max}-1)\rfloor + 1$ elements, where $rf_{\max}$ is the maximum supported rf. In particular, γ_0 is constant and always equal to 1, and is added just to simplify the mathematical notation. The remaining $\gamma_{1:L-1}$, instead, are the main knobs that control dilation. We binarize $\boldsymbol{\gamma}$ following a BinaryConnect-like approach [19]. In forward-propagation, we use a Heaviside step function and a threshold δ, fixed to 0.5 in our experiments:

$$\mathcal{H}(\hat{\gamma_i} - \delta) = \begin{cases} 1, & for\ \hat{\gamma_i} \geq \delta \\ 0, & for\ \hat{\gamma_i} < \delta \end{cases} \qquad (2)$$

where $\hat{\gamma_i}$ is the floating point version of γ_i. In backward passes, because the derivative of this function is zero almost everywhere, its gradient is treated with a straight-through estimator [19], i.e., the step is replaced with an identity function for the purpose of propagating gradients.

To restrict the search space to regular dilation patterns, the trainable parameters in $\boldsymbol{\gamma}$ are combined together in the way depicted in the red part of Fig. 2. We start by multiplying together the elements in $\boldsymbol{\gamma}$ to form a new set of $\boldsymbol{\Gamma}$, which will be the actual values used to perform the masking:

$$\boldsymbol{\Gamma_i} = \prod_{k=0}^{L-1-i} \boldsymbol{\gamma_k} \qquad (3)$$

Notice that, in Fig. 2, γ_0 has been directly replaced with 1. Intuitively, the parameters $\boldsymbol{\Gamma}$ have to act as on/off selectors that enable or disable entire time slices of the convolution filter, encoding regular patterns of power-of-two dilation.

As shown in the figure, $d = 1$ is achieved only when all trainable parameters are 1, i.e., when $\Gamma_0 = \gamma_0 \cdot \gamma_1 \cdots \gamma_{L-1} = 1$. To encode patterns with larger dilation, every time we double d we remove the condition on one γ_i, so for $d = 2$ we use $\Gamma_1 = \gamma_0 \cdot \gamma_1 \cdots \gamma_{L-2} = 1$, etc., up to the highest supported dilation ($d = 2^{L-1}$), which is obtained with the (always true) condition $\Gamma_{L-1} = \gamma_0 = 1$.

To obtain this behavior, we further combine the elements of $\boldsymbol{\Gamma}$ in a *mask vector* $\boldsymbol{M}$, of length $rf_{\max}$, as shown in Fig. 2. Each element of $\boldsymbol{M}$ is then multiplied with *all filter weights* relative to the corresponding time-step (and all channels) to perform the actual masking. The right side of Fig. 2 shows how the various dilation alternatives are mapped to the trainable $\boldsymbol{\gamma}$ through the masking procedure described above, for the case of $rf_{\max} = 9$ ($L = 4$).

In order to train $\boldsymbol{\gamma}$ with a DNAS method, the constructive description of the mask vector $\mathbf{M}$ given above must be expressed in a differentiable form. To do so, we use the following tensor transformation:

$$\boldsymbol{M} = \prod_{columns} \{[(\boldsymbol{\gamma} \cdot \mathbb{1}_{1\times L}) \odot \boldsymbol{T} + (\mathbb{1}_{L\times L} - \boldsymbol{T})] \cdot \boldsymbol{K}\} \qquad (4)$$

where $\prod_{columns}$ indicates the product of all elements in each column of the final matrix, $\mathbb{1}_{i\times j}$ is a matrix of 1s of size $i \times j$ and $\odot$ is the Hadamard product. $\boldsymbol{T}$ and $\boldsymbol{K}$ are two constant matrices of 0s and 1s. An example of these two matrices and of the entire transformation is shown in Fig. 3, for the same $rf_{\max}$ and L used in Fig. 2. Again, γ_0 has been directly replaced with 1 for clarity. Notice that $\boldsymbol{T}$ is just 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 (procedure not shown for sake of space).

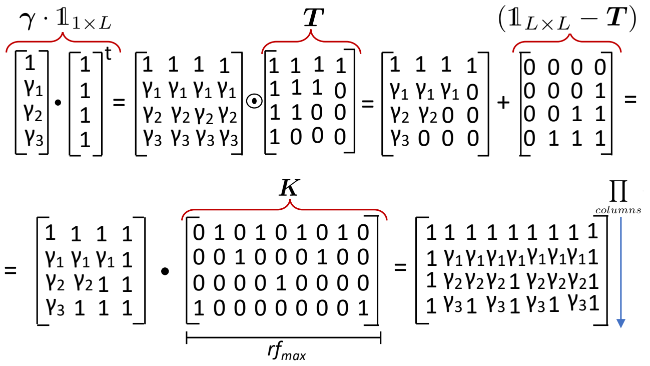
$\gamma \cdot \mathbb{1}_{1\times L}$
$\boldsymbol{T}$
$(\mathbb{1}_{L\times L} - \boldsymbol{T})$
$\boldsymbol{K}$
$\prod_{columns}$
rf_{max}

Fig. 3. Example of generation of the $\boldsymbol{M}$ mask vector with differentiable tensor operations, for the same layer size as Fig. 2.

Finally, the complete layer equation for dilated convolution in PIT becomes:

$$\mathbf{y}_t^m = \sum_{i=0}^{rf_{\max}-1} \sum_{l=0}^{C_{in}-1} \mathbf{x}_{t-i}^l \cdot (\boldsymbol{M_i} \odot \mathbf{W}_i^{l,m}) \tag{5}$$

where all symbols have been defined in previous Eq. 1-4.

B. Dilation Regularizer

Having modified all convolutional layers with γ vectors, PIT performs a standard training, where the loss function is augmented with a Lasso regularization term, to promote the sparsification (pruning) of the γ_i, thus enabling the exploration of architectures with $d > 1$.

The regularization term can be customized to direct the search towards a target cost metric. In this work, we consider the network size as our proxy for cost. Therefore, we use the following regularizer:

$$\mathcal{L}_R^{size}(\gamma) = \lambda \sum_{l=1}^{layers} C_{in}^{(l)} \cdot C_{out}^{(l)} \sum_{i=1}^{L-1} \text{round}\left(\frac{rf_{\max}-1}{2^{L-i}}\right) |\hat{\gamma}_i^{(l)}| \tag{6}$$

where λ controls the strength of the regularization and superscript (l) refers to the l-th layer. $C_{in}^{(l)}$ and $C_{out}^{(l)}$ are the number of input/output channels in the l-th layer, both of which influence the layer size in a way which is linearly proportional to the amount of non-pruned time-slices in the filter matrix. Finally, round $\left(\frac{rf_{\max}-1}{2^{L-i}}\right)$ is the number of alive (i.e. non-masked) filter time-slices added by each non-zero $\gamma_i^{(l)}$ (see Fig. 2).

Globally, the loss function optimized in the pruning phase by PIT is the following:

$$\mathcal{L}_{PIT}(\boldsymbol{W}, \gamma) = \mathcal{L}^{perf}(\boldsymbol{W}) + \mathcal{L}_R^{size}(\gamma) \tag{7}$$

where $\mathcal{L}^{perf}(\boldsymbol{W})$ is the loss term related to the TCN's performance, e.g., the classification accuracy.

C. Training Procedure

Algorithm 1 summarizes the training procedure implemented in PIT. As shown, the process is composed of three separate phases. Initially, we perform a warmup lasting Steps_{wu} training steps. In this phase, all elements in γ vectors are initialized to 1. Therefore, we only train the weights of the seed network, with maximally sized filters and $d = 1$ in all convolution layers, and considering only the network performance as objective. Next, the main pruning loop is initiated. Here, we concurrently update the weights and the γ vectors with the regularized loss. Once the pruning phase reached convergence (i.e., loss not improving on the validation set), we enter the third phase, where all γ are frozen to their latest binarized values, and the resulting network with dilation is fine-tuned, again considering only performance in the loss. We found that both warmup and fine-tuning significantly improve the final accuracy of the pruned networks.

Algorithm 1 PIT - Pruning in Time

```
for $i \leftarrow 1, \dots, \text{Steps}_{\text{wu}}$ do #warmup loop
    Update $\boldsymbol{W}$ based on $\nabla_{\boldsymbol{W}} \mathcal{L}_{perf}(\boldsymbol{W})$
end for
while not converged do #pruning loop
    Update $\boldsymbol{W}$ and $\gamma$ based on $\nabla_{\boldsymbol{W},\gamma} \mathcal{L}_{PIT}(\boldsymbol{W}, \gamma)$
end while
for $i \leftarrow 1, \dots, \text{Steps}_{\text{ft}}$ do #fine-tuning loop
    Update $\boldsymbol{W}$ based on $\nabla_{\boldsymbol{W}} \mathcal{L}_{perf}(\boldsymbol{W})$
end for
```

The procedure of Algorithm 1 can be repeated with different regularizer strenghts (λ in Eq. 6) to explore the performance versus cost design space. Similarly, also the number of warmup steps has an impact on the same trade-off. Indeed, as explained in [12], a shorter warmup, yielding a less accurate network at the beginning of pruning, tends to favor model simplifications, as their impact on accuracy is less at the beginning of the pruning phase.

Importantly, PIT can be easily integrated with other DMaskingNAS techniques that affect different hyper-parameters, e.g., [10] to tune the number of channels in each layer, simply by adding further regularization terms and masking parameters, to perform a wider exploration.

IV. Experimental Results

A. Setup

Datasets & architectures. We benchmark PIT using two different datasets and 1D-CNN seed networks. The first is Notthingham, a polyphonic music dataset extracted from 1200 American and British folk tunes [6]. Each input is a sequence of samples, each represented by a 88-bit binary code, corresponding to the 88 keys of a piano. As a starting point to build the seed model for this dataset, we use the TCN presented in [6], here refered as ResTCN. In detail, to automatically search for the best dilation parameters, we start from ResTCN with identical receptive fields as the one of [6], but setting $d = 1$ in each layer. The second dataset is PPGDalia [20], the largest publicly available dataset for PPG-based heart rate estimation. It includes measurements from a PPG-sensor and a 3D-accelerometer, together with golden HR values for 15 subject and a total of 37.5 hours of recording. In this case we use TEMPONet, first introduced in [1], as

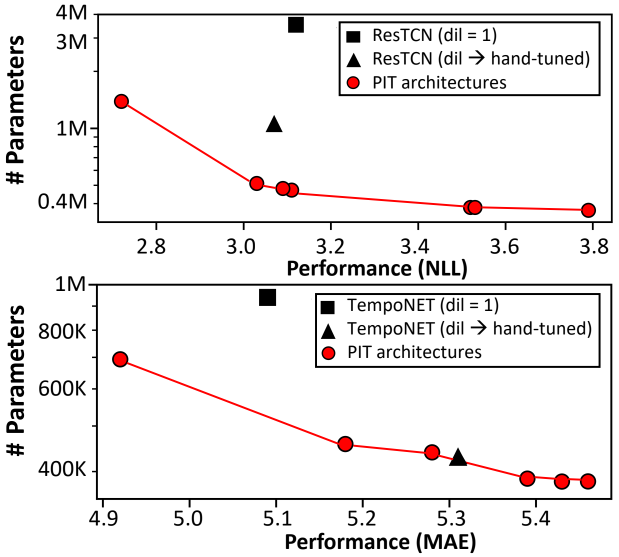

Fig. 4. PIT Pareto frontiers obtained starting from two seed architectures on the Nottingham dataset (top) and on PPGDalia (bottom). Each plot also reports the seed model (square) and the original hand-engineered TCN (triangle).

our seed architecture, since this network achieved excellent results on similar bio-signal processing tasks. Again, the seed is obtained setting $d = 1$ in all layers while maintaining the receptive fields. The PIT code and the results for all the other benchmarks introduced in [6] are released open source at `https://github.com/matteorisso/PIT` and are not reported here for sake of space.

Deployment. To measure latency and energy gains obtained by PIT, we deploy models produced by our tool on the GreenWaves Technologies' GAP8 System-on-Chip (SoC) [13], a parallel ultra-low-power platform that features one I/O core and an 8-core cluster with a RISC-V Instruction Set Architecture extension for enhanced digital signal processing. The SoC includes a two level memory hierarchy: a single-cycle clock latency 64 kB L1 scratchpad, and a 512 kB L2 memory. Additionally, an off-chip L3 memory can be plugged to expand the storage capability. GAP8 also includes two general-purpose Direct Memory Access (DMA) controllers to move data between memories, reducing the memory access bottleneck.

We deploy int8-quantized models using GreenWaves' proprietary neural network deployment flow, called NN-Tool, which supports dilated 1D convolutions. We set the I/O core frequency and the 8-core cluster frequency to 100 MHz.

B. Design Space Exploration

Starting from the two seed architectures detailed above, we perform a complete design space exploration, by tweaking the λ regularization-strength of PIT and the warmup duration (Steps$_{wu}$). Results are summarized in Fig. 4.

TABLE I
DILATIONS OBTAINED FOR THE DIFFERENT CONVOLUTIONAL LAYERS OF RESTCN AND TEMPONET IN DIFFERENT PIT OUTPUTS.

network	PIT dilations
ResTCN dil=hand-tuned	(1, 1, 2, 2, 4, 4, 8, 8)
PIT ResTCN small	(4, 4, 8, 8, 16, 16 32, 32)
PIT ResTCN medium	(4, 1, 4, 8, 16, 16, 32, 32)
PIT ResTCN large	(1, 4, 8, 8, 16, 16, 8, 1)
TEMPONet dil=hand-tuned	(2, 2, 1, 4, 4, 8, 8)
PIT TEMPONet small	(2, 4, 4, 8, 8, 16, 16)
PIT TEMPONet medium	(1, 2, 4, 2, 1, 8, 16)
PIT TEMPONet large	(1, 1, 1, 1, 1, 1, 16)

TABLE II
COMPARISON BETWEEN PIT AND PROXYLESSNAS, WITH TEMPONET AS SEED ARCHITECTURE AND PPGDALIA AS DATASET.

	ProxylessNAS		Pruning in Time	
	# weights	Perf. [MAE]	# weights	Perf. [MAE]
small	381k	5.43	381k	5.43
medium	517k	5.21	440k	5.28
large	731k	5.15	694k	4.92

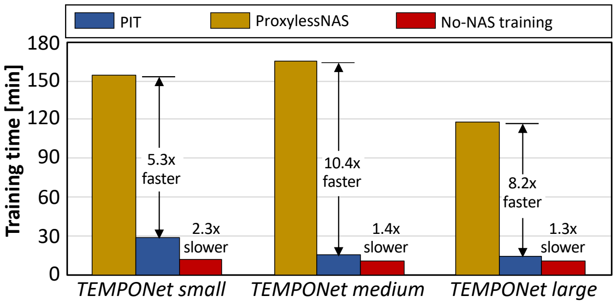

Fig. 5. Comparison of training time for ProxylessNAS, PIT, and a single NN training using a seed TEMPONet network and the PPGDalia dataset.

Considering all possible power-of-two dilations achievable given the seed network's receptive fields, PIT operates in a search space of $\sim 10^5$ different solutions for the ResTCN, spanning from NNs with 400k to 3M parameters. For TEMPONet, the search includes $\sim 10^4$ alternatives, ranging from 400k to 900k parameters. The dilations obtained by PIT for three different architectures, namely the largest (large), the smallest (small) and the closest in size to the original hand-designed ResTCN/TEMPONet (medium) are reported in Table I. For both benchmarks, PIT finds new Pareto-optimal architectures that improve both the accuracy and the network size simultaneously, compared to the seed baselines without dilation (black squares in the figure). In detail, we find a ResTCN-variant with 2.54× less parameters and an accuracy improvement of 13% (from 3.12 to 2.72 Negative Log-Likelihood - NLL loss) with respect to the seed; concerning TEMPONet, the top-performing architecture shows a compression of 1.35× with a Mean Absolute Error (MAE) reduction of 0.16. PIT also finds a new ResTCN-based architecture that dominates the *hand-tuned* original network presented in [6] in the Pareto sense, resulting in a 54% reduction of the parameters with almost identical performance. While the same is not true for TEMPONet, the hand-engineered network sits on the Pareto frontier in this case, showing the good quality of the architectures identified by PIT.

C. Comparison with state-of-the-art

Table II compares solutions found by PIT with the ones of the state-of-the-art ProxylessNAS [12], a DNAS algorithm based on the supernet idea, that can be adapted to search over different dilation factors in a 1D-CNN by manually including

TABLE III
PIT SOLUTIONS COMPARED TO ORIGINAL SEED NETWORKS WITHOUT DILATION AND WITH HAND-TUNED DILATION DEPLOYED ON GAP8 SOC.

	# weights	loss	latency	energy
ResTCN dil=1	3.53M	3.12 NLL	1002 ms	262.7 mJ
ResTCN dil=h.-t.	1.05M	3.07 NLL	500 ms	131 mJ
PIT ResTCN s.	**0.37M**	3.79 NLL	336.7 ms	88.2 mJ
PIT ResTCN m.	0.48M	3.09 NLL	**335.9 ms**	**87.9 mJ**
PIT ResTCN l.	1.39M	**2.72 NLL**	539.2 ms	141.3 mJ
TEMPONet dil=1	939K	5.08 MAE	112.6 ms	29.5 mJ
TEMPONet dil=h.-t.	423K	5.31 MAE	58.8 ms	15.4 mJ
PIT TEMPONet s.	**381K**	5.43 MAE	**54.8 ms**	**14.4 mJ**
PIT TEMPONet m.	440K	5.28 MAE	59.8 ms	15.7 mJ
PIT TEMPONet l.	694K	**4.92 MAE**	86.3 ms	22.6 mJ

all layer variants in the supernet. Specifically, for each layer, ProxylessNAS chooses between multiple alternatives, that we manually specified by increasing d and keeping C_{in} and C_{out} constant, so to match exactly the search space explored by PIT. For sake of space, the comparison is reported only on TEMPONet as seed architecture with PPGDalia as dataset. In Table II, we report three different architectures found by each algorithm (small, medium and large, selected with the same rationale of Table I). Both algorithms converge to same *small* network, while solutions are similar for the other two cases. Noteworthy, in the *large* case, PIT finds an architecture that is both smaller (694k vs. 731k parameters) and more accurate (4.92 vs 5.15 MAE) than ProxylessNAS.

In Fig. 5, we compare the training time required by PIT and ProxylessNAS to find the three networks. All the experiments have been performed with the same hardware set-up, namely a single *NVIDIA-GTX 1080Ti* GPU, and the same training algorithm parameters, including batch size = 128 and an early-stop patience of 50 epochs. PIT reduces the search time compared to ProxylessNAS by up to 10.4×, being only 1.3×-2.3× slower than the training of a single hand-designed TEMPONet architecture. This is mainly due to the concurrent training of *all weights* and masking parameters performed by PIT. In contrast, ProxylessNAS trains only one path of the supernet (and the corresponding weights) in each iteration of the training loop, causing a significant time overhead.

D. Deployment on GAP8

We deployed the original hand-engineered TCNs, the seed networks with unitary dilation, and the three different PIT outputs for each dataset reported in Table I, on the 8-cores cluster of GAP8. Table III reports the results in terms of network size, loss (either NLL or MAE for the two datasets), latency and energy.

Deploying the "medium" PIT ResTCN, we obtain a loss equivalent to that of the hand-tuned network presented in [6] with 2.2× less parameters, and 1.5× lower latency and energy. For TEMPONet, we obtain a medium network which is equivalent to the hand-engineered one in every metric.

Noteworthy, our small (medium) ResTCN networks have a 20% higher (0.9% lower) loss compared to the *seed* network, but are 9.5× (7.4×) smaller and 3.0× (3.0×) faster. On TEMPONet, the small (medium) models obtain a 6.9% (3.9%) worse error compared to the seed, with 2.5× (2.1×) less weights and 2.1× (1.9×) lower latency and energy.

V. CONCLUSIONS

NAS methods are becoming essential tools for deep learning, enabling a fast exploration of the model architecture design space without the tedious trial-and-error approach that characterized the development of new neural networks in the past. Therefore, it is fundamental to develop efficient NAS methods, which do not require extreme hardware resources and/or an enormous training times. At the same time, these tools must be capable of exploring all important hyper-parameters, which in case of a TCN include the dilation. The proposed PIT is a light-weight NAS framework that goes in that direction, and successfully generates size-optimized and hardware-friendly TCNs. PIT is able to generate improved versions of existing state-of-the-art architectures, with a compression of up to 54% with negligible accuracy drop, enabling their efficient deployment on resource-constrained MCUs.

REFERENCES

[1] M. Zanghieri et al, "Robust real-time embedded emg recognition framework using temporal convolutional networks on a multicore iot processor," *IEEE TBioCAS*, vol. 14, no. 2, pp. 244-256, 2020.

[2] N. D. Truong et al, "Convolutional neural networks for seizure prediction using intracranial and scalp electroencephalogram," *Neural Networks*, vol. 105, pp. 104 – 111, 2018.

[3] M. Azimi et al, "Data-driven structural health monitoring and damage detection through deep learning: State-of-the-art review," *Sensors*, vol. 20, no. 10, p. 2778, 2020.

[4] W. He et al, "Deep neural networks for multiple speaker detection and localization," in *2018 IEEE ICRA*, 2018, pp. 74–79.

[5] Z. C. Lipton et al, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint:1506.00019*, 2015.

[6] S. Bai et al, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint:1803.01271*, 2018.

[7] W. Shi et al, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct 2016.

[8] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[9] A. Wan et al, "Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions," in *IEEE CVPR*, 2020, pp. 965–974.

[10] A. Gordon et al, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," in *IEEE CVPR*, 2018, pp. 1586–1595.

[11] M. Tan et al, "Mnasnet: Platform-aware neural architecture search for mobile," in *IEEE CVPR*, 2019, pp. 2820–2828.

[12] H. Cai et al, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[13] E. Flamand et al, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE ASAP*, pp. 1–4.

[14] J. Lin et al, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems 33*, 2020.

[15] H. Cai et al, "Once-for-all: Train one network and specialize it for efficient deployment," in *ICLR*, 2020.

[16] H. Liu et al, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2019.

[17] ST Microelectronics. (2017) X-cube-ai. [Online] https://www.st.com /en/embedded-software/x-cube-ai.html. Accessed Nov. 2020.

[18] GreenWaves Technologies. (2019) Gap8 nntool. [Online]. https://greenwaves-technologies.com/manuals/. Accessed Nov. 2020.

[19] M. Courbariaux et al, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[20] A. Reiss et al, "Deep ppg: large-scale heart rate estimation with convolutional neural networks," *Sensors*, vol. 19, no. 14, p. 3079, 2019.