Politecnico di Torino 1859 | ScuDo Scuola di Dottorato ~ Doctoral School WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer Engineering (36th cycle)

Implementation of Machine Learning Algorithms on Ultra-Low-Power Hardware for In-Sensor Inference

By

Francesco Daghero
******

Supervisor(s):
Prof. Enrico Macii, Supervisor
Prof. Massimo Poncino, Co-Supervisor
Prof. Daniele Jahier Pagliari, Co-Supervisor

Politecnico di Torino
2024

Declaration

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Acknowledgements

First, I would like to thank my supervisor Enrico Macii and co-supervisor Massimo Poncino for allowing me to pursue a doctorate. I would like also to thank my co-supervisor Daniele Jahier Pagliari for the support during my PhD, providing invaluable suggestions and interesting research directions in these years. Special thanks also to the EML-EDA research group, in particular to Alessio Burrello, for his help and ideas for projects and publications.

Finally, I would like to thank my family and friends for their support during these years. To my loving parents, thanks for being my role model and for helping me grow up. To my amazing sister Michela, thank you for your invaluable support, I know I can always rely on you.

Abstract

Edge computing has become increasingly popular for Internet of Things (IoT) applications powered by Machine Learning (ML) inference. Moving the ML models directly to the data source can improve privacy, lower the latency, and yield higher energy efficiency, without requiring a constant internet connection as for a cloud-centric approach.

Nonetheless, ML models are known to be memory- and energy-hungry, requiring a significant amount of resources, not available on ultra-low-power edge devices, such as sensors, often based on Microcontrollers (MCUs).

As a consequence, an increasing research effort has been put into making ML more efficient, trading off limited accuracy for large savings in terms of energy or memory. This research branch has taken the name of Edge AI and it is the focus of this thesis.

In particular, this dissertation focuses on optimizing two popular models for edge AI: tree ensembles and deep neural networks (DNNs). Tree ensembles reach high accuracy with a limited memory and energy footprint, making them an ideal choice to deploy on resource-constrained hardware. Nonetheless, accurate ensembles often feature many trees, rapidly growing in memory and inference latency. In the first chapter of this work, I focus on how these models can be further optimized, reducing the memory footprint thanks to an efficient implementation and other approaches such as quantization. Moreover, thanks to a dynamic inference approach, I show a way to reduce the inference latency with little to no accuracy drops. All approaches detailed in this work concerning the optimization of tree ensembles have been collected and included in an open-source Python library.

The second chapter of this thesis focuses on deep learning (DL). DL models often reach state-of-the-art accuracy, coming however at the cost of a high number of parameters to be stored and computations to be performed. Therefore, I introduce a flow to obtain memory-inexpensive yet accurate DNNs that leverage sub-byte quantization and mixed precision. Then, I introduce three dynamic inference approaches to lower the average energy and latency per inference of DNNs. The first slices the network by its width, running only a subset of the channels and neurons depending on the input complexity. The second leverages the different complexity of the classes in a dataset, running an easy and inexpensive model to recognize the simplest classes while leveraging larger DNNs only for the other classes. The last one introduces an enhanced early-stopping mechanism tailored for datasets with class frequency imbalance, a common occurrence in edge ML, leading to higher accuracy and increased energy savings w.r.t. other approaches.

In conclusion, the contributions of this work are twofold. A novel deployment flow for tree ensembles is introduced, focusing on optimizations both at compile and run time. Then, multiple optimizations for efficient DNN deployments are proposed, both in terms of compile-time and run-time approaches, allowing the deployment of small yet accurate models even on the most constrained edge devices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, Machine Learning (ML) has become increasingly present in our society, becoming the core component of applications in a wide set of domains, such as autonomous driving [1], medical analysis [2, 3] and smart manufacturing [4]. Notably, Deep Learning (DL) algorithms for computer vision [5, 6], natural language processing [7], and speech recognition [8] tasks have achieved outstanding results, outperforming even humans in some tasks [5]. Commonly, these applications leverage a cloud-centric paradigm, deploying the ML model on remote infrastructures, where both training and inference are performed on powerful hardware.

The recent rise of Internet of Things (IoT) applications has led to a growing interest in moving ML models *to the edge*, performing the inference directly on the resource-constrained embedded devices collecting the data. Edge computing leads to several benefits, mostly stemming from the reduced reliance on an internet connection. For instance, applications handling sensitive data (e.g., geographical position) may cause privacy concerns when transmitting data to the cloud, as a secure connection is not always available. Real-time applications, such as autonomous driving ones, have tight latency constraints, nearly impossible to meet if large amounts of data have to be sent to remote servers. Moreover, transmitting data is an energy-hungry operation, severely impacting the lifetime of battery-operated devices expected to run for as long as possible (e.g., sensors). In the aforementioned cases then, performing the computations locally would be highly beneficial. Moreover, edge computing brings a reduced operating cost, easing the deployment infrastructure companies have to manage while also diminishing the carbon footprint [9].

Modern ML models however are memory and computationally intensive, requiring up to billions of Multiply-And-Accumulate (MACs) operations and thousands of bytes for storage. For instance, the popular and ”small" network EfficientNet-B0 [6] requires storing 5.3M parameters and performing 0.39B FLOating-Point OPerations (FLOPs) per inference for an RGB image of dimension 224x224. On the other hand, embedded devices, such as Microcontrollers (MCUs), generally present significant limitations in terms of on-chip memory ($\simeq 1MiB$) and computational power, due to the need to keep limited costs and long operating times when battery-powered.

Consequently, an increasing effort has been put into enabling ML inferences of accurate yet tiny models on edge devices. This work goes in this direction, introducing multiple optimizations at the algorithm level, trading off as little prediction quality as possible for large savings of resources. In particular, in this dissertation, I focus on the optimization of tree ensembles (e.g., Random Forests) and Deep Neural Networks (DNNs), as both are popular options when considering a deployment at the edge. Tree ensembles feature low memory and energy footprint when deployed, achieving similar accuracy to resource-intensive DL models for easy tasks, making them an ideal choice for deployment targets too constrained for DL-based solutions. For instance, the DL model proposed in [10] for an irregular heartbeat detection task requires 200k FLOPs and parameters to be stored. On the other hand, the Random Forest benchmarked in [11] on the same task requires only 1k operations and 2k parameters, while achieving similar accuracy. Nonetheless, the footprint of larger and more accurate ensembles grows rapidly, saturating the device memory or still requiring a significant number of operations. This makes them an ideal target for optimization, presenting an interesting and open challenge to enable efficient deployments.

DNNs instead, while reaching state-of-the-art accuracy on several tasks, are generally too large to be directly deployed at the edge, in particular on constrained Microcontrollers (MCUs). Their optimization in terms of memory and energy is an open challenge with new techniques being constantly introduced in the literature.

In this work, I propose a set of optimizations focusing both on reducing the memory footprint and the inference energy cost, allowing the deployment of increasingly accurate models and thus enabling even more applications to be performed on edge devices. The focus in particular is on dynamic inference approaches, changing at runtime the computations performed by the model to adapt to changes in external conditions (e.g., low battery).

This work is organized as follows. Chapter 2 presents the concepts that are crucial to understanding the contributions detailed in this thesis. Chapter 3 introduces the datasets and the deployment targets used for benchmarking the effectiveness of the optimizations introduced. Chapter 4 details the proposed optimizations for tree ensembles, focusing both on reducing the memory footprint of the models and the energy required at inference time. Chapter 5 focuses instead on the optimization for DNNs, first for memory reduction and then for latency optimization. Finally, Chapter 6 reports the conclusion.

Chapter 2

Background

In this chapter, I introduce the required background for this work. First, I provide an overview of common applications and tasks in the field of embedded Machine Learning (ML). Second, I introduce the fundamental concepts about tree ensembles and Deep Neural Networks (DNNs), as both are the targets for the optimization I propose in the following chapters. Finally, I provide some details on IoT edge nodes, describing the main characteristics of the typical deployment target for embedded ML.

2.1 Target Applications

The proliferation of IoT devices has led to an exponential growth of applications that can benefit from an *on-device* execution. For instance, wake-word recognition, object detection, facial recognition, and anomaly detection are tasks already commonly executed at the edge [9]. Notably, while a cloud-based approach for such tasks is often feasible, it would lead to sub-optimal performances in terms of inference latency, energy efficiency, and security. In the following sections, I introduce the most common applications for embedded ML.

2.1.1 Computer Vision

Computer Vision (CV) is one of the most explored fields of ML. In particular, DL solutions have shown outstanding results, being able to outperform even humans in some applications [5, 1]. Tasks such as person detection, and surveillance have become increasingly common in edge computing scenarios , and new and complex ones, such as autonomous driving, are being tackled. These applications usually rely on data collection from cameras, that requires immediate processing afterward, often with tight latency constraints (e.g., person detection in autonomous driving). A decentralized inference, relying on an internet connection, may then violate these constraints. Additionally, tasks that collect sensitive data such as face recognition (e.g., pictures of faces) benefit from an on-device inference both in terms of security, as data does not leave the device, and in terms of energy efficiency, avoiding an energy-hungry operation such as transmission. Edge CV has shown promising results also for video-processing applications, more computationally intensive than simple image classification tasks, yielding acceptable frame rates at a limited energy cost.

2.1.2 Speech Processing

As for computer vision, Deep Learning has shown outstanding results in the speech recognition field [5]. In this domain, the prediction latency, while not as critical as for CV applications, has a huge impact on the user experience. Therefore, cloud-based solutions, often introducing noticeable latency, lead to sub-optimal products. On the contrary, edge computing becomes an ideal solution, yielding lower latency and thus enhancing the user experience. Finally, some applications of speech recognition, e.g. wake-word recognition for smart speakers, require continuous inferences to be performed in the background. A cloud-based approach in this case poses significant limitations, as large amounts of sensitive data (e.g., any audio trace recorded by the smart speaker) have to be transmitted to servers. Such an approach would be detrimental for privacy and energy efficiency, while also scaling poorly, as billions of devices would need to constantly connect to a limited number of remote servers. For the aforementioned reasons, smart speakers are one of the applications that leverage edge computing the most, performing the wake word task locally and offloading to remote models only complex voice commands. In fact, complex speech recognition models (e.g., translation) are generally too computationally intensive for on-device deployment, leaving this field an open challenge.

2.1.3 Time-series processing

Smart devices, such as smartphones and wearables, collect data continuously from multiple sensors they are equipped with. These time series can be processed for a wide range of tasks, enhancing the user experience through smart decisions directly taken from the device. Popular time series applications include data-driven power optimization [12], audio processing [13], health monitoring [14], and logging [15].

Smart sensors have also become increasingly present in smart factories and smart cities, continuously collecting data. Popular applications leveraging ML or DL include load prediction and balancing in smart grids [16], traffic monitoring [17], predictive maintenance of industrial equipment [4], and soil monitoring in agriculture [18]. In the aforementioned applications, avoiding or limiting data transmission is even more crucial than for smartphones or wearables for energy efficiency, as these sensors are often battery-operated and need to be operational for months or years.

2.2 Machine Learning at the edge

The main challenge faced by ML at the edge lies in the severe limitations of resources such as processing power, energy, and memory on IoT devices. Modern ML models in fact, and in particular DL architectures, are notoriously memory-hungry and computationally expensive, requiring the storage of hundreds of thousands of parameters and comparable FLOPs at inference time.

This led to the introduction of efficient and compact ML and DL algorithms, that paired with optimization techniques such as quantization, enable inferences even on ultra-low-power devices. Concerning classical ML, in this work I focus on Decision Trees (DTs) and their ensembles, as they are accurate and lightweight, often matching the performances of far larger DL architectures for easier tasks. For example, a DL model for Electrocardiogram anomaly detection can require around 200k FLOPs and a similar amount of parameters to be stored [19]. On the other hand, a tree ensemble can reach iso-accuracy with 2k parameters and 1k operations [11]. Then, I introduce memory and computationally efficient DNNs together with common optimization techniques that make DL a suitable candidate for edge inferences.

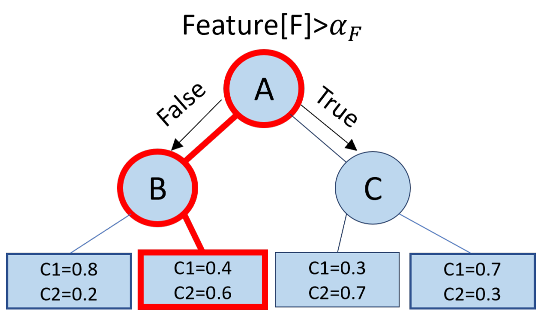


Fig. 2.1 Overview of a DT for a binary classification problem. Leaves are shown as rectangles, reporting the class probabilities of an input reaching them. Non-terminal nodes are represented as circles.

2.2.1 Decision Trees and Tree Ensembles

Decision Trees

Decision Trees are non-parametric and shallow Machine Learning algorithms that can be employed both for regression and classification in supervised tasks.

During the training phase (also known as the “growing” or “fitting” phase), DTs create piece-wise constant approximations of the target variable in the form of a set of decision rules. As extracting the optimal rule set is an NP-hard problem, common training algorithms are based on heuristics, greedily extracting the rules through a divide-and-conquer approach. Notably, several training algorithms have been introduced in the literature, each featuring different criteria in the extraction of the rules and the learning of the thresholds. As this work focuses on *inference* optimizations, the training phase details are out of the scope and an interested reader may refer to [20]. In the following, the details of the inference phase are provided.

Figure 2.1 shows a trained DT for a binary classification task. Nodes are depicted as circles and leaves as rectangles. Each node stores the index F of the input feature used for the split and the learned splitting threshold α. Depending on the condition in the node, either the left branch or the right branch is selected. Specifically, if the condition is true (false), the right (left) child is reached. Leaves instead store the DT predictions, i.e. the probability of an input reaching the leaf belonging to each class for classification or a continuous scalar in case of regression. The *level* of a node can be defined as the number of nodes that, starting from the root, have to be visited to reach it. The *depth* (D) of a tree is equivalent to the maximum level, that is, the longest path from the root node to a leaf node.

Algorithm 1 Decision Tree Inference

```
$n = \text{Root}(t)$
While $n \notin \text{Leaves}(t)$
    if Feature(n) $> \alpha(n)$:
        $n = \text{RightChild}(n)$
    else:
        $n = \text{LeftChild}(n)$
$P = \text{Prediction}(n)$
```

Algorithm 1 reports a high-level overview of the DT inference, denoting respectively as Root(T) and Leaves(T) the root node and the leaves of tree T. LeftChild(n) and RightChild(n) are the left and right children of node *n*, while Feature(n) and Alpha(n) denote the input feature and the respective threshold for the comparison. Lastly, Prediction(n) is a field defined only for leaves and contains either the class scores or the continuous value predicted.

The memory complexity of DTs grows with O(2^D), i.e. proportionally w.r.t the number of nodes, with the worst case being a *perfect* tree, a tree with 2^D nodes [20]. The time complexity of DTs' inference is O(D+M), where M is the number of classes in the leaves. In the worst case, D branching operations are needed, followed by an argmax operation on M elements to determine the most likely class. Noteworthy, M=1 for regression tasks.

DTs' main strength lies in their lightweight operations and limited memory footprint, making them an ideal candidate for embedded devices. Nonetheless, their main flaw stems from their tendency to overfit and to introduce a bias towards the majority class in datasets that are unbalanced [20].

Tree Ensembles

To overcome the limitations of DTs, several types of ensembling techniques have been introduced in literature [21–23]. These models aggregate multiple DTs, denoted as "weak learners", running them on the same input before merging their predictions. While this comes at the cost of an increased number of computations and a larger memory footprint, it enhances the model's resistance against overfitting, improving

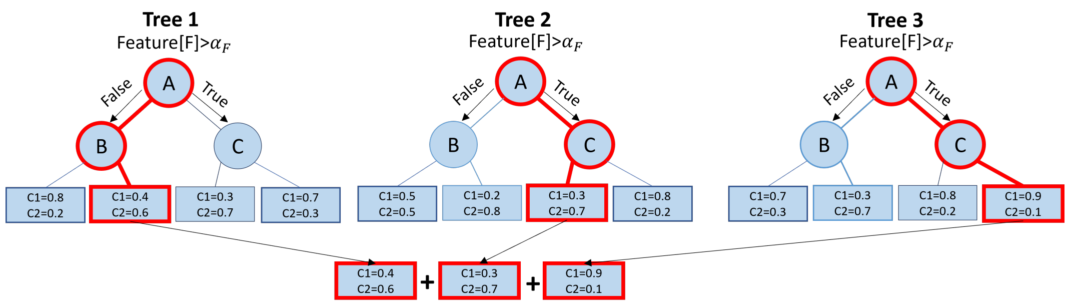


Fig. 2.2 Overview of an RF with depth and width of 3.

its generalization and prediction quality. In this work, I focus on two popular tree ensembles for classification tasks: *Random Forests* (RFs) and *Gradient Boosting Trees* (GBTs).

RFs [21] are ensembles of DTs each trained on random subsets of the data (an approach denoted as *boosting*) and on random subsets of input features. Thanks to these training techniques, the DTs composing a forest have low correlation among each other, leading to improved accuracy and enhancing the model's resilience against overfitting.

Figure 2.2 depicts an RF with D=3, M=2 and N=3. N denotes the width of the ensemble, that is, the number of estimators, equivalent for RFs to the number of trees.

Algorithm 2 Random Forest Inference

```
$P = \mathbf{0}_M$ // array of 0s of size $M$
for $t \in$ Forest:
    $P = P + \text{TreeInference}(t)$
$class = \text{argmax(P)}$
```

An overview of the inference of an RF is provided by Algorithm 2, where TreeInference(t) corresponds to Algorithm 1 for tree t. Each DT performs an inference pass on the same input, with a final aggregation step among the trees' predictions. This aggregation step can be performed either through a sum or an average, depending on the RF implementation. Note that older libraries stored only the most likely class in the trees, computing the final class prediction with a majority voting among the weak learners. On the other hand, modern implementations average (or sum) the tree scores, slightly increasing the complexity of the inference

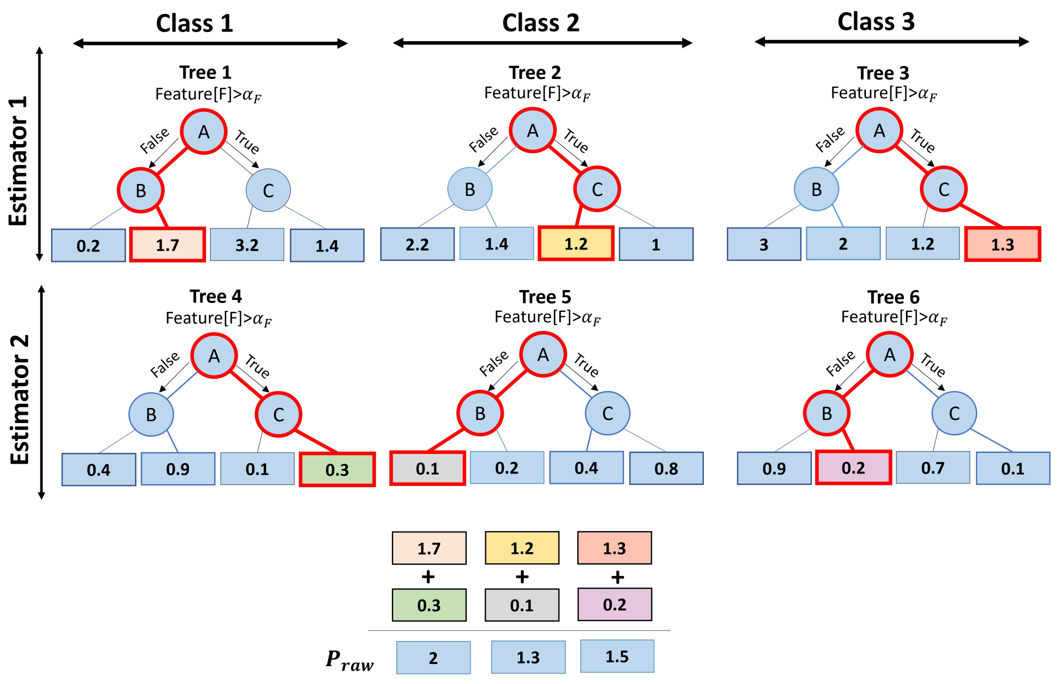


Fig. 2.3 Overview of a GBT for a 3-class classification task (M=3), with 2 estimators (N=2) and a depth of 3 (D=3).

but improving the accuracy of the model. In this work, I follow the trend of modern libraries such as [24], using weak learners that aggregate their output rather than a simple class vote. Concerning both the time and memory complexity of RFs, they mirror the ones of the DTs composing them, scaled by a factor N. Therefore, the time complexity of an RF grows with O(N*D), while the memory complexity with O($N * 2^D$). Note that D denotes the maximum depth among all DTs in the ensembles and is usually fixed for all trees during the training.

GBTs [22] are ensembles of DTs spawned with the *gradient boosting* technique, that is, an iterative optimization algorithm working in a step-wise fashion. At each step, a new DT is generated to correct the errors of the previous ones, minimizing the residual errors. Noteworthy, GBTs' estimators are composed of sets of M regression DTs, one for each class in the classification task. As a consequence, single weak learners in GBTs do not have multiple class scores but store instead a scalar value. An overview of a GBT with M=3, D=3 and N=2 is shown in Figure 2.3.

Algorithm 3 reports the pseudo-code of a GBT inference, where t_i denotes the DT belonging to the estimator e and handling class i. The raw outputs of the GBT are stored initially in P (P_{raw} in Figure 2.3) and converted at the end of the inference

Algorithm 3 Gradient Boosting Trees Inference

```
$P = \mathbf{0}_M$ // array of 0s of size $M$
for $e \in$ Estimators: // $e$ array of $M$ trees
    for $t_i \in$ e:
        $P_i = P_i + \text{TreeInference}(t_i)$
$class$ = argmax(compute_probabilities($P$))
```

with a formula that depends on the training loss. The memory complexity of GBTs is scaled both by the width of the ensemble and the number of classes of the task, becoming O($N * M * 2^D$). The same holds for the time complexity, growing with O(N*M*D).

2.2.2 Compact Deep Neural Networks

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become increasingly popular for structured data (e.g., images and time series) in recent years, achieving state-of-the-art accuracy on several tasks [5]. In particular, 1D CNNs have shown outstanding results for time-series analysis, while providing significant benefits w.r.t to other popular architectures such as LSTM [25] and transformers [7]. Specifically, CNNs are more resilient to vanishing and exploding gradients at training time, while also being more memory efficient. Moreover, at inference time, CNNs show lower memory footprints [26], a crucial property for efficient inferences at the edge.

The key components of CNNs are Convolutional (Conv) layers, applying on the input data multiple sliding window filters. Mathematically, a 1d convolution (Conv1D) can be computed as follows:

$$O_{k,t} = \sum_{c}^{C} \sum_{i}^{F} I_{c,t+i} * W_{k,c,i} \tag{2.1}$$

with I, W, O denoting input, weights, and output tensor at the output timestep t and input/output channel c/k. F represents instead the dimension of the filter weights. The equation assumes a stride (S) of 1 for simplicity. The computational complexity of a convolution is $K * C * F * T$, where T is the number of output timesteps and C/K is the number of input/output channels. The memory footprint is $K*C*F+K*T_{in}+C*T_{out}$, i.e. the elements of the convolutional filter and the input/output buffers. Noteworthy, one of the main strengths of convolutions lies in their weight reuse, as the same filter is applied to multiple inputs, making CNNs far more memory efficient than architectures such as the MultiLayer Perceptron (MLP).

Conv layers are generally followed by a Batch Normalization (BN) layer and then by an element-wise function, such as ReLU [27].The former applies an affine transformation to all elements in each channel, improving the training stability and favoring the applications of optimizations such as quantization. The batch normalization equation is the following:

$$O = \frac{I-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}}\gamma+\beta \tag{2.2}$$

where γ and β are parameters learned at training time and ε is a small constant value added for numerical stability. The values σ and μ are the standard deviation and the mean of the input mini-batch during training or of the whole training data during inference. Notably, at inference time, BN layers can be converted into an affine transform with equation $O = A*I+B$.

Other layers commonly found in modern CNNs are Pooling (Pool) and Fully-Connected (FC) layers.

Pooling layers are used to downsample the spatial dimension of the data, usually through a windowed max (MaxPool) or average (AvgPool) operation, reducing the time and memory complexity of the network, while retaining the important features.

Fully Connected layers are generally used as the last component of the networks, converting a flattened version of the input into class probabilities. Mathematically, a fully connected layer can be computed as follows:

$$O_{f_o} = \sum_{i}^{F_i} I_{f_i} * W_{f_o,f_i} \tag{2.3}$$

where f_o and f_i denote respectively the output and input features.

Quantization

Design-time (or static) software-level optimizations to enhance the energy efficiency or reduce the memory footprint of DL models have become increasingly popular in recent years [28, 29]. One of the most popular techniques is *quantization* [28], now implemented even in general-purpose frameworks for DL such as PyTorch [30] and Tensorflow [31].

Quantization is based on the fact that 32-bit floating-point numbers to represent the *weights* and *activations* tensors of a DNN are redundant at inference time. Several works have in fact shown that reducing the precision to 8-bit integers introduces little to no accuracy degradation [32–34]. On the contrary, *quantization* presents several benefits, reducing the model's memory occupation, the inference latency, and the energy cost. Leveraging a lower bit-width not only requires less storage but reduces also the memory bandwidth needed to transfer the network's tensors from and to the on-chip memory, often a dominant part of the total inference latency and energy [35]. Noteworthy, sub-byte quantization has been also explored [32, 33], however, it often introduces a non-negligible loss of accuracy, while also causing a latency overhead due to the packing/unpacking of the data before processing. Therefore, it is effective only for a reduced number of tasks, as opposed to 8-bit quantization which is rarely harmful for CNNs.

Several quantization algorithms have been introduced in the literature and can be categorized depending on the distance they assign to adjacent fixed-point values (*uniform* vs *non-uniform*), and whether the integer distribution is centered around zero (*symmetric* or *asymmetric*). The distance between fixed-point values is denoted as the quantization step, while the shift of the integer distribution is the zero point. Uniform quantization, while introducing minimal overheads, underperforms if the data is not distributed uniformly, a common occurrence for DNNs' weights and activations [28]. Non-uniform quantization however has to be carefully implemented to avoid large overheads, as most HW platforms exploit a uniform number representation for integers. A popular non-uniform quantization approach is based on a base-2 *logarithmic* scale, as smaller magnitude data is stored with finer granularity, a desirable property for DNNs. On the actual HW, a base-2 logarithmic quantization allows multiplications to be implemented as bitwise shifts [36], yielding efficient inferences.

Quantization can be applied either during training (*quantization-aware training*) or post-training, with the former significantly reducing the accuracy drops w.r.t the floating point version [28]. Intuitively, quantization-aware training forces the network to consider the loss of accuracy introduced with quantization, guiding the training algorithm to make up for it with different weight values.

Finally, different quantization granularities have been proposed. Network-wise quantization is the simplest approach, assigning a single bit-width to the whole network. This approach, while the easiest to implement, leads to smaller savings in terms of memory and latency. Layer-wise quantization (or mixed-precision) assigns to each layer a different precision, finding better trade-offs in terms of memory versus accuracy. The training however becomes significantly more complex, as finding the optimal precision to assign to each weight and activation tensor in a network is a non-trivial task.

A particular case of sub-byte quantization is *binarization*, representing the most aggressive case of quantization, as the precision is reduced to 1-bit [37]. The most common form of binarization consists of representing -1 and +1 respectively as 0 and 1, obtaining the conversion with a $sign()$ function. Binarized Neural Networks (BNNs), featuring both activations and weights at 1-bit, gain significant advantages both in terms of memory (up to 32x less w.r.t. 32-bit floating point) and computations (up to 32x). In fact, floating-point operations can be eliminated, with all Multiply-And-Accumulate (MAC) operations being replaced by bit-wise operations [37]. The dot product then can be implemented with the following equation:

$$O = 2 * \mathbf{P}_i^B w_i \otimes x_i - B \tag{2.4}$$

where **P**, $\otimes$, and B denote respectively the *popcount* operator, the bitwise XNOR, and the length of the vectors. Popcount is the bitwise equivalent of an accumulation, counting the number of 1s in a vector, while XNOR corresponds to a bit-wise multiplication. Intuitively, these operations can be performed in 32-bit registers even on general-purpose hardware, leading to the computation of up to 32 multiplications in parallel and thus achieving significant speedups [37]. However, binarization generally comes with large accuracy drops for complex tasks, making it feasible for a limited set of use cases.

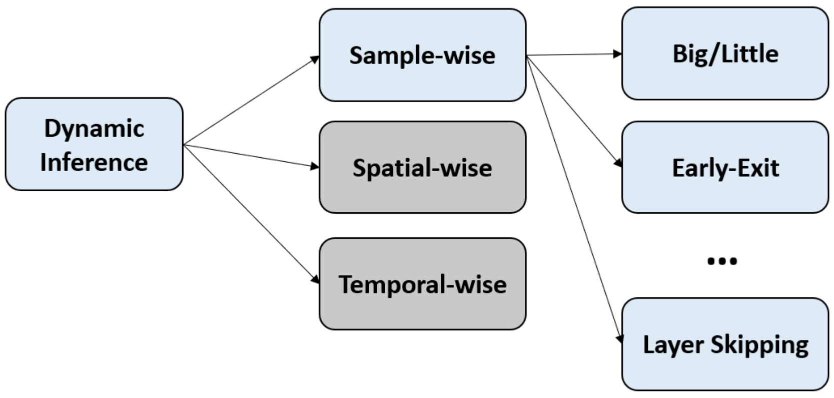


Fig. 2.4 Overview of dynamic inference methods. Approaches related to the ones introduced in this work are shown in blue.

2.2.3 Dynamic Inference

The main limitation of optimizations such as quantization [28] for DL and pruning for tree ensembles [38] lies in their static nature. Once the desired complexity versus accuracy trade-off has been achieved, the model is fixed and is unable to change further at runtime. In contrast, adapting depending on external conditions (e.g. battery lifetime) is a desirable feature given the many deployment scenarios that IoT nodes have to handle.

A naive implementation of a system that provides different accuracy vs complexity trade-offs consists of deploying multiple independent models on the target hardware, switching among them at runtime. However, this approach leads to huge memory overheads, as for each operating mode a distinct ML model has to be deployed. Dynamic (or adaptive) inference approaches consist of changing at runtime the computational graph depending on external conditions, providing different operating modes while keeping the memory overhead minimal [39–41]. Several variants of adaptive inference have been proposed in the literature [41, 40, 39, 42, 43], each with different trade-offs in terms of memory overhead, operating modes, and prediction quality. Dynamic inference approaches can be categorized according to their granularity, i.e. temporal-wise when operating on the temporal dimension of time-series data, spatial-wise when operating at pixel-level, and sample-wise on single input data. Figure 2.4 shows a high-level overview of dynamic inference approaches, introduced in the following sections. The figure reports in blue methods that are tied to the approaches introduced in this work, falling in the sample-wise category.

Temporal-wise

Temporal-wise approaches adapt the computations along the temporal dimension of sequential data, allocating less or no resources to unimportant elements in sequences, such as consecutive similar frames in videos. Adaptive approaches falling in this category include updating selectively [44, 45] or skipping entirely [46] the update of the hidden states in recurrent neural networks (RNNs) or even adaptively discarding input tokens [47].

Spatial-wise

Spatial-wise approaches are popular in CV tasks, where it has been found that not all the input regions in CNNs contribute equally to the final prediction [48]. Therefore, correct predictions can be achieved even when only a fraction of the input is processed. Popular dynamic techniques falling in the spatial-wise category include *dynamic sparse convolution* and *dynamic additional refinement*. Dynamic sparse convolutions perform the computations on a subset of the pixels, selected at runtime depending on i) the input sparsity [49], ii) a prediction on where the output pixels will be zero [50, 51], iii) an estimation of the pixel importance [52–54].

Dynamic additional refinement instead consists of performing an initial set of cheap computations on the whole input feature maps and additional refinement operations only on a subset of the input [55].

Noteworthy, pixel-level approaches can lead to sub-optimal performances when deployed on real hardware, as they force sparse and/or irregular computations. As a consequence, works proposing region-level spatial-wise granularity (i.e., working on regions of the input features) have been introduced. For instance, resolution-level dynamic networks deploy multiple sub-networks in a cascading [56] or parallel [57] fashion, each performing a forward pass on the same input image rescaled at a different resolution.

Sample-wise

Sample-wise approaches are among the most popular dynamic inference mechanisms, tuning the computations according to the input sample. They obtain an increased energy efficiency w.r.t. static architectures thanks to their reduced set of computations for easy inputs while being close to iso-accuracy for complex ones. The approaches presented in Chapter 4 and 5 fall into this category.

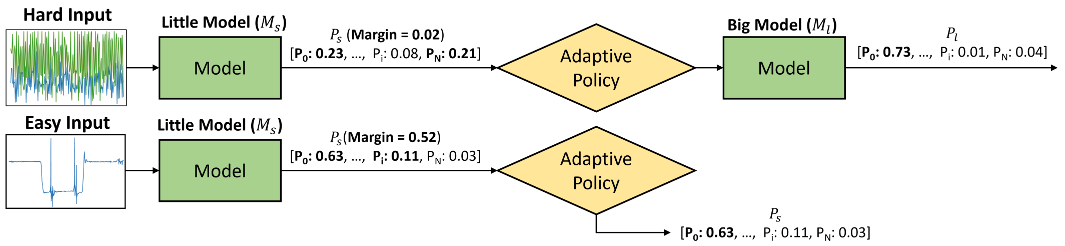

Hard Input
Little Model (M_s)
Model
P_s (Margin = 0.02)
[P_0: 0.23, ..., P_i: 0.08, P_N: 0.21]
Adaptive Policy
Big Model (M_l)
Model
P_l
[P_0: 0.73, ..., P_i: 0.01, P_N: 0.04]
Easy Input
Little Model (M_s)
Model
P_s(Margin = 0.52)
[P_0: 0.63, ..., P_i: 0.11, P_N: 0.03]
Adaptive Policy
P_s
[P_0: 0.63, ..., P_i: 0.11, P_N: 0.03]

Fig. 2.5 Big-Little Dynamic Inference for time-series data.

Figure 2.5 shows the first and most popular implementation of sample-wise dynamic inference, named *Big-Little*. Two models of increasing complexity and accuracy are deployed in cascade. The first, the *little* model M_s, performs an initial inference pass on the input and uses an early stopping (or adaptive) policy to determine whether it is safe (in terms of prediction quality) to stop the execution. If the early stop is triggered, the output of the first model is taken as the final one. In the opposite case, the *big* model M_l performs an inference on the same input, and its output is used as the result. The whole Big-Little approach is based on the assumption that easy inputs are more frequent than hard ones [39]. Therefore, most of the time, the less accurate and lightweight model M_s is enough to obtain a correct classification. On the contrary, the few complex inputs require the execution of both M_l and M_s, introducing a small overhead, but still obtaining a correct prediction. Intuitively, this scheme is convenient as long as the big model is rarely activated, reducing significantly the average energy per inference and yielding iso-accuracy w.r.t. to a large and accurate model.

The adaptive policy plays a fundamental role in this schema and leverages the prediction confidence of M_l to determine whether the inference can be stopped. An inaccurate confidence estimation algorithm would either underuse or overuse the big model, respectively reducing the accuracy or the energy efficiency of this approach. Moreover, if the computation of the adaptive policy is too expensive, the introduced overhead may outweigh any gains in terms of energy obtained through the dynamic approach.

One of the most effective confidence estimation metrics is the *Max Score* (S), leveraging as an indicator of confidence the largest probability in the little model predictions P. Intuitively, a large top probability indicates a class that is far more likely than the others and a model that is confident in its prediction. The computational cost of this policy is only O(M), where M is the number of classes, as it requires M comparison to be performed at runtime. However, this approach has also significant limitations. For instance, in a 4-class classification task, given an output $P = [0.4, 0.4, 0.1, 0.1]$, the obtained max score would be $S = 0.4$. While this value is far from the random guess, the classifier is not confident, as $P_0 = P_1$. Nonetheless, a policy based on the max score may still trigger the early stopping, thus degrading the final accuracy.

The *Score Margin* (SM) [40, 39] is a confidence metric that tries to solve the aforementioned issue, as it considers also the second most likely probability in P, computing its value as:

$$sm = max(P) - max_{2nd}(P) \tag{2.5}$$

This metric, while requiring twice as many comparisons as the Max Score, is more robust. In the example above, while $S = 0.4$, we obtain $SM = 0$, clearly indicating that the model is not confident about the prediction. As a consequence, the Score Margin has rapidly become more popular than the Max Score in the literature [40, 39].

At inference time, the early stopping policy is computed after running M_s and then compared with a user-defined threshold th. The latter controls the energy versus accuracy trade-off, as altering its value either increases or decreases the frequency of activation of M_l and can be changed according to any external conditions (e.g., battery lifetime).

Noteworthy, the score margin and the max score are identical in the case of binary classification, as the largest probability is the complement of the smallest. Moreover, the big-little mechanism can be easily extended to a cascade of K models, with $K > 2$, computing the early stopping mechanism after each model before the final one.

Another dynamic approach named *Early-Exit* [58] is shown in Figure 2.6, introducing additional possible exit points to CNNs, named *branches*. At the end of each

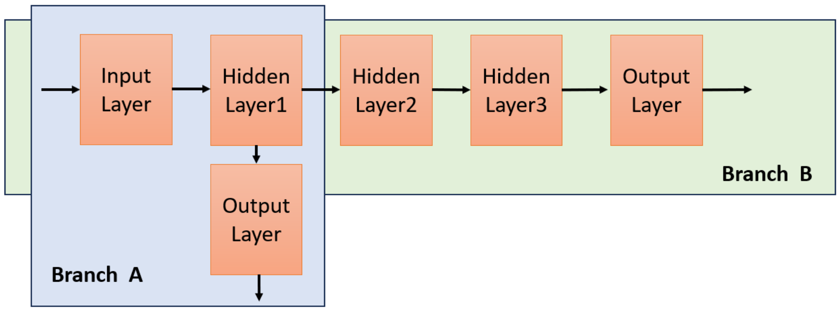


Fig. 2.6 Overview of the early-exiting mechanism

intermediate branch, an adaptive policy is used to determine whether the output of the branch is reliable enough (e.g., using the score margin). In the positive case, the output of the branch is taken as the final one. In the negative case, the computations restart from the last shared branch of the computational graph.

Early-exit mechanisms, while introducing lower memory overhead than cascades of DNNs, may lead to reduced performances, as some critical features of the input sample may be extracted only by deeper convolutions.

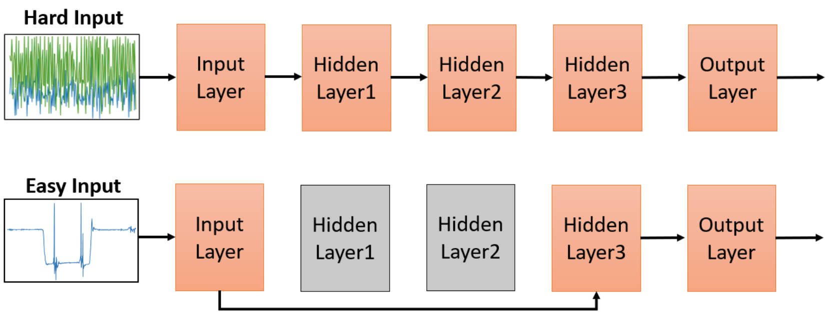


Fig. 2.7 An example of a layer skipping mechanism.

Lastly, Figure 2.7 shows the *layer skipping* approach proposed in [41], introducing a mechanism to skip the computation of a subset of the layers depending on the input sample. To do so, small DNNs named *gates* are added before layers, with the task of determining whether the following layer/s have to be enabled. Noteworthy, gates are significantly more lightweight than the corresponding set of layers they control in the main architecture.

2.3 IoT End Nodes

IoT end-nodes are often based on low-power microcontrollers (MCUs), featuring a general-purpose CPU based on a RISC instruction set. In particular, the low price and great flexibility of MCUs make them an ideal candidate for low-cost low-power edge devices. On the contrary, Application-Specific Integrated Circuits (ASICs) or Field-ProGrammable Arrays (FPGAs), while significantly more energy efficient, feature prohibitive costs, making their use feasible only for high-end edge nodes.

To bridge the performance gap w.r.t. ASICs, modern MCUs, and Systems-on-Chips (SoCs) have started introducing specific architectural features tailored for efficient executions of ML or DL workloads at the edge while preserving programmability, generality, and limited cost. An increasingly popular architectural advancement introduced specifically for ML is the presence of custom instruction sets, often based on the extensible RISC-V ISA, as for the XPulp extension [59]. The latter implements vectorial operations such as single-latency dot-products of 4 8-bit elements, hardware loops, and bit-extraction primitives for sub-byte quantization, yielding instruction reductions of up to 9x for 8-bit convolutions.

Examples of architectures tailored for ML workloads are already present in the industry, for instance by STM [60], NXP [61], and GreenWaves [62]. In particular, GAP8 from GreenWaves is a commercial SoC based on the PULP-paradigm featuring a single RI5CY core that orchestrates I/O and sensors, and a cluster of 8-core RI5CY processors that can be used to accelerate computations such as Multiply-And-ACcumulate (MACs) operations.

Chapter 3

Datasets and Deployment Targets

In this chapter, I introduce the datasets and the deployment platforms used as benchmarks for the optimization algorithms described in this dissertation.

3.1 Datasets

In this work, I employ 7 state-of-the-art public datasets and a private one, provided by STMicroelectronics. The aforementioned datasets have been selected because they feature classification tasks popular in real-world embedded scenarios, they are well-documented and present enough samples both for training classical ML learning algorithms and data-hungry DL architectures. Finally, these datasets feature different types of data (time series, images) and complexity.

3.1.1 Time-series datasets

The first task I consider in my work is Human Activity Recognition (HAR) based on Inertial Measurement Units (IMU), as it is an increasingly common feature of smart devices such as wearables. The latter, often based on MCUs, are an ideal target for the optimizations proposed in this work. The HAR task consists of classifying the activity performed by a user in a given time window of IMU readings and is commonly addressed with ML or DL algorithms. In this dissertation, I use 4 HAR datasets: WALK, WISDM, UCI HAPT, and UniMiB-SHAR.

WALK, a private dataset from STMicroelectronics, features 2387232 readings from a tri-axial accelerometer, sampled at 25 Hz. The position of the sensor changes depending on the sample, ranging from the backpack of the subject to a pocket. The subjects have been recorded during one out of seven possible activities: sitting in a car/bus, cycling, riding on a bike, standing still, walking, or running. In this work, the dataset task is binary, consisting of recognizing whether the subject is walking or not in a given time window. The test set is 20% of the available data.

WISDM [63] includes 1.098 million sensor readings collected from a tri-axial accelerometer sensor at 20 Hz. The data was collected by a smartphone located in the front pant legs pocket of 29 subjects while performing one out of 6 daily life activities: walking, jogging, sitting, standing, descending stairs, or ascending stairs. The data pre-processing is left unchanged w.r.t to the one proposed by the authors, that is, extracting non-overlapping windows of 10s. The test set is composed of 10% of the total samples.

UCI HAPT [64] features 958500 samples collected from the gyroscope and accelerometer of an Android phone. The data is collected with a sampling frequency of 50 Hz, repeating the experiments on 30 subjects. The activities recorded for this dataset are 12. The pre-processing of the data mirrors the one proposed by the dataset authors, with the samples divided into windows of 5s with no overlap and per-subject train-test split, with 1/3 of the subjects being in the test set. Notably, this dataset represents an interesting benchmark due to the presence of the gyroscope data.

UniMiB-SHAR [65] consists of 11711 samples recorded from the tri-axial accelerometer of an Android phone. The device has been positioned in the front leg pocket of the 30 subjects while recording one out of 9 daily life activities or one out of 8 kinds of falls (e.g., falling backward from a standing/sitting position). The data collected at 50 Hz is provided already pre-processed in windows of 3s, extracted around accelerations peaks larger than 1.5 g (with g being the gravitational acceleration). The authors propose several variant tasks with different complexity (e.g., considering/excluding falls, grouping activities, etc.). In this work, only *AF-17* is used, that is the variant that considers all 17 classes. As proposed by the authors, I use 20% of the data for the test set.

Noteworthy, the aforementioned datasets present noticeable differences both in terms of input size (from windows of 32 samples to 250, per channel) and complexity (from 2 to 17 classes), making the models required to handle them significantly different and thus providing an interesting comparison.

Besides HAR datasets based on IMU readings, in this work, I also employ Ninapro DB1, a state-of-the-art dataset for gesture recognition that leverages surface Electromyography (sEMG) signals. Specifically, this dataset features sEMG data collected from 27 healthy subjects while performing different hand movements. In this work, I mirror the pre-processing proposed in [66], limiting the classification task to 14 hand movements with a 10-channel sEMG signal. The signals are divided into windows of 150 ms, and collected at 100 Hz, yielding a dataset with around 207000 samples. The training is performed per patient, selecting different sessions for the training, test, and validation sets.

The last dataset that features time-series data is *Backblaze*, whose task is anomaly detection. In particular, *Backblaze* [67] contains 19 Self-Monitoring Analysis and Reporting Technology (SMART) features collected daily from hard disks during their lifetime in a data center, from 2014 to 2019. The goal is to determine whether a failure will happen in the following 7 days. In this work, the setup from [68] is mirrored, extracting 90-days windows and thus obtaining around 707k elements.

3.1.2 Image-based datasets

In this dissertation, two state-of-the-art CV datasets are used as benchmarks: *Cifar10* [69] and *German Traffic Sign Recognition Benchmark* (GTSRB) [70].

Cifar10 is a popular dataset for image classification algorithms, featuring 60000 32x32 color images divided into 10 classes, each representing a different category (e.g., animal or transportation method) or object (horses, frogs, trucks, ships, etc.). The class frequency is balanced, each featuring 6000 images, with 5000 used in the training set.

GTSRB is a dataset consisting of 50000 images of 43 different real-world traffic signs. It is a common benchmark for autonomous driving tasks, specifically traffic sign recognition, featuring RGB images collected with different resolutions (from 35x35 to 100x100 pixels) and different lighting conditions. The class distribution is unbalanced, with traffic signs such as the 20 km/h speed limit appearing less than 500 times in the training set as opposed to its counterpart of 30 km/h represented more than 2000 times. Intuitively, the uneven class frequency and the high number of classes make the traffic sign classification task significantly more complex than the one of *Cifar10*. For this dataset, as CNNs commonly require a fixed-dimension input, we rescale all the images to 60x60.

3.1.3 Speech Recognition

Wake-word recognition is a common task for embedded devices and it has already been implemented in several commercial smart speakers. For this reason, in this work, I benchmark one of the presented optimizations on *Google SPeech commands* (GSP) dataset, featuring 65000 one-second-long recordings of 30 words. In this work, GSP is pre-processed as in [71], reducing the number of classes from 30 to 12, with one being a fallback class for unknown words, and then extracting a 32x32 spectrogram from each audio signal.

3.2 Deployment Targets

In the following paragraphs, I introduce the MCUs and SoCs used as deployment targets in this work.

3.2.1 STM32H743

STM32H743 [60] is an MCU produced by STMicroelectronics, based on the Arm Cortex - M7 core, a general-purpose core equipped with an FPU for floating point computations. This MCU is designed to maximize computing capability for general-purpose workloads while still having a tight power envelope. It includes two caches to enhance performances, at the cost of an increased energy consumption. Specifically, the two caches are used one as an instruction cache and the other as a data cache, reducing the time required to fetch instruction and move data to registers. DMA peripherals are available to support moving data from sensors or external memories to the core rapidly. The M7 core can reach a frequency of up to 480 MHz, with a power consumption of 234 mW.

The MCU is equipped with several peripheral interfaces, such as UART, I2C, and USB. While less energy efficient than the PULP family of SoCs for ML and DL workloads, it allows an automatic deployment of models through X-Cube-AI. The latter allows exporting and deploying the high-level trained model directly on the device, moving the complexity away from developers.

3.2.2 Pulpissimo

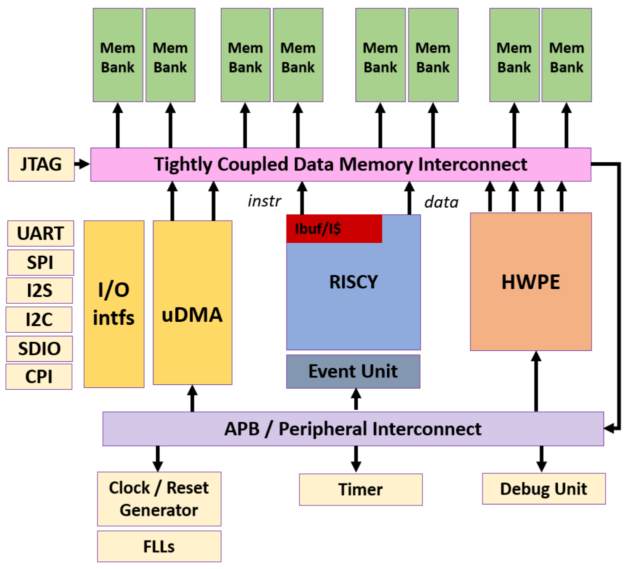


Fig. 3.1 Pulpissimo Block Diagram

Pulpissimo [72] is a 32-bit single-core RISC-V MCU that belongs to the Parallel-Ultra-Low-Power (PULP) family of architectures. It is based on a RI5CY core with a 4-stage, single-issue, in-order pipeline and implements an RV32IMC ISA enhanced with domain-specific extensions for DSP, such as hardware loops, loads/stores with index increment and Single Instruction Multiple Data (SIMD) operations. These features have been designed to provide significant speedups for ML applications. In this dissertation, I refer to a 22 nm implementation of Pulpissimo named Quentin [72] (shown in Figure 3.1), set to run at 205 MHz and equipped with 512 kB L2 memory. Quentin's memory layout is organized as 4 114 kB word-level interleaved banks that minimize the conflicts during accesses performed through the masters. Moreover, it includes 2 32 kB private banks that can be used to store the program and the stack so that banking conflicts can be avoided. This System-on-Chip (SoC) includes a full set of peripherals, ranging from Quad Serial Peripheral Interface (SPI), a parallel camera interface, UART, GPIOs, JTAGs, and a DDR HyperBus interface. It also includes an I/O Direct Memory Access (DMA) that manages data transfers through peripherals so that the workload on the processor is minimized. All inference cycle results have been estimated with GVSoC [73], a cycle-accurate virtual platform. The energy values instead are derived from [74].

3.2.3 GAP8

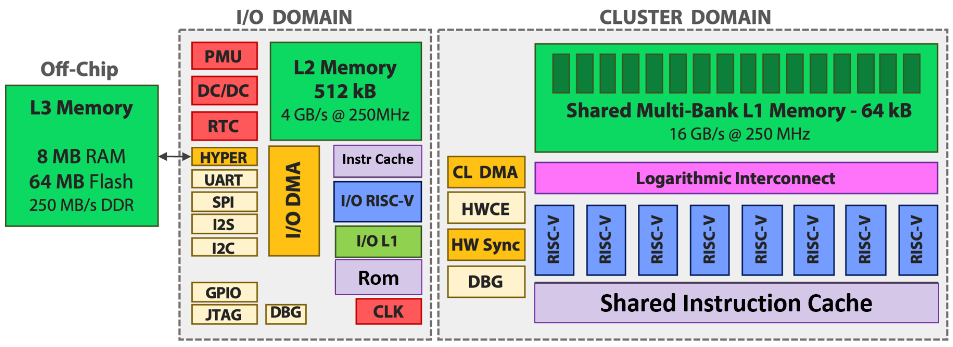


Fig. 3.2 GAP8 Block Diagram

GAP8 [62] (shown in Figure 3.2) is a commercial System-on-Chip (SoC) from GreenWaves. It features 9 extended RISC-V cores (one used for I/O and 8 as cluster), embodying the Parallel-Ultra Low Power (PULP) paradigm, one of the most advanced architectural advancements to accelerate ML workloads on MCUs. Each core is a 4-stage single-issue pipeline in-order RI5CY [75] core, exploiting the custom RISC-V RV32IMCXPulpV2 [59] instruction set architecture. The cluster cores share the first level of memory, a 64 kB Tightly-Coupled Data Memory (TCDM) that can be accessed through a single-cycle latency and high-bandwidth logarithmic interconnect. Data transfers between L1 memory and the second-level memory of 512 kB (also managed as scratchpad) are handled by the cluster DMA with a bandwidth of up to 2 GB/s. Optionally, an L3 off-chip memory can be equipped to further extend the storage capability of the SoC, but it is not employed in this work. All inference cycle results have been estimated with GVSoC [73].

Chapter 4

Optimization Methods for Tree Ensembles

Tree ensembles are considered a lightweight alternative to deep learning [76], as on simple embedded tasks they can achieve comparable accuracy with lower memory footprint and operations per inference [77]. Nonetheless, accurate ensembles still deploy hundreds of trees, requiring thousands of cycles per inference on ultra-low-power IoT devices or worse, saturating the deployment target memory. As a consequence, even lightweight models such as RFs and GBTs benefit from software optimizations, whose goal is trading off as little accuracy as possible for large energy/memory savings.

In this chapter, first, I provide an overview of the current landscape of tree ensembles' optimizations, from static/dynamic techniques to efficient implementation and deployment libraries. Then, I introduce my contributions on this topic, focusing on an implementation and related optimizations designed specifically for efficient inference at the edge.

The work described in this chapter has been published in [11, 78, 79] and has been made open-source at the link `https://github.com/eml-eda/eden`.

4.1 Related Works

4.1.1 Static Optimizations

Statically pruning trees or branches in DTs is among the most common memory and energy optimization approaches for ensembles. For instance, *cost complexity pruning* [21] removes the less meaningful branches (e.g., rarely taken) from DTs, leading to better generalization and fewer parameters to be stored.

At the ensemble level, pruning approaches consist of removing less accurate [38] or similar [80] (e.g., in terms of output predictions) trees. In [38, 80], the authors show that besides reducing the memory footprint of the ensemble, pruning also reduces the latency and energy per inference, achieving iso-accuracy w.r.t. an unpruned ensemble with a reduction of more than 84% of the trees. However, these works often consider large starting ensembles (e.g. $\simeq$ 300 DTs), obtaining, post-pruning, around 50-100 trees, that considering an edge AI scenario still lead to cumbersome inference latency and energy, and large memory requirements. More aggressive forms of pruning, while further reducing resource utilization, sharply deteriorate the ensemble's accuracy, leading to sub-optimal performances. Noteworthy, these algorithms are orthogonal to the ones I introduce in this work and can be still used to statically reduce the dimension of the ensemble.

4.1.2 Dynamic Inference

Dynamic inference implementations tailored for tree ensembles have been rarely proposed in the literature, with most works focusing on deep learning [39, 41, 81, 40]. In [82], the authors introduce an early-stopping criterion that models the prediction confidence after a multinomial or binomial distribution (according to the number of classes). Once a suitable subset of DTs of the ensemble has been executed, they halt the inference. With benchmarks on seven small public datasets and a private one, the authors show reductions in terms of mean executed trees per inference of up to 63%. Nonetheless, to efficiently compute the distributions at runtime, this approach requires the storage of large lookup tables, growing in the order of O(N^2) w.r.t the number of estimators.

The authors of [83] introduce an approach that at runtime determines the optimal order of execution of the weak learners according to the most likely class predicted by the already executed trees. Finding the optimal tree ordering takes into consideration both the accuracy and computational cost of a weak learner, as it may rely on features that have not yet been computed. Additionally, the authors design a probabilistic model of the ensemble, based on a mixture of Gaussian distributions, that is used to determine at runtime whether an early stop can be triggered according to the posterior probabilities. Even though the authors introduce a dimensionality reduction technique to limit the computations required to select the next DT to execute, the additional overhead of such an approach is complex to handle on MCUs. In fact, according to the authors, this approach is feasible only when relying on complex feature extractions, a rare occurrence in IoT applications [83].

Finally, the authors of [43] propose a work that is close to the optimizations I introduce, named *Quit When You Can* (QWYC). Specifically, two probability thresholds are extracted for each weak learner after training, named ε_+ and ε_-, and are used as boundaries to trigger the early stop in binary classification tasks. Moreover, a static sorting algorithm for DTs that minimizes the average number of executed trees is introduced, running first the weak learners that trigger the early stop more often. Nonetheless, the authors evaluate QWYC only on binary tasks and provide no deployment results, leaving the effectiveness of this approach unexplored for IoT devices.

4.1.3 Deployment Libraries for MCUs

Due to the popularity of tree ensembles in ML applications, multiple implementations have been proposed. Libraries such as [84–86] introduce implementations of tree ensembles with optimizations specifically tailored for high-end hardware.

The authors of [86] provide an implementation in C++ of the RF algorithm. DTs support only floating point thresholds (α) and inputs, with leaves storing only the most likely class instead of an array of class probabilities. Additionally, their framework supports both training and inference and leverages a complex object-oriented representation of the trees.

In [85], the authors introduce a C++ implementation of both RFs and GBTs, representing the trees with a structure storing pointers to the children, alphas, and other additional fields. Their implementation supports threshold quantization, although only post-training and at 32-bit integers.

Finally, in [84], the authors propose an implementation of the DT data structure that closely mirrors the one in [24]. That is, storing in arrays the right/left child indexes, the class values, the alphas, and the feature indexes of each node. No support for quantization is provided.

The aforementioned libraries, while optimized for efficient inferences, do not prioritize reducing the model memory footprint, a crucial challenge on IoT devices. Moreover, as they often support both training and inference, their code size becomes unnecessarily large. As a consequence, using them for deployment on MCUs, while feasible, would be sub-optimal.

Libraries specifically tailored for IoT devices are introduced in [76, 87]. The library introduced in [76] is targeted for GAP8, as the one presented in this work, and leverages a similar array-based representation. However, it does not support dynamic inference and it provides no optimal buffer allocation on the different levels of the memory hierarchy. Moreover, it does not support quantization. Finally, the authors of [87] benchmark different implementations of the RF inference algorithm, ranging from recursive to fully unrolled trees, on Pulpissimo. Additionally, they test several compiler-level optimizations and different tree data structures, obtaining up to 4x speedups w.r.t. an unoptimized inference. Nonetheless, in this case, as well, no quantization or dynamic inference is performed.

4.2 EDEN: Efficient Decision tree ENsembles

In this section, I focus on software optimizations for efficient inferences of tree ensembles, targeted either at reducing the memory or the latency/energy footprints. Specifically, the experiments focus on GBTs and RFs, introduced in Chapter 2, as common in several embedded scenarios. Noteworthy, the proposed algorithms are not strictly tied to these models and can work on other tree ensembles for classification tasks, such as LightGBM [23].

This section is structured as follows. First, I introduce a lightweight dynamic inference approach tailored for tree ensembles, extending the one introduced in Chapter 2.

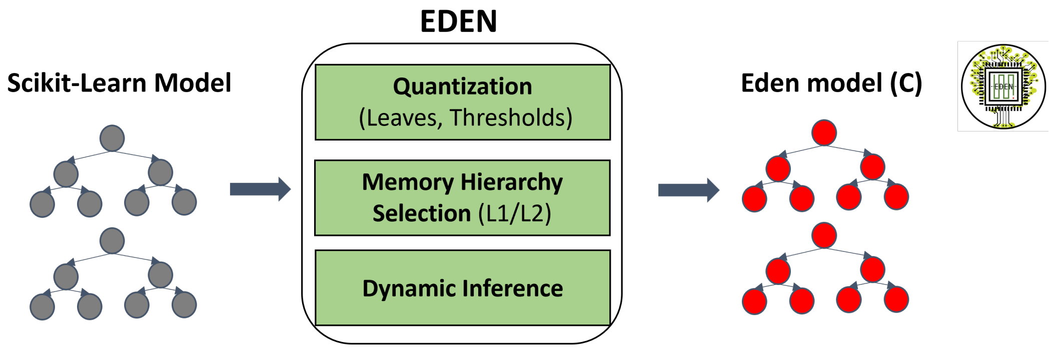

EDEN
Scikit-Learn Model
Quantization (Leaves, Thresholds)
Memory Hierarchy Selection (L1/L2)
Dynamic Inference
Eden model (C)

Fig. 4.1 EDEN library overview. Starting from the Python model, a C model tailored for efficient inference on MCU is generated.

Then, I detail the specific optimizations for MCUs, with a focus on multi-core IoT devices featuring a multi-level hierarchy memory such as GAP8.

Finally, I provide an extensive set of benchmarks, highlighting the effectiveness of the introduced optimizations and comparing GBTs and RFs in an embedded scenario.

Noteworthy, I included all the introduced optimizations in an open-source Python library named EDEN, whose flow is shown in Figure 4.1. Starting from the high-level tree ensemble trained with a general-purpose library such as [24], my library exports a set of C files that can be directly compiled on the target MCU, implementing a target-optimized inference. While the experiments in this section refer only to GAP8, as it represents the most challenging implementation, EDEN supports both Pulpissimo and standard x86 targets.

4.2.1 Dynamic Tree Ensembles

Iterative dynamic inference approaches such as the Big-Little mechanism introduced in Chapter 2 are based on the assumption that the models in the cascade are increasingly accurate [39]. Therefore, given N models, it makes sense to consider only the prediction of the last one executed P^t to compute the early stopping policy, as its output is more reliable than $P^{(t-1)}$.

However, this assumption does not hold in tree ensembles, where the weak learners possess similar accuracy. Therefore, leveraging only P^t, i.e. the last executed DT or estimator, becomes sub-optimal. As a consequence, I propose an enhanced version of the early stopping policies introduced in Section 2.2.3, that is Max Score (S) and Score Margin (SM), leveraging the accumulated predictions $P^{[1:t]}$ of the trees already executed.

The strength of this approach lies in the fact that the accumulated probabilities skew in favor of a class already after running a few trees. In this case, it rapidly becomes unlikely or mathematically impossible for the leftover trees to overturn the current predicted class, making their execution pointless and energy-inefficient. The partial output probabilities of an RF up to the t*th* weak learner can be defined as follows:

$$P^{[1:t]} = \sum_{i}^{t} P^{i} \tag{4.1}$$

Then both confidence metrics, Max Score and Score Margin can be redefined using the aggregated probabilities. Specifically, the Max Score (S) becomes the Aggregated Max Score (S^t), defined as follows:

$$S^t = max(P^{[1:t]}) \tag{4.2}$$

and the Aggregated Score Margin (SM^t) becomes:

$$SM^t = max(P^{[1:t]}) - max_{2nd}(P^{[1:t]}) \tag{4.3}$$

Noteworthy, GBTs represent a significant difference, as the DTs composing them are *regression* trees, whose output is converted into probabilities through a computationally expensive formula. On MCUs, performing this conversion after each estimator would lead to significant overhead, outweighing any possible energy gain obtained with the early stopping. Therefore, I exploit the fact that the conversion formula is monotonically increasing and perform the confidence estimation directly on the raw predictions.

Figure 4.2 shows an overview of a dynamic inference for an RF with $N = 3$, $M = 3$, and $D = 3$, where M and D are respectively the number of classes and depth. The decision path of a hypothetical input in the weak learners is highlighted in red, while the batch size (B), detailed in the following sections, is set to 1. The confidence metric used is the Aggregated Score Margin, evaluated on the accumulated output probabilities after executing a weak learner. When the condition $SM^t > th$ is met,

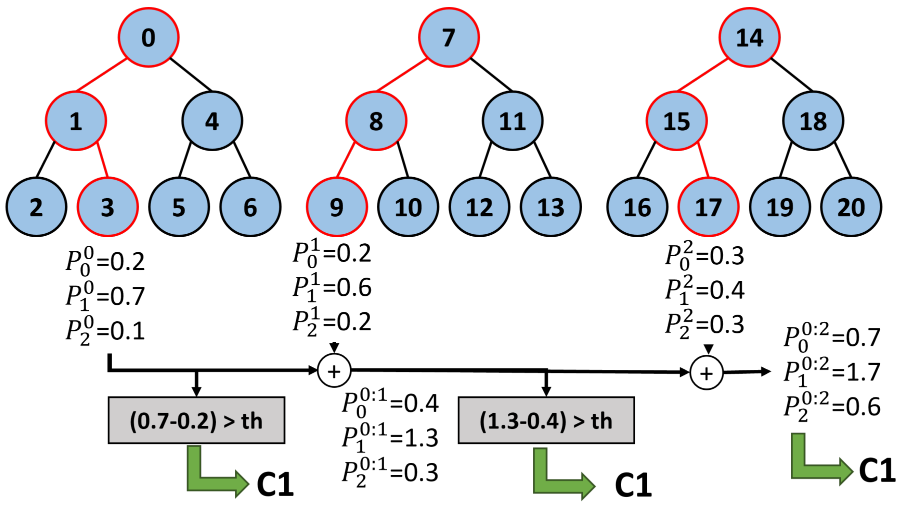


Fig. 4.2 Dynamic Inference for an RF with M=3, N=3, and D=3 using the Aggregated Score Margin as early stopping policy.

the early stopping is triggered, and the most likely class C, extracted with an argmax operation, is used as the final prediction.

Note that the main challenge in deriving an early stopping policy for tree ensembles lies in the time complexity of the approach. Introducing an overhead (i.e., the policy computation) that is comparable to the execution of the following model in the cascade may in fact lead to large overheads in terms of latency. Intuitively, this is rarely an issue when working with a cascade of computationally intensive DNNs, but can become a bottleneck in a cascade of shallow trees (i.e. M » D). Therefore, the limited complexity of both the Aggregated Max and Aggregated Score Margin proves to be extremely beneficial, as opposed to complex policies implemented in other works in the literature. Noteworthy, the overhead introduced by the policy is generally not considered in hardware-unaware setups, using metrics such as the average number of executed trees.

4.2.2 Deploying on MCUs

Ensemble structure

Taking inspiration from the RF implementation proposed in the OpenCV library [86], I designed a more compact and efficient data structure to store DTs and ensembles

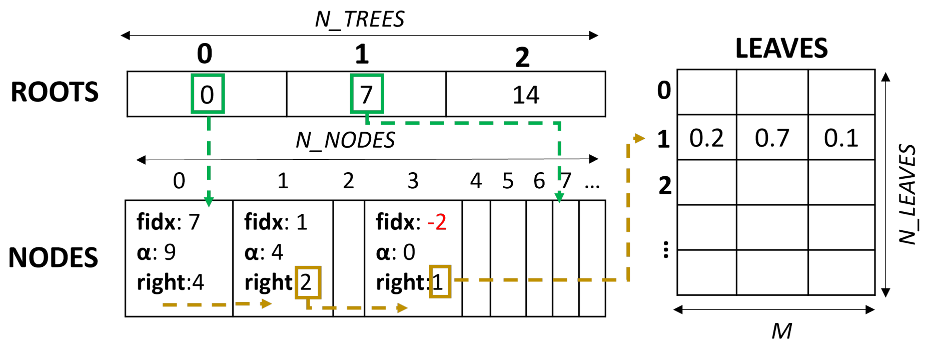


Fig. 4.3 Data structure of a tree ensemble. The arrows show the inference steps for the first tree of Figure 4.2.

on MCUs. Specifically, lists are substituted in favor of C arrays, for the improved data locality and compactness.

Figure 4.3 shows the three main structures used to store an ensemble. The arrows highlight instead the element visited to run an inference with the first tree of Figure 4.2.

The core element is the NODES array, composed of C *structs* that store in separate fields all the information regarding a DT node. Each struct element in NODES features the following fields:

- $fidx$: stores the index of the feature in the input buffer used for the branching operation. At runtime, the value indicated by fidx is compared with the threshold α to determine the next node to be visited. Leaf nodes have this field set to -2, for compatibility with the training library [24].

- α: the threshold to be compared against the element stored at position $fidx$ in the input buffer. If the latter is smaller or equal to the former, the left child is selected. In the opposite case, the right child is reached.

- $right$: the offset in NODES between the index of the current node and the index of its right child. Leaf nodes reuse this field to store the index of the row in the LEAVES matrix that stores their class probabilities.

The LEAVES array stores the class probabilities of all the leaves in the ensemble, while ROOTS stores the index of the root node in NODES for each tree in the ensemble. The latter is necessary to achieve a fast iteration of all DTs stored at inference time. Notably, this implementation does not store the index of the left child, significantly reducing the memory footprint of the model. Instead, the nodes of each DT are stored in pre-order, so that the left child of each node is always the next element in the NODES array.

An additional optimization pass is performed for GBTs and RFs for binary tasks. As GBTs are composed of regression trees, their leaves store a single scalar value. Therefore, the LEAVES data structure is removed and the value belonging to each leaf is stored directly in the α field. The same optimization can be applied to RFs for binary classification tasks, as leaves store a single class probability (since $P_0 = 1 - P_1$).

Algorithm 4 Static multi-class RF inference pseudo-code

```
run_tree(t, P, INPUT, ROOTS, NODES, LEAVES) {
  if (core_id == (t%C)) {
    n=NODES[ROOTS[t]];
    while(n.fidx != -2) {
      if(INPUT[n.fidx]>n.alpha) n+=n.right;
      else n=n+1;
    }
    lock_in();
    for(j=0;J<M;j++) P[j]=P[j]+LEAVES[n.right][j];
    lock_out();
  }
}

P = {0};
parallel for (t=0; t<N; t++)
    run_tree(t, P, INPUT, ROOTS, NODES, LEAVES);
barrier();
if(core_id == 0) res = argmax(P);
```

Algorithm 4 reports the inference pseudo-code for a single DT of a multi-class RF (the *run_tree* function), as implemented by the EDEN library. The function loops on the tree nodes until the exit condition ($fidx == -2$) is met, meaning that a leaf has been reached, and then updates the prediction buffer P with the values in leaves. C denotes the number of cores of the target MCUs and is explained in detail in Section 4.2.2.

Quantization

As introduced in Chapter 2, quantization is a common optimization for DNNs when deployed on edge nodes, often possessing slow or no FPUs. On the other hand, concerning tree ensembles, quantization is less explored, generally deploying the models using floating point thresholds or with 32-bit integer quantization in a few cases. Nonetheless, a more aggressive integer quantization may prove beneficial also for this kind of model, leading to significant reductions in memory.

In the case of decision trees and ensembles, three elements are feasible targets for quantization: *alphas*, *inputs*, and class probabilities. However, as the first two are directly compared, they should be quantized using the same precision and format.

As for DL, quantization for inputs/alphas can be applied either post-training or at training time (through a *quantization-aware training*). I apply the latter, as it commonly leads to lower accuracy drops [28]. To do so, the input data is quantized with a symmetric quantizer (introduced in Chapter 2) before starting the training process. The integer inputs are then fed to the training library [24], which will still derive float α values. Nonetheless, as inputs are integers, it is sufficient to truncate the fractional part of the thresholds, leaving the decision unaltered.

Class probabilities (or leaf values in case of regressions) can be quantized post-training, similarly to DNNs' weight tensors, as their range is fixed and known after fitting.

Memory-Layout Selection

Modern IoT nodes often implement complex multi-level memory hierarchies, providing developers a direct control of the data transfers between memories. For instance, GAP8, the target platform for this work, features a small yet fast L1 memory and a slower but larger L2 memory. Maximizing the L1 usage, while crucial for efficient inferences, is also challenging. For instance, RFs such as the one used as a benchmark in this work are too large to fit in the small L1 of GAP8 (64 kB).

DL libraries for edge deployments [35] leverage the regularity of DL computations to load dynamically in L1 only parts of the data, needed to execute a small part of the computations. Such an approach, named *tiling*, is effective because it is a compile-time decision, and even more importantly the data (e.g., convolution weights for different parts of the input) once moved in L1 is often re-used multiple times. On the contrary, tree ensembles feature data-driven computations (the visited nodes depend on the input data) that cannot be optimized at compile time. Moreover, the access ratio of the NODES structure is logarithmic, requiring the transfer to L1 of up to 2^D elements and accessing at most D elements. The LEAVES data structure features an even sparser access ratio of its elements, with only 1 row per 2^D elements being accessed per inference. As a consequence, a tiling approach on the tree data would be sub-optimal, leading to large transfers of unused data and introducing a significant overhead.

Nonetheless, arrays such as INPUT, P, and ROOTS are accessed multiple times, and storing them in L1 at deployment time can lead to faster inferences. Selecting the best data structure to store in L1 becomes then an optimization problem that can be solved at deployment time. In particular, I tackle this problem with a Knapsack 0/1 algorithm, a dynamic programming algorithm used to efficiently select a subset of items with given weights and values, subject to a limit on the total weight.

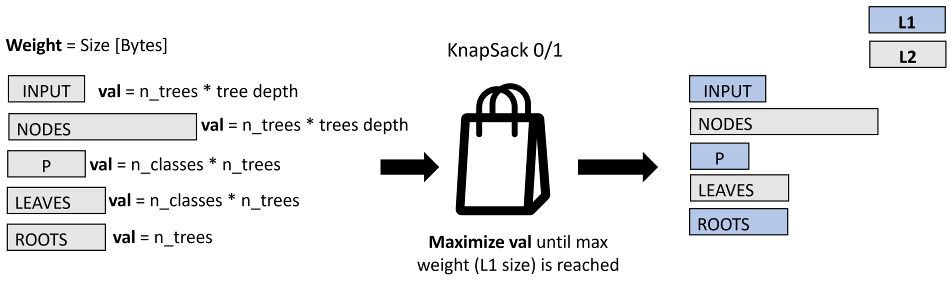

Weight = Size [Bytes]
INPUT
val = n_trees * tree depth
NODES
val = n_trees * trees depth
P
val = n_classes * n_trees
LEAVES
val = n_classes * n_trees
ROOTS
val = n_trees
KnapSack 0/1
Maximize val until max weight (L1 size) is reached
L1
L2
INPUT
NODES
P
LEAVES
ROOTS

Fig. 4.4 Knapsack 0/1 algorithm applied to the data structure of the ensemble. Each array is assigned to a memory level (L1,L2) depending on the accesses per inference (val) and size in memory (weight).

Figure 4.4 shows an overview of the approach, where the total weight is the L1 size and each array possesses a weight equal to its dimension in bytes. The value of an array is given by the average accesses to the array for an inference on a single input. Noteworthy, this approach can be scaled to multiple memory levels (e.g. L2 vs L3) and can easily extend additional data structures needed by the tree ensembles, making it flexible for future extensions. Furthermore, it requires changing only a constant (the L1 size) to automatically support other deployment targets.

This allocation pass in the deployment leads to significant performance improvements w.r.t. to a tree-wise tiling (i.e. moving all the data required by a tree

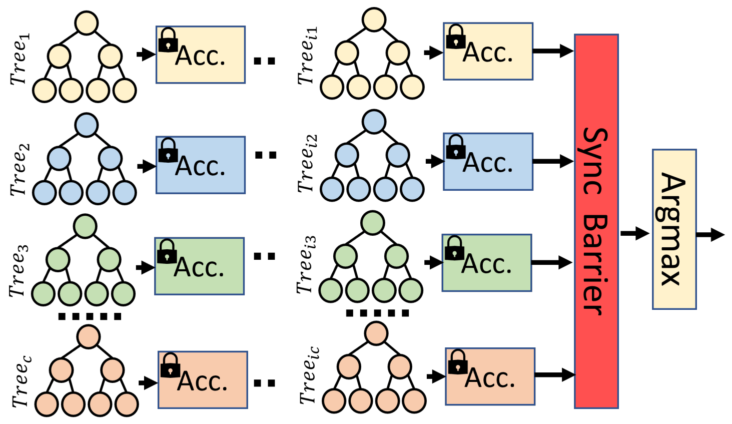


Fig. 4.5 Multicore static inference overview. Cores are represented with different colors.

dynamically in L1). Note that this can be entirely avoided when the target SoC features hardware-controlled caches.

Multicore Inference

In this work, I take inspiration from the implementation of RFs proposed in [76] to derive the RF and GBT static inference implementation. A high-level overview is shown in Figure 4.5 for a RF. Trees are statically assigned to a core (each represented with a different color) according to their index in the ensemble. At inference time, trees are executed in parallel (see Algorithm 4) while updates to the shared P vector (*Acc.* in the Figure) are performed through mutual exclusive access (depicted with a lock in the Figure and with *lock*/*unlock* in Algorithm 4), implemented with a lock.

The final step consists of a synchronization barrier, ensuring that all trees have been executed. Afterward, only Core0 performs an argmax operation on P. Noteworthy, in the case of GBTs, trees belonging to different estimators can be run in parallel. Moreover, as global synchronization is required only at the end, there is no constraint on the order of execution of the trees.

Concerning dynamic inference, previous works always assume a sequential execution of the trees [83, 82], evaluating their early stopping policy after each of them. Intuitively, forcing a sequential inference on a multi-core setup is sub-optimal, as it would lead to a significant under-utilization of the available resources. For instance, with C=8, evaluating the policy after 1 or 8 DTs consumes a similar amount

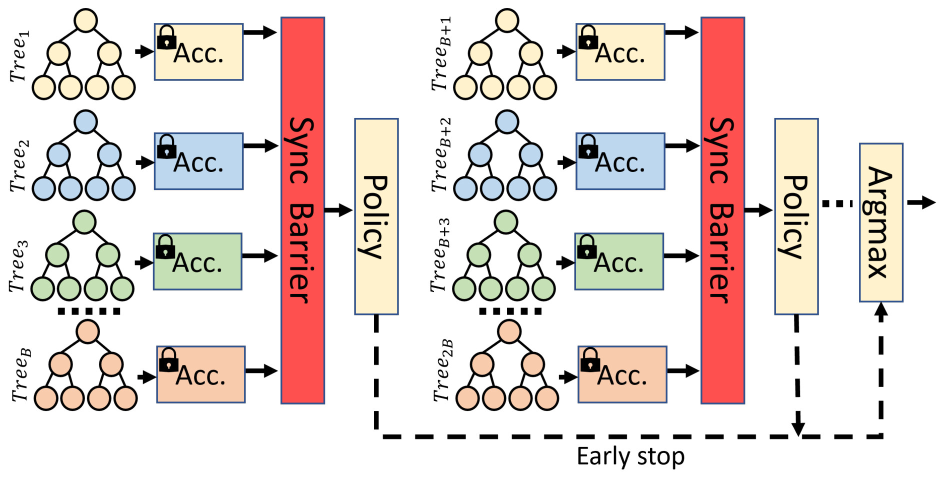


Fig. 4.6 Multi-core dynamic inference of an ensemble.

of energy, however, with the first option we ignore the output of 7 trees, which leads to a more accurate decision for the early stopping.

Algorithm 5 Dynamic multi-class RF inference pseudo-code

```
1  P = {0};
2  t = 0;
3  stop = 0;
4  for(int bt=0; bt < POLICY_CALLS && !stop; bt++) {
5    for (int i = 0; i < B; i++)
6      run_tree(t++, P, INPUT, ROOTS, NODES, LEAVES);
7    barrier();
8    if (core_id == 0)
9      stop = policy(P) > th;
10   barrier();
11 }
12 if (!stop) {
13   while(t<N)
14     run_tree(t++, P, INPUT, ROOTS, NODES, LEAVES);
15 }
16 barrier();
17 if (core_id == 0) res = argmax(P);
```

As a consequence, I introduce a configurable *batching* mechanism, represented in Figure 4.6 and Algorithm 5. Specifically, the early stopping policy is not evaluated after the execution of each tree, but rather after B trees, where B is the batch.

This is clearly shown in Algorithm 5, where the external loop starting at row 4 corresponds to the execution of a batch (a column in Figure 4.6) and is repeated

POLICY_CALLS times, a constant statically computed at compile time. The loop at line 13 computes the leftover DTs when N is not a multiple of B. The *policy* function computes either SM^t or S^t.

Compared to a static inference, the dynamic one features an additional synchronization barrier after each B tree, ensuring that all the required updates to $P[1:n]$ have been performed before letting Core0 handle the policy evaluation. If the early stop is triggered, then the final argmax is performed. The second barrier introduced at line 10 is needed to ensure that all cores have performed the required early exit before resuming with the new batch execution. Nonetheless, this synchronization overhead is often minimal, as shown in the results. Concerning GBTs, the early-stopping policy can be triggered only after executing an estimator, i.e. M trees.

In this work, I set $C = B$, to ensure that all cores are fully utilized during the dynamic inference. Nonetheless, setting B to be any multiple of C can lead to additional tradeoffs, as it balances the policy overhead w.r.t. the execution cost of the trees. For instance, for ensembles with $D < M$ and $C = 1$, the evaluation of a policy such as the SM can take more than the tree execution. Therefore, the gains in terms of energy may be overshadowed by the additional computations required for the early stopping policy. Setting $B > 1$ in this case forces the execution of multiple trees before evaluating the policy, balancing the overhead.

4.2.3 Experimental Setup

The optimizations introduced in the sections above are benchmarked on 3 datasets: Ninapro DB1, BackBlaze and UniMIB-SHAR. Noteworthy, these datasets have been selected because they feature different complexity in terms of the number of classes. Intuitively, as M grows larger, the overhead to compute a metric such as SM^t increases and becomes comparable to the inference time for a single tree. Therefore, it makes saving energy with a dynamic approach more challenging.

Regarding the data preprocessing, I apply an initial data augmentation of the training set to oversample the minority classes, as a way to handle the class imbalance.

Concerning the models, I used a grid search to train all RFs and GBTs with the following combination of hyperparameters: depth in the range 1,15, input/leaves quantization to 8/16/32 bits, and a number of estimators in 1,40. This leads to a total of 5400 architectures tested for each dataset and each model type. Since for Ninapro DB1 the training is performed separately for each subject, the grid search has been repeated independently 27 times. For this dataset, for the sake of space, I report the graphs obtained on the first two subjects (S1 and S2), while in the tables I report the average results over all subjects.

For Backblaze, I use as a reference the model introduced in [68]. Specifically, I fix the depth to 38 and explore all ensembles with a number of estimators in 1,30.

After performing this hyperparameter exploration, the models not fitting in the 512 kB L2 memory of GAP8 have been excluded and the most accurate one on the validation set has been selected as the starting point for the dynamic model.

Concerning the metrics, I use the number of visited nodes on average per inference (#VisitedNodes) to model the time complexity of the ensembles in a hardware-independent way. For the hardware-dependent results, I obtain the cycles from GVSoC, stimulating a deployment on GAP8 with a cluster running at 100 MHz.

4.2.4 Hardware-Independent Results

In this section, I report the results obtained from a hardware-agnostic perspective, that is, reporting the inference time complexity of each ensemble with the #VisitedNodes metric. Memory results instead refer to the space required to store the ensembles, not changing depending on the target platform.

In particular, in the first part, I report a comparison between models with different quantization precisions for inputs and leaves. Then, I compare static GBTs and RFs, both in terms of memory footprint vs accuracy and time complexity vs accuracy, exploring if one model is more suitable than the other for edge ML scenarios. Finally, I introduce the results obtained with a dynamic inference, concluding with a set of experiments to determine if an optimal execution order of the weak learners can be found.

Ensemble Quantization

Figure 4.7 shows the memory vs accuracy Pareto fronts extracted from the validation set for each combination of quantization precisions for inputs/alphas (B_{input}) and

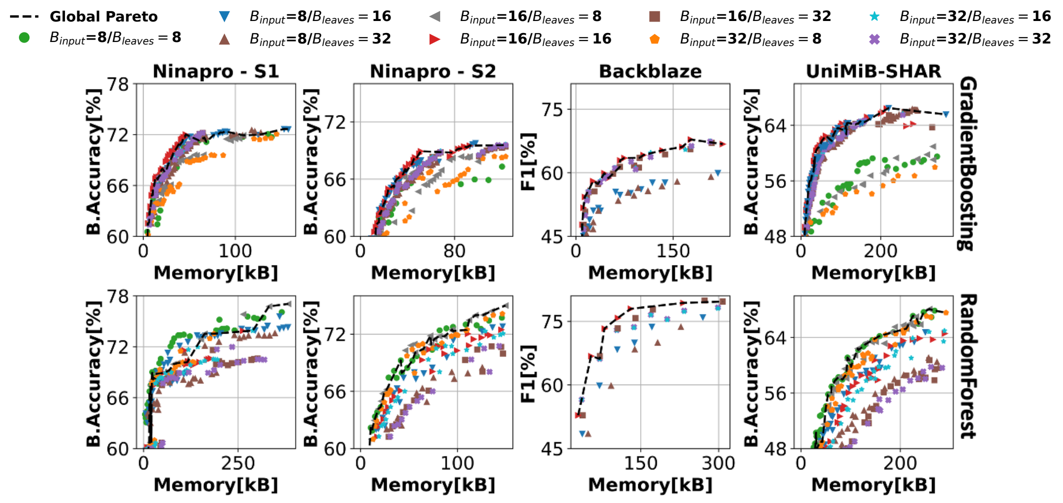


Fig. 4.7 Pareto fronts for ensembles using different quantization precisions for input/alphas (B_{input}) and leaves values (B_{leaves}). Each point denotes an ensemble with different hyperparameters.

outputs (B_{leaves}) and scored on the test set. Concerning RFs, due to the narrow ranges of the probabilities stored in the leaves, 8-bit quantization is often enough, both for inputs and outputs. Notably, the only exception is Backblaze, where 8-bit quantization causes sharp drops in accuracy.

Regarding GBTs, the 8-bit Pareto front underperforms instead, causing sharp drops in accuracy. This is caused by the wider range of the GBT outputs, requiring more bits than RFs. On Backblaze, the 8-bit Pareto front for GBTs reduces the F1 score significantly and has been omitted from the Figure for the sake of clarity.

For both ensembles, 32-bit models are rarely on the Pareto fronts. The reasons for this are two-fold and orthogonal to each other. The first and most straightforward is the sharp increase in memory of these models, rapidly saturating the device's L2 memory. Only ensembles with a lower number of shallower trees can be deployed with this precision, enabling the use of far less predictive models. The second reason is the regularization effect introduced by the quantization, similar to the one already observed in DNNs [28].

GBT vs RF comparison

Figure 4.8 shows the global Pareto front in terms of accuracy versus memory extracted from the validation set and scored on the test set. On all datasets, GBTs

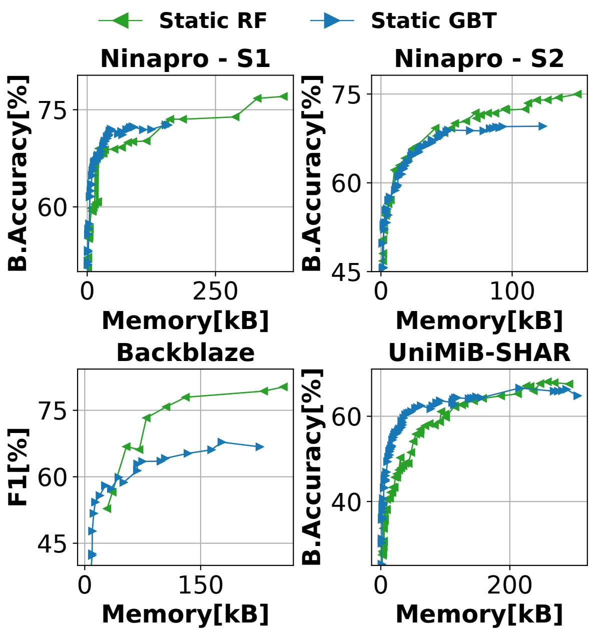


Fig. 4.8 Static Pareto fronts in terms of memory vs prediction scores of GBTs and RFs. Each point shows an ensemble with different hyperparameters.

outperform RFs for small models (less than 40-150 kB depending on the task), achieving higher accuracy at iso-memory. On the contrary, RFs become optimal for less tight memory constraints and additionally, reach the top accuracy within the constraints of the L2 memory of GAP8 for all datasets. Specifically, on Ninapro DB1, for the first subject (S1), RFs outperform GBTs by more than 4% balanced accuracy (77.05% vs 72.64%). The pattern is similar for S2, where RFs score up to 74.98% balanced accuracy, while GBTs saturate at 69.56%. On Backblaze, the difference is even sharper, with GBTs achieving up to 66% F1 score and RFs up to 79%. Finally, on UniMiB-SHAR, RFs achieve 67% balanced accuracy, while GBTs 65%. Nonetheless, for this dataset, GBTs perform significantly better for small models, with a memory footprint lower by 4x at 52% accuracy. The reason for this trend lies in the structure of the classifiers, as GBTs do not need an external LEAVES matrix, since the scalar values are stored in NODES. This is more evident for smaller models, where the size of these two data structures is similar.

Figure 4.9 shows the global Pareto front in terms of accuracy versus time complexity, reported as the number of nodes visited per inference averaged over all input samples. This metric is necessary to compare fairly trees with different depths. Notably, the Pareto models shown are different from the ones of Figure 4.8. Con-

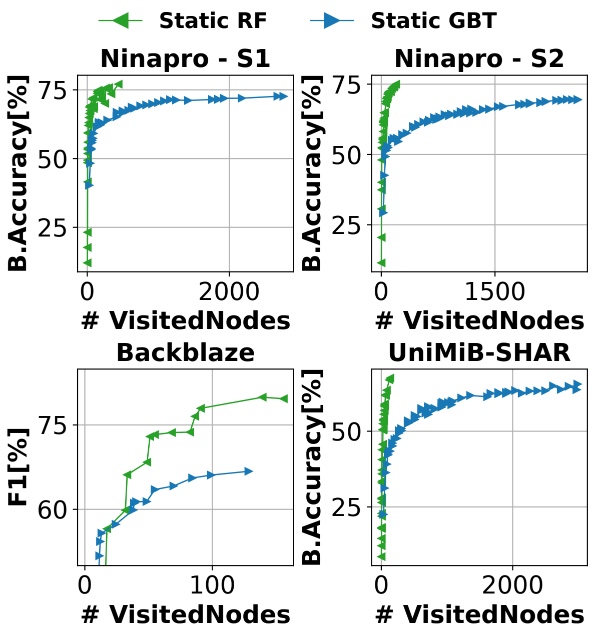


Fig. 4.9 Static Pareto fronts in terms of time complexity vs prediction scores of GBTs and RFs. Each point shows an ensemble with different hyperparameters.

cerning the time complexity, RFs are superior, with GBTs being 2x to 45x slower at iso-accuracy for Ninapro and from 4x to 30x for UniMiB-SHAR. This is again caused by the structure of the classifiers. Each GBT estimator is composed of M regression trees (i.e. one per class), while RFs feature a single classification tree per estimator. This is different for Backblaze, as for binary classification tasks the number of trees per estimator is only one for GBTs, hence closely resembling the RFs algorithm in terms of complexity.

In conclusion, while RFs outperform GBTs in terms of time complexity, they are less accurate for lower memory budgets. For this reason, I explore dynamic inference approaches on both models.

Dynamic Inference

In this section, I show the results obtained with the dynamic inference policies detailed in the previous sections: Agg. Max and Agg. Score-Margin. As the goal of dynamic inference is minimizing the inference time complexity, I report the results in terms of accuracy versus number of visited nodes. I compare the aforementioned policies with a static inference and with other state-of-the-art early-stopping approaches such as Max, Score Margin and QWYC. The latter is reported both with and without the tree re-ordering mechanism proposed by the authors (respectively *QWYC order* and *QWYC unordered*).

Dataset	**Depth**	**#VisitedNodes**	**#Estimators**	B_{input}	B_{leaves}	**Score [%]**	**Memory [kB]**
GBTs							
Ninapro	5.9 [±0.7]	3060 [±387]	37[±3.5]	17[±10]	20[±9]	75[±6]	199.5[±65]
Backblaze	15	128.53	9	16	16	66	226
UniMiB-SHAR	8	2987	22	8	16	65	363
RFs							
Ninapro	13.8 [±1.28]	348 [±81]	31.8 [±7]	16 [±9.5]	12.4 [±4.5]	77 [±6]	335.83 [±97]
Backblaze	38	155	9	16	32	79	308
UniMiB-SHAR	15	136	10	32	8	67	292

Table 4.1 Hardware agnostic deployment results.

Table 4.1 reports the static models used to derive the dynamic ensembles, corresponding to the most accurate points of Figure 4.10, denoted as*seed models*. For each model, I report the depth of the ensemble (Depth), the number of estimators (#Estimators), the average number of visited nodes per inference over the test set (#VisitedNodes), the score on the test set (balanced accuracy or F1), the bit-width of the inputs (B_{input}) and outputs (B_{leaves}) and the memory footprint of the model (Memory). Concerning Ninapro, I report the average of each field over all 27 subjects, with its standard deviation in square brackets. Noteworthy, this table shows the influence of the hyperparameters on the final score, as each dataset presents different depths and numbers of estimators, highlighting the importance of the initial grid search.

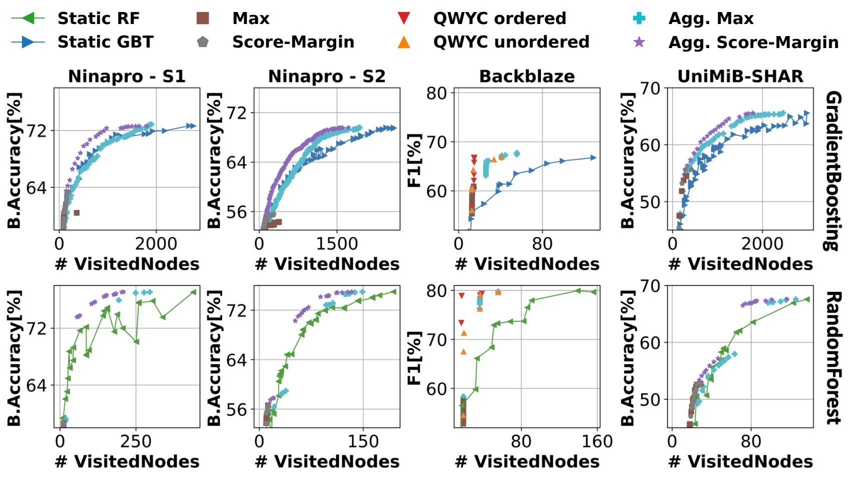


Fig. 4.10 Pareto fronts for GBTs and RFs comparing a static inference and a dynamic one.

Figure 4.10 reports on the first row the results obtained using a static GBT (blue curve) or the different adaptive policies. The second row reports the same comparison for RFs, with the green curve representing the static RFs. In both cases, I report the adaptive Pareto curves obtained by applying the respective policies to the seed models, i.e. changing the early-stopping threshold th. On the contrary, points belonging to the static Pareto fronts represent different architectures, obtained with a distinct set of hyper-parameters. The Max and Score Margin curves show the results of the two adaptive policies applied in their original form, that is, only using the predictions of the last executed tree. The Agg.Max (S^t) and Agg.Score-Margin (SM^t) denote instead the results obtained with the policies I introduced in the previous sections. Finally, the QWYC approach is only applied to the Backblaze dataset as it is the only binary task in this benchmark. Notably, in this experiment, the batching mechanism is not considered (B=1) and the early stopping policy is evaluated after each estimator.

Concerning the first subject (S1) of Ninapro, dynamic RFs using SM^t obtain a reduction of up to 83% visited nodes at 73% balanced accuracy. On the other hand, existing policies such as the Score Margin (SM) achieve only 59% maximum accuracy. For GBTs, we reduce the number of visited nodes by 54% at 71% balanced accuracy, with again the state-of-the-art policies causing an accuracy drop of 9%.

The same pattern is repeated for S2. Dynamic RFs using SM^t achieve a node reduction of up to 45% at 74% accuracy, with SM instead yielding an accuracy of only 56%. Concerning GBTs, I achieve a visited node reduction of up to 51% at 65% balanced accuracy, while SM causes a reduction of 14%. When considering all subjects in the Ninapro, the maximum reduction of visited nodes achieved at iso-score is 58.8[$\pm$1.2]% for RFs and 58.5 [$\pm$9]% with GBTs.

Concerning Backblaze, the top gain is 69% for RFs and 88% for GBTs, respectively obtained at 73% and 66% F1 score. In this case, QWYC achieves the best results due to its double threshold mechanism, increasing its accuracy when few DTs are employed. As for Ninapro, the previous state-of-the-art policy (Max) leads to large score drops, respectively 22% for RFs and 6% for GBTs.

Finally, regarding UniMib-SHAR, the number of visited nodes is reduced by up to 41% by RFs and 58% by GBTs with SM^t, respectively at 66% and 63% balanced accuracy. Again, SM causes sharp score drops, achieving a maximum balanced accuracy of 52% with RFs and 56% with GBTs.

Noteworthy, aside from the aforementioned reductions in time complexity, the strength of dynamic inference lies also in its flexibility, as deploying the entire curves reported in Figure 4.10 requires only changing the threshold value th. On the other hand, deploying the corresponding static curve requires storing in memory several ensembles at the same time, an unfeasible option due to memory constraints.

Dataset	**Model**	**#VisitedNodes**	**#Estimators**	**Policy**
		GBTs		
Ninapro	S.	3060[387]	37[3.5]	
	A.-Iso	1805[315]	22 [3.73]	Agg.SM
	A.-1%	1096[191]	13.4[2.6]	Agg.SM
Backblaze	S.	128	9	
	A.-Iso	14.89	1.01	QWYC o.
	A.-1%	14.75	1.003	QWYC o.
UniMiB	S.	2987	22	
	A.-Iso	1766	13.02	Agg.SM
	A.-1%	1286	9.48	Agg.SM
		RFs		
Ninapro	S.	348[81]	31.8[7]	
	A.-Iso	175[49]	15[3.9]	Agg.SM
	A.-1%	104[30]	9[2]	Agg.SM
Backblaze	S.	156	9	
	A.-Iso	55	3.03	Agg.Max
	A.-1%	17	1.0005	QWYC o.
UniMiB	S.	136	10	
	A.-Iso	116	8.46	Agg.SM
	A.-1%	73	5.37	Agg.SM

Table 4.2 Hardware agnostic deployment results. Abbreviations: Seed model (S), Iso accuracy with seed (A.Iso), 1% accuracy drop w.r.t to seed (A.-1%), QWYC ordered (QWYC o.)

As an additional comparison, Table 4.2 reports the statistics of the seed model (denoted as S) used as a starting point for the dynamic ensemble and the best dynamic configurations built on top of them. Specifically, I report the dynamic configuration able to achieve the largest reduction in terms of visited nodes at iso-score with the seed (A-Iso) and allowing a 1% score drop (A-1%). As all trees executed by all three models have identical depth (dynamic ones run a subset of the seed model), I report as well the average number of estimators executed per inference over the whole test set and the adaptive policy used.

At iso-accuracy, I achieve a reduction of visited nodes of up to 88% with Backblaze, 49% on Ninapro, and 41% with UniMiB-SHAR. The savings increase significantly for the last two datasets if we allow a 1% score drop, achieving a reduction of up to 70% for Ninapro and 57% for UniMiB-SHAR. On Backblaze, the A-1% configuration achieves a reduction of 89% in terms of visited nodes.

Overall, the Agg. Score Margin (Agg.SM) is the top-performing policy when considering multi-class classification tasks. On the binary classification task featured by Backblaze, the top performing policy is 3 times out of 4, the QWYC with ordering. The reason is that the former, thanks to the double threshold mechanism implemented, manages to trigger more early stops when an easy class is predicted.

Finally, both Max and Score Margin cause significant score drops, always failing to achieve iso-score with the seed model. Intuitively, this is due to the fact all weak learners are almost equally predictive in an ensemble, thus using only one to trigger the early stop leads often to wrong decisions. As a consequence, both Max and Score Margin are sub-optimal in this scenario and are not considered for deployment.

Tree Ordering

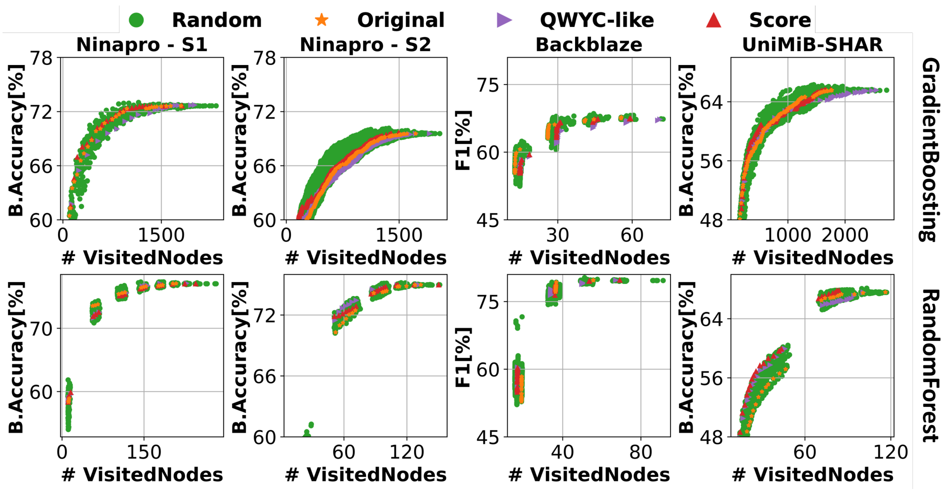


Fig. 4.11 Effects of different tree orders on the dynamic inference.

Intuitively, finding the optimal order of execution of DTs of a dynamic ensemble plays a significant impact on the obtained savings and final accuracy. For instance, running first the most accurate weak learner could seem the best way to force an even quicker activation of the early stopping mechanism. Nonetheless, I found that this is generally not the case. For instance, considering the QWYC algorithm in Figure 4.14, the performance of the ordered algorithm underperforms w.r.t. the unordered one. This indicates that sorting the DTs based on the validation score does not lead to optimal inferences on the test set. To further investigate this issue, I performed a benchmark by sorting the trees in the seed static ensembles according to different metrics, reporting the results in Figure 4.11. Specifically, I benchmark the same dynamic ensemble using the Agg.Score Margin, sorting its weak learners according to i) 50 randomly spawned orders (Random), ii) two greedy ordering algorithms (Score and QWYC-like), or iii) the original training order (Original). The Score ordering sorts the trees in descending order of score on the validation set. The QWYC-like sorting algorithm takes inspiration from [43], ordering the estimators so that the number of visited nodes needed to reach iso-accuracy with the seed model is minimized. Finally, different points have been obtained by changing the early-stopping threshold th.

Notably, no smart ordering algorithm outperforms the random one, with the original training order being the middle of multiple other random curves. However, selecting the top-performing random curves is unfeasible in practice, as there is no correlation between the best ordering on the validation set and the one on the test set.

To conclude, using the validation set to determine an ordering on the test set is not an effective approach to optimize the efficiency of the dynamic inference. Therefore, I use the training order for the other experiments presented in this dissertation.

4.2.5 Deployment Results

GBT vs RF comparison

Figure 4.12 shows a comparison between static RFs and GBTs deployed on GAP8 with C=8. As opposed to the results obtained in terms of the number of visited nodes, GBTs gain the edge at lower scores in terms of cycles. This is caused by the accumulation step of the two models, performed in both cases in a critical section, i.e. sequentially. RFs aggregate vectors of M values, while GBTs only accumulate a single value, freeing faster the lock on the output vector. Nonetheless, as already shown in Figure 4.9, RFs rapidly become more accurate, sharply increasing in terms of score, while GBTs' score saturate.

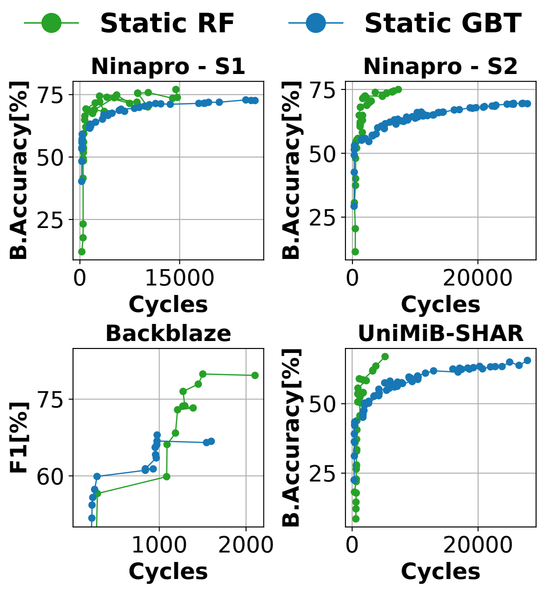


Fig. 4.12 Score versus average clock cycles per input on GAP8.

Dynamic Inference

In this section, I report the deployment results of the dynamic policies already shown in Figure 4.10. The number of visited nodes is replaced with the average cycles per inference on the test set as it provides a proxy for latency and energy consumption. I report the results obtained with a batch B of 1,2,4 and 8, always setting $B = C$ both for static and dynamic models.

Figures 4.14 and 4.13 report the dynamic inference results. For batch sizes up to 4, dynamic solutions are frequently Pareto optimal. The reason is the lower parallelization, making the overhead of the adaptive policy (computed in a critical section) lower w.r.t to the static ensemble, yielding consistent savings in terms of cycles. Considering $B = 4$, on Ninapro, the largest savings are obtained by an RF at 74% balanced accuracy both for S1 and S2. Specifically, using SM^t, for S1 I achieve a reduction of 71.8% cycles, while for S2 a 27.1% reduction. Concerning Backblaze, the largest saving is obtained with a GBT at 66.8% F1 score, achieving a reduction of 36.6% of average cycles per inference. Finally, on UniMib-SHAR, I achieved a reduction of up to 47.7% cycles compared to a static ensemble at iso-score (63.4%).

When considering B=8, the dynamic inference overhead becomes significant w.r.t to the highly parallel execution of the static ensemble. As a consequence, only a few dynamic points are Pareto optimal. This leads to the conclusion that as the number

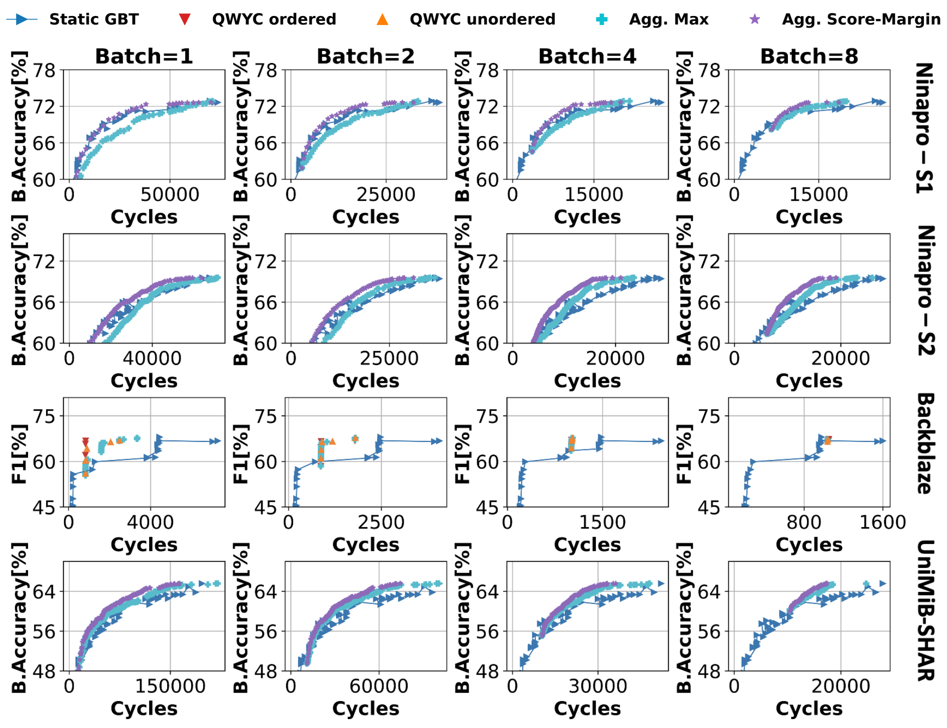


Fig. 4.13 Dynamic and static Pareto fronts in terms of accuracy vs cycles for Gradient Boosting Trees.

of cores increases, the effectiveness of a dynamic inference approach is reduced. Nonetheless, when compared to the most accurate static models benchmarked, a dynamic inference approach can yield a large reduction of cycles.

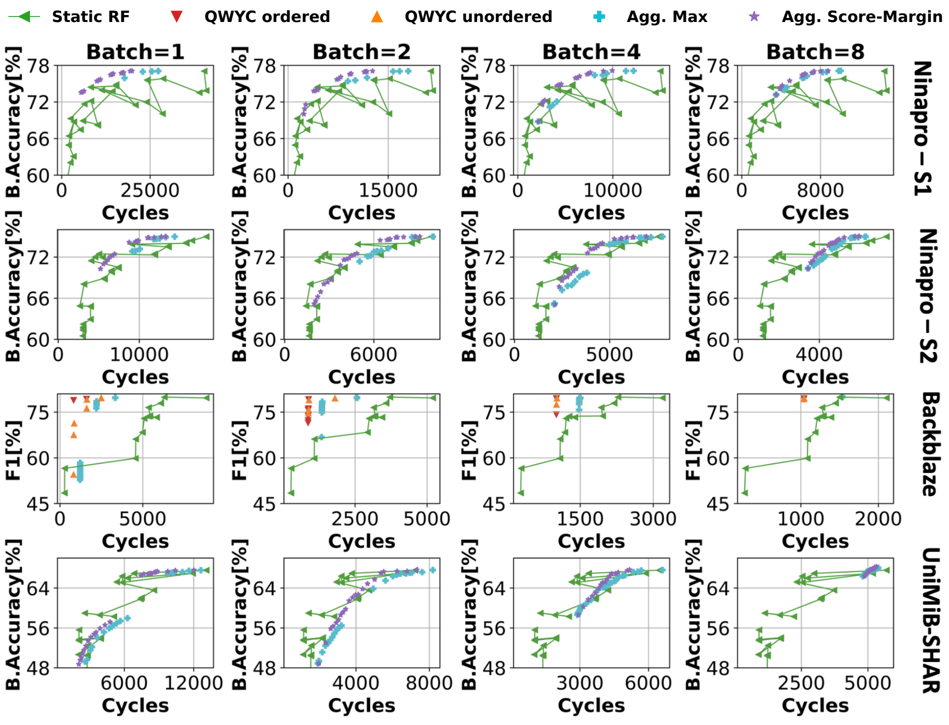


Fig. 4.14 Dynamic and static Pareto fronts in terms of accuracy vs cycles for Random Forests.

		1 CORE				8 CORES						
Dataset	Model	Trees C.	Acc. C.	Policy C.	Total C.	Trees C.	Acc. C.	Policy C.	Total C.	E.	Lat.	Policy
						GBTs						
	S.	169005	9324	n.a.	178329	21330 (7.92×)	9324	n.a.	30654 (5.81×)	15.63	30.65	
Ninapro	A.-Iso	86878	7700	1496	94578	11950 (7.27×)	7700	204 (7.33×)	19854 (4.76×)	10.13	19.85	Agg. SM
	A.-1%	52233	4690	911	56923	8381 (6.23×)	4690	136 (6.69×)	13207 (4.31×)	6.73	13.21	Agg. SM
	S.	7105	162	n.a.	7267	1436 (4.95×)	162	n.a.	1598 (4.54×)	0.81	1.60	
Backblaze	A.-Iso	4624	25	50	4699	985 (4.69×)	25	25 (2.0×)	1035 (4.54×)	0.53	1.04	Agg. Max
	A.-1%	4624	25	50	4699	985 (4.69×)	25	25 (2.0×)	1035 (4.54×)	0.53	1.04	Agg. Max
	S.	193868	2992	n.a.	196860	24844 (7.80×)	2992	n.a.	27836 (7.07×)	14.20	27.84	
UniMiB	A.-Iso	75324	5533	1250	82107	11573 (6.51×)	5533	192 (6.51×)	17298 (4.74×)	8.82	17.30	Agg. SM
	A.-1%	52106	4029	910	57045	10685 (4.88×)	4029	192 (4.73×)	14906 (3.83×)	7.60	14.91	Agg. SM
						RFs						
	S.	22055	6720	n.a.	28775	2892 (7.62×)	6720	n.a.	9612 (2.99×)	4.90	9.61	
Ninapro	A.-Iso	12580	3150	1020	16750	2316 (5.43×)	3150	136 (7.5×)	5602 (2.99×)	2.86	5.60	Agg. SM
	A.-1%	9007	1890	612	11509	1823 (4.94×)	1890	136 (4.5×)	3849 (2.99×)	1.96	3.85	Agg. SM
	S.	8676	162	n.a.	8838	1941 (4.47×)	162	n.a.	2103 (4.20×)	1.07	2.10	
Backblaze	A.-Iso	6289	75	75	6439	1433 (4.39×)	75	25 (3.0×)	1533 (4.20×)	0.78	1.53	Agg. Max
	A.-1%	4300	25	35	4360	978 (4.40×)	25	35 (1.0×)	1038 (4.20×)	0.53	1.04	QWYC u.
	S.	9400	3700	n.a.	13100	2046 (4.59×)	3700	n.a.	5746 (2.28×)	2.93	5.75	
UniMiB	A.-Iso	7851	3130	643	11624	1794 (4.38×)	3130	152 (4.23×)	5076 (2.29×)	2.59	5.08	Agg. SM
	A.-1%	8960	1986	188	11134	2841 (3.15×)	1986	35 (5.37×)	4862 (2.29×)	2.48	4.86	Agg. Max

Table 4.3 Dynamic Inference deployment results at 1 core and 8 cores. Abbreviations: *C.* cycles, *E.* energy, *Lat.* Latency

Table 4.3 reports a detailed comparison of cycles, energy, and latency results obtained by the static seed models and the most efficient dynamic models at iso-score or with a score drop of up to 1% w.r.t. to the seed. The statistics have been reported both for the case $B = C = 1$ and $B = C = 8$, to provide detailed insights on the effects of the parallelization on inference. To do so, the profiling has been performed on the distinct parts composing the ensemble inference, namely the trees' execution cycles (Trees C.), the probability accumulation step (Acc. C.) and the policy evaluation (Policy C.), reporting their average value over the test set. Moreover, latency (Lat.) and energy (E.) results have been provided for the $C = 8$ case.

When comparing the sequential execution of the trees (Trees C.) to the parallel one on 8 cores, the speedup observed ranges from 3.15x to 7.92x. This sub-optimal speed-up is due mainly to the leftover trees executed in the last batch and the imbalance between trees. For instance, the seed model used for UniMiB-SHAR features 10 trees, with the first 8 being run in parallel first and then executing the last two. The maximum speed-up is then given by $\frac{10}{2} = 5x$.

The speedup shown for the policy computation (Policy C.) is instead due to the policy being triggered $C\times$ fewer times, as we set $B = C$. In conclusion, when considering $C = 8$, a dynamic inference approach can lead to reductions of latency and energy of up to 41.7% for Ninapro, 35.2% for Backblaze and 37.9% for UniMiB-SHAR at iso-score. If a 1% drop in score is allowed, then the gains rise respectively to 60%, 50.6%, and 46.5%.

Chapter 5

Optimization Methods for Deep Learning

In this chapter, I focus on DNNs. In particular, in the first part of the chapter, I provide an overview of the current landscape of DL-based solutions for Human Activity Recognition (HAR). Then, I provide an overview of dynamic inference methods, focusing mainly on iterative and hierarchical approaches.

In the second part of this chapter, I benchmark the effect of sub-byte quantization and mixed precision on DNNs for Human Activity Recognition. Finally, I describe three dynamic inference approaches to reduce the energy footprint of DNNs.

5.1 Related Works

5.1.1 Human Activity Recognition

ML approaches have become increasingly popular for Human Activity Recognition tasks, as they lead to superior results w.r.t. classical algorithms based on signal thresholding or filtering.

Commonly, these works exploit shallow ML algorithms, such as RFs, k-Nearest Neighbors (k-NN), and Support Vector Machines (SVMs). For instance, the authors of [88] leverage a low-pass filter pre-processing, followed by a feature extraction phase and one out of six ML classifiers. In particular, they benchmark logistic model trees (LMT), logit boost (LB), logistic regression (LR), SVM, RF, and a shallow Artificial Neural Network (ANN) on the target dataset.

In [89], the authors benchmark an SVM on a multi-class HAR dataset collected with smartphones. As a further optimization, they quantize the model to reduce the memory footprint of the classifier, introducing a minimal accuracy drop.

Even more recently, DL approaches for HAR have been able to achieve state-of-the-art accuracy on several datasets [90]. The authors of [91] propose a hierarchical model based on a CNN and a Bidirectional LSTM (BiLSTM), benchmarking their solution on two public datasets, UCI HAPT and MobiAct [92]. Moreover, they compare other classifiers such as CNNs, SVMs, k-NNs, CNNs-LSTMs. On a reduced version of UCI HAPT (postural transitions are clustered into 2 groups, yielding 8 classes instead of 12) they achieve 97.98% accuracy and 96.16% on MobiAct.

The authors of [93] introduce custom convolutional filters named LEGO, leading to significant reductions in terms of memory and minimal accuracy drops. With benchmarks on 5 datasets, they achieve F1 scores up to 97.51% for WISDM and 74.46% for UniMiB-SHAR.

In [94], the authors propose a CNN-based solution to recognize 9 different activities on a self-collected dataset. On the public dataset UCI HAR, their solution achieves up to 92.5% accuracy. The authors of [95] propose a partially binarized CNN that achieves 93.67% accuracy on the PAMAP2 dataset. Their architecture is then deployed on an FPGA.

Noteworthy, while DL solutions are often more accurate than shallow ML algorithms, they are also more computationally and memory-demanding. For instance, the authors of [96], using lightweight RNNs, consider as a deployment target a Raspberry Pi3 equipped with 1 GB of RAM and several Watts of active power consumption. The deployment platforms considered in this dissertation are instead MCUs, equipped with less than 1 MB of L1 memory and featuring a power consumption of three orders of magnitude lower. The models proposed in [91, 93] feature a size of at least 1.2 MB, too large for MCU deployment. Even the model used in [95], proposing an aggressively optimized CNN, is deployed on FPGA and not on a general-purpose MCU.

5.1.2 Dynamic Inference

The main limitation of the Big/Little schema introduced in Chapter 2 lies in the memory overhead that it introduces, as it requires storing two models instead of a single one. This overhead may become a significant constraint on MCUs, where the memory may be barely enough to deploy a single DNN.

The authors of [40] propose to build the *little* model from the *big* one, activating only a portion of the layers' channels/neurons for easy inputs. Only complex inputs instead are classified with the original-width model. As a consequence, the weights of the two models are shared, removing entirely the memory overhead. However, this approach requires a custom training algorithm, while even pre-trained networks could be used for the original big/little approach. Furthermore, the weight-sharing mechanism may lead to slight losses in terms of accuracy.

In [97], the authors propose using multiple versions of the same DNN obtained through quantization at different precisions. In practice, first, the network is quantized to the largest bit-width that satisfies a memory vs accuracy constraint. Then, lower precision versions are obtained by truncating the weights obtained from the largest configuration. Notably, this approach requires no additional weights to be stored, while also requiring no retraining.

Another dynamic inference mechanism is hierarchical inference. A task is split into multiple sub-tasks, with the easiest being the most common. The sub-tasks are then executed in order of increasing complexity, implementing an early stop to save energy. The strength of this approach lies in its flexibility, as complex sub-tasks can be easily offloaded to servers running computationally demanding models. On the contrary, easy tasks can be still performed locally. Nonetheless, this approach is task-dependent, as it is not always possible to find easier sub-tasks. For instance, the authors of [98] split a facial recognition task for devices such as smartphones. The sub-tasks are: i) whether the input picture contains a face, ii) whether the face belongs to the phone owner or not, and in the negative case, iii) whether the face belongs to one of the phone owner's favorite contacts. The tasks are run by increasingly complex CNNs deployed on a custom accelerator.

Noteworthy, the limitations of the dynamic inference approaches in the literature are twofold. First, they benchmark large DNNs, featuring memory and energy footprints unsuitable for embedded targets such as MCUs. Second, they focus almost

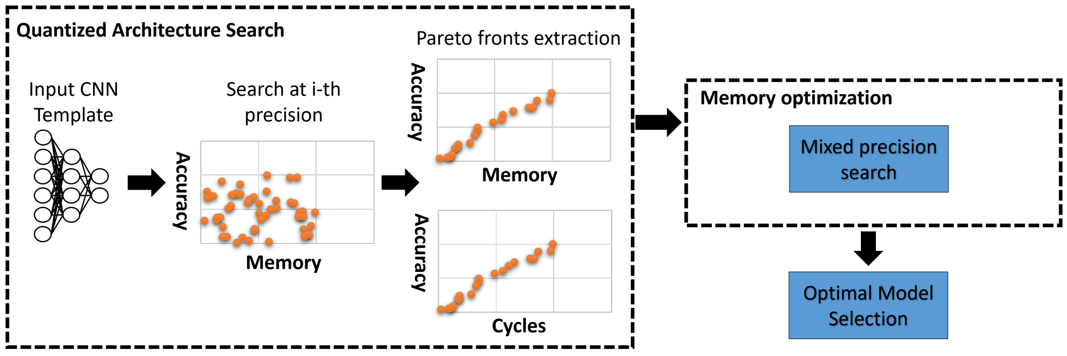


Fig. 5.1 Overview of the search flow for uniform- and mixed-precision DNNs.

exclusively on computer vision tasks, as opposed to this work where the focus is time-series, common in embedded scenarios. Therefore, two of the contributions of this work are the effects of dynamic inference i) for less computationally demanding models, often being less stable during training or less calibrated, ii) for time-series data.

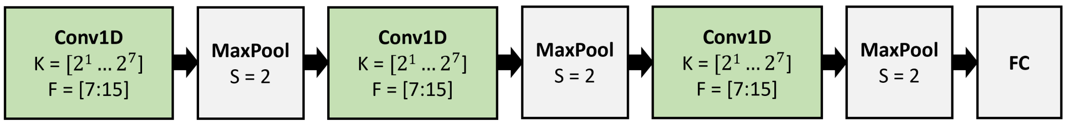
5.2 Sub-Byte DNNs for Human Activity Recognition

5.2.1 Introduction

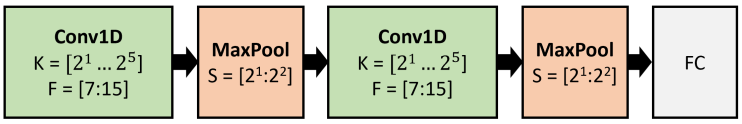
Human Activity Recognition based on IMU is an increasingly popular task for embedded ML. Nonetheless, due to the high memory requirements of DL models, most HAR systems are based on classical ML solutions. In this section, I show that it is possible to find CNNs that are both memory-efficient and accurate, becoming then a feasible option for deployment on MCUs, benchmarking the results on Pulpissimo.

In detail, I leverage a flow that starting from an extensive parameter search to find suitable architectures, explores the effects of sub-byte quantization and mixed precision. The intuition is that aggressive forms of quantization allow finding better trade-offs in terms of accuracy vs memory than the popular yet limited 8-bit precision.

The proposed flow is shown in Figure 5.1. The first step is named *Quantized Architecture Search* and consists of an exhaustive grid search of the hyperparameters of a template CNN. This search is repeated for all the precisions considered in this work, namely 1,2,4, and 8 bits.



(a) UCI HAPT, WISDM, UniMiB-SHAR



(b) WALK

Fig. 5.2 CNN templates used as starting point for the grid search.

The second step, named *Memory Optimization*, leverages a modified version of a differential Neural Architecture Search (dNAS) for fast exploration of all combinations of precisions of weights and activations, finding an optimal configuration for each layer in the network.

Finally, I show an extensive set of results on four state-of-the-art HAR datasets, benchmarking the effects of different quantization precisions on this task.

Note that while previous works have explored mixed precision and sub-byte quantization, they are limited to image data (2D CNNs) and feature large networks. On the other hand, in this dissertation, I explore the advantages of sub-byte quantization and mixed precision on a new task and type of network (1D CNNs).

The work presented in this Section has been published in [99, 42].

5.2.2 Quantized Architecture Search

Architecture Search

Figure 5.2 shows the CNN templates used as a starting point for architecture exploration, depicting layers in green or red if their hyperparameters are changed during the grid search. The structure of the networks has been selected empirically, focusing on models that achieve acceptable accuracy with a low number of parameters. In particular, I took inspiration from classical CNN models, such as LeNet [100], that usually feature one or more convolutional blocks terminating in an FC layer, converting the networks into their 1D counterparts for time-series processing. Note that in Figure 5.2, both ReLU activation functions and batch normalization (BN) layers always follow Conv layers, but are not shown for the sake of simplicity.

Concerning the hyper-parameters search space, I keep it identical for three of the datasets I target in this work: UCI HAPT, UniMIB-SHAR, and WISDM. In particular, for the datasets above, I explore all power-of-two values between 2 and 128 for the output channels (K) and either 7 or 15 for the kernel size (F) for Conv1D layers. MaxPool layers are kept unchanged, using a pooling size and stride of 2.

On the WALK dataset, as the binary task featured is simpler, I benchmark a smaller template CNN, featuring only two Conv layers. The third Conv layer led to no noticeable improvement in terms of accuracy and, as a consequence, it was removed. For the same reason, the Conv output channels explored are limited to powers of two up to 32. Nonetheless, to avoid shrinking excessively the search space, I explore for Pool layers a stride/window size (the two values are always kept identical) of 2 or 4 or removing the layer entirely.

Noteworthy, I selected a grid-search algorithm to explore the search space as the training duration of each network and its memory footprint is almost negligible when performing the computations on high-end hardware. Moreover, such an approach guarantees finding all Pareto-optimal networks in the search space. In case of longer training time, a possible alternative to a grid-search algorithm would be applying a more refined NAS tool [101–103].

To obtain the best possible accuracy in each dataset, the interaction of the bit-width on the training and network hyper-parameters has to be taken into account. Intuitively, aggressive quantizations, such as the one performed in Binary Neural Networks (BNNs), may lead to significant drops in accuracy [37], that can be recovered only using a larger number of channels/neurons per layer. As a consequence, a Pareto-optimal architecture at 8-bit is generally sub-optimal when binarized, and this may cause specific bit-width to underperform. Following this reasoning, I repeat the grid search for each bit-width, always performing a *Quantization-Aware Training* [34]. In fact, post-training quantization generally leads to large drops in accuracy, especially when quantizing to sub-byte precision. The quantization algorithm used in this work is PArametrized Clipping acTivation (PACT) [104], yielding accurate networks even at low precision. Moreover, PACT quantization is compatible with the DNNs' deployment toolchain for the target hardware.

Pareto Front Extraction

After performing the architecture search, I extract the Pareto-optimal architectures for each precision. In particular, I extract two fronts, the first in terms of memory versus accuracy and the second in terms of cycles versus accuracy. Noteworthy, while for memory occupation a reliable metric is the number of parameters of the CNN, obtaining an accurate estimate of the inference latency or energy cost is complex. Relying on metrics such as the number of Multiply-and-Accumulate (MAC) operations in the network would lead to inaccurate results, favoring sub-byte quantization as no overhead for packing/unpacking data, needed on general-purpose MCUs, is modeled. Therefore through a set of C-code templates, I converted and compiled automatically all the networks found with the architecture search, using as reference the framework and the C-language DL primitives for the target MCUs introduced in [105, 106]. Noteworthy, BNNs leverage custom primitives that I designed specifically for the target MCU [72], and are detailed in the next section.

UltraCompact BNNs

Existing BNN libraries, commonly designed for CV tasks and 2D data, pad the number of input/output channels (C/K) or neurons to multiples of 32 [37, 107, 108]. This design choice stems from the fact that storing all the elements required to compute an output element in exactly one or more 32-bit words, simplifies the implementation of the binarized DL primitives (kernels), as there are no leftovers. Moreover, common DL architectures for CV have a high number of channels, making the padding overhead negligible w.r.t. the original computations of the network.

On the other hand, HAR networks are far smaller, possibly needing 2-4 channels per convolution. Such a large amount of padding would then be sub-optimal, making, for instance, it pointless to explore all K/C$< 2^5$ in the proposed hyperparameter search, as they would be all padded to 32. Therefore, I introduced a set of DL primitives for Pulpissimo that avoid the padding entirely, taking inspiration from the one proposed in [106].

Figure 5.3 shows an overview of the convolution kernel implementation, while Algorithm 6 shows its pseudo-code. Data is stored in a time-major order, with all channels belonging to a single timestep being contiguous in memory. This layout allows accessing all elements required to compute an output element of the

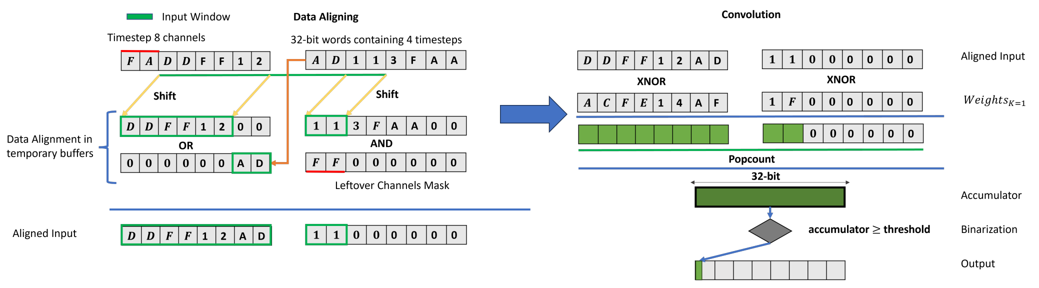


Fig. 5.3 Ultra-Compact binarized convolution with F=5 and K=C=8. The left part shows the data alignment step, moving the input data at the beginning of one or more 32-bit words. The right part shows the convolution computation for the first output channel.

Algorithm 6 Binarized convolution kernel

```
for t in range(0,T-1):
    I_a = AlignData(I,t)
    for k in range(0,K-1):
        W_a = AlignData(W,t)
        O=Popcount(XNOR(W_a,I_a))
```

convolution by loading consecutive words in memory. For instance, for a convolution with F=5 and C=8, shown in Figure 5.3, all input elements required to compute an output timestep are stored contiguously in 40 bits and can be loaded in two 32-bit memory registers. The loop order shown in Algorithm 6 has a significant impact, as it determines the data reuse pattern. I found that the proposed one led to the best performance on the target HW, computing output channels tied to the same output timestep before moving to the following.

The main difference w.r.t to multiples of 32-channel libraries lies in the fact that the convolution input data (both inputs and weights) may occupy a non-integer number of words. This is depicted in Figure 5.3, where the input window data occupies 40 bits, i.e. 1 and 1/4 words for 32-bit processors. Note that padding both weights and inputs would lead to a computational overhead of more than 33%, an unacceptable option when energy efficiency and latency are important. It becomes then necessary to handle the data alignment and the leftover issues, without lowering excessively the parallelization benefits granted by the binarization.

The left part of Figure 5.3 shows the operations performed to align the input data buffer for a convolution with F=5 and K=C=8. In particular, the Figure refers to the moment when the layer has to start computing the second timestep ($t+1$), shifting the input by C (i.e., 8 bits in this example). As shown, the input data is stored in two consecutive words and as the weights always start from the most significant bits of a word, it is misaligned. Therefore, through a series of shift and masking operations, I move the required data so that it starts at the beginning of the first word. Note that I also mask out the last word, to avoid computing the XNOR operator on bits not belonging to the considered timesteps. To avoid any destructive operations on the input/weight tensors, I use an additional intermediate buffer.

The right part of Figure 5.3 shows the convolution on the aligned data. This part is kept identical to other implementations [37], computing the convolution through an XNOR operation followed by popcount. Then, the accumulated value is binarized with a threshold derived from the BN layer. Noteworthy, due to the light overhead of shift and masking operations, this implementation achieves up to 44% cycle reductions w.r.t. to a padded binarized convolution when considering $C = 2$ and $K = 32$.

To further improve the throughput, I take inspiration from the kernels proposed in [109], manually unrolling parts of the loops of the convolution. That is, I compute multiple output timesteps and channels at each iteration of Algorithm 6. The goal is to increase the arithmetic intensity of the kernel, that is the number of XNOR/Popcount per operation per memory load. By profiling the performance of different convolutions on the target hardware, I found that the best performances are achieved with a 2x2 kernel, computing at each iteration 2 output channels of 2 consecutive timesteps. Further unrolling leads to register spilling, leading to sharp decreases in performances.

A further optimization consists of fusing the MaxPool operator with the convolutional kernel. Intuitively, a max operation can be implemented with a bitwise OR. By modifying how the convolution selects the output bit to write, this operation can be implemented directly in the kernel, reducing input/output write operations. Moreover, no custom implementation for MaxPool is necessary, reducing the code size when deploying the model.

5.2.3 Memory Optimization with Mixed Precision Quantization

Previous works [32, 33] have shown that an optimal trade-off between accuracy and memory is not always found through a network-level quantization. On the contrary, mixed-precision approaches, where a different precision is assigned to

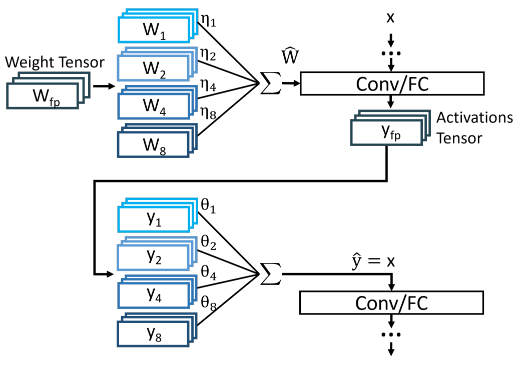


Fig. 5.4 Overview of how EdMIPS searches for the most suitable bit-width for activations and weights.

each set of weights and activations, allow greater flexibility, leading to improved results. Intuitively, a mixed-precision approach can assign greater precision to layers having a major impact on the accuracy (e.g., the first and last one in DNNs), while quantizing aggressively the others.

Following this intuition, I apply the mixed-precision approach to HAR. The key problem for mixed-precision quantization is assigning a bit-width to each tensor, as an exhaustive search is unfeasible even for small DNNs. For instance, for the template CNNs shown in Figure 5.2 featuring 3 convolutions, exploring all possible combinations of precisions for 1,2,4, and 8 bits would require performing $2^1$6 training runs. Moreover, the training runs should be repeated for each Pareto model found in the architecture exploration grid-search, as they refer to a single architecture.

As a consequence, I leverage an open-source dNAS named EdMIPS [110] that searches for the optimal bit-width of each tensor at training time, that is, while optimizing the network weights. When compared to other tools for mixed-precision search leveraging evolutionary or reinforcement learning algorithms, EdMIPS performs significantly faster, terminating the search in a time comparable to the one required for standard network training. Figure 5.4 shows an overview of how EdMIPS makes precision assignment differentiable, allowing a gradient-based optimizer to determine the optimal precision for each tensor.

In particular, the NAS introduces into the standard network parameters two sets of trainable coefficients named η and θ. During the training phase of the network, the quantization of both output activations y and weights W is simulated for all the explored precisions in the forward step.

Taking as reference the output activations, a different quantized version for each precision is created starting from the original floating point value y_{fp}. These quantized versions, i.e y_1, y_2, y_4, y_8 respectively for 1-,2-,4- and 8- bits precisions are only simulated, as the original value is still stored in floating point. However, the values are quantized according to the selected quantization algorithm, simulating the effect of the integer representation.

The θ coefficients are obtained with a SoftMax operation (summing to 1) and are used to combine the different precision activations with the following equation:

$$\hat{y} = y_1 * \theta_1 + y_2 * \theta_2 + y_4 * \theta_4 + y_8 * \theta_8 \tag{5.1}$$

Finally, $\hat{y}$ is used as output activation of the corresponding FC or Convolutional Layer. The same procedure is applied to the weights W using the η coefficients.

At training time, the loss is changed to:

$$\mathscr{L} = \mathscr{L}_{task}(W, \eta, \theta) + \lambda * \mathscr{L}_{cost}(\eta, \theta) \tag{5.2}$$

adding to the normal task loss $\mathscr{L}_{task}$ (e.g., cross-entropy for multi-class classification problems) an additional loss $\mathscr{L}_{cost}$, measuring the deployment cost of the networks, based on the current values of η and θ. As an example, the estimated cost for storing the weights of a layer can be computed as follows:

$$cost = (1 * \eta_1 + 2 * \eta_2 + 4 * \eta_4 + 8 * \eta_8) * W_{size} \tag{5.3}$$

where W_{size} denotes the number of elements in W_{fp}.

Thanks to the combined loss, both η and θ converge to the values that better balance the accuracy drop caused by quantization and the accuracy cost. Then, the best network configuration can be extracted by selecting the precisions with the highest η and θ coefficients.

Intuitively, changing the value of λ changes the importance of the accuracy w.r.t to the network cost and can be used to obtain several trade-off points, giving high flexibility while significantly reducing the search cost w.r.t to a grid search. In my work, I adapted EdMIPS to handle Conv1D networks, as it was originally designed for 2D networks. Moreover, I introduced the PACT [104] algorithm simulation, substituting the original one proposed in [110], as it was less hardware-friendly.

5.2.4 Results

Experimental Setup

All the networks have been trained using Python 3.8 and PyTorch [30] using an Adam optimizer with an LR scheduler that multiplies the LR by 0.1 when the training loss is stale for three consecutive epochs. Moreover, I added an early-stop mechanism to the training, setting the patience to 5 epochs.

The datasets used are UniMIB-SHAR, WISDM, WALK, and UCI HAPT, with the preprocessing detailed in Chapter 3. I set the LR to 0.001 for UniMiB-SHAR and WALK, and 0.01 for UCI HAPT and WISDM. The batch size is 32 for all datasets but UCI HAPT, where it is set to 128.

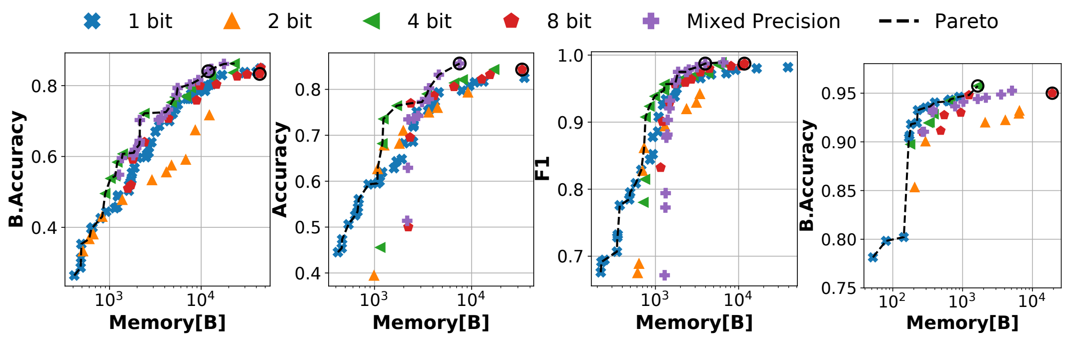
The random forests used in the state-of-the-art comparison are taken directly from the original papers and I use the same metric of those works to report the classification score of the models.

The C-language deep learning primitives used to deploy DNNs are taken and adapted from [106] concerning bit-widths 2, 4, and 8. The mixed precision models have been obtained by repeating the search with 10 different λ values, ranging from 10^{-4} to 10^{-3}. Values outside this range yielded networks either fully binarized or fully quantized at 8 bits.

The deployment target is Pulpissimo, with cycles and energy results obtained as detailed in Chapter 3.

Uniform vs Mixed Precision quantization

Memory occupation: Figure 5.5 shows the CNNs obtained with the detailed deployment flow. Specifically, for each dataset, the graph shows the Pareto fronts in terms of prediction quality (B.Accuracy, Accuracy, or F1) versus memory of the DNNs obtained with the grid search and mixed precision exploration. The precision used is depicted with different colors (e.g., network-level quantization at 8-bit in red), with each point representing a different architecture (e.g., different number of channels). The graph uses logarithmic axes and the global Pareto front is represented as a dashed black line. The top-scoring models quantized at 8-bit and using sub-byte quantization (mixed or network-level) have been highlighted with black cycles.



(a) UniMiB-SHAR (b) UCI HAPT (c) WISDM (d) WALK

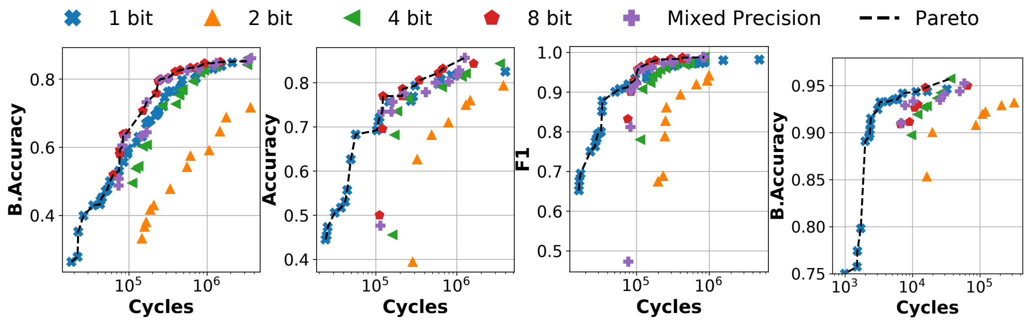
Fig. 5.5 Pareto fronts in terms of score vs memory of the models found with the architecture exploration. Each point shows a different architecture with distinct hyper-parameters.

The figure shows a similar trend for UniMiB-SHAR, WISDM, and UCI HAPT. On UniMiB-SHAR, 1-bit, 4-bit network-level quantization, and mixed precision yield the best results respectively at low (<45%), mid, and high (>75%) balanced accuracy. When comparing mixed-precision to the standard 8-bit quantization, I obtain a memory reduction of up to 72% at iso-accuracy. Moreover, thanks to the regularizing effect of the quantization [28], the top-performing model is 4-bit quantized, improving w.r.t to the most accurate 8-bit one by 1.23%.

Similar considerations can be applied to UCI HAPT, where 4-bit quantization is on the Pareto front for intermediate sizes, while mixed precision gains the edge for higher accuracy (>80%). Concerning the most accurate 8-bit model, a mixed-precision one achieves +1.4% additional accuracy (85.63% vs 84.23%), while requiring 77% less memory. When considering a reduced accuracy (<60%), BNNs gain the edge, performing well considering the complexity of the task.

Concerning the WISDM dataset, due to the simpler classification task, all sub-byte quantizations perform similarly, obtaining improved trade-offs w.r.t to 8-bit quantization. In particular, a mixed precision CNN can reduce the memory occupation by up to 66% w.r.t. to 8-bit quantization at iso-score.

Differently from the other datasets, BNNs occupy most of the Pareto front for WALK. Due to the task being only a binary classification, smaller and more compact models outperform the others, with the top balanced accuracy being achieved by 4-bit quantization at 95.74% while reducing the memory footprint by 91% w.r.t. the most accurate 8-bit model.



(a) UniMiB-SHAR (b) UCI HAPT (c) WISDM (d) WALK

Fig. 5.6 Pareto fronts in terms of score vs cycles of the models found with the architecture exploration.

Execution Cycles: Figure 5.6 shows the Pareto optimal models in terms of classification score versus inference clock cycles on Pulpissimo. Noteworthy, while points and colors have been kept identical in meaning w.r.t. Figure 5.5, the Pareto models represented are different, as an optimal architecture in terms of accuracy vs memory does not necessarily perform as well in terms of accuracy vs cycles. In fact, sub-byte quantization introduces an overhead in terms of packing/unpacking operations, needed before performing any computations on general-purpose MCUs such as Pulpissimo. This is clearly shown in Figure 5.6, where models with 8-bit quantization are generally Pareto optimal for high accuracy. Notably, BNNs implemented using the kernels introduced in this work, are still Pareto optimal in terms of cycles for the WALK dataset, as the overhead is generally minimal. Mixed-precision solutions can be Pareto optimal, as the overhead due to packing/unpacking is minimal and they may feature improved accuracy w.r.t. 8-bit quantization.

Dataset	**Score Range [%]**	**Memory Range [kB]**	**Cycles Range** $[\cdot 10^3]$
UNIMIB-SHAR	26.24:86.24 [BAcc.]	0.41:23.17	18.4:3316
UCI HAPT	44.53:85.63 [Acc.]	0.42:7.54	24:1253
WISDM	67.6:98.9 [F1]	0.22:6.22	16:859
WALK	78.13:95.74 [BAcc.]	0.05:1.65	0.9:36.4

Table 5.1 Summary of the characteristics of the Pareto optimal models for each dataset.

Table 5.1 summarizes the characteristics of the models found with the proposed exploration. Specifically, it reports the ranges of classification score, memory, and cycles of the models on the Pareto front of each dataset. Notably, CNNs on the Pareto fronts spawn up to two orders of magnitude both in memory and cycles and $\pm$ 60% in terms of classification metrics.

Comparison with state-of-the-art

	Ours (Best)		**Previous DL Works**			**Baseline RF**	
Dataset	**Score [%]**	**Mem. [kB]**	**Paper**	**Score [%]**	**Mem. [kB]**	**Score [%]**	**Mem. [kB]**
UniMiB	86.24/90.66 [BAcc./F1]	23.16	[93]	n.a./77.8 [BAcc./F1]	5800	58.05/65.61 [BAcc./F1]	202.17
UCI HAPT	85.63 [Acc.]	7.53	[91]*	97.98 [Acc.]	17939	74.16 [Acc.]	51.71
WISDM	98.9/98.81 [F1/Acc.]	6.21	[93]	98.8/n.a. [F1/Acc.]	1640	93.91/94.16 [F1/Acc.]	255.74
WALK	95.74 [BAcc.]	1.64	n.a	n.a	n.a	91.86 [BAcc.]	8.26

Table 5.2 Comparison with state-of-the art. *8 class classification task.

Table 5.2 reports a comparison between the most accurate quantized DNNs found in this work, other DL solutions and Random Forest found in the literature both in terms of memory and classification accuracy. Notably, I refer to the state-of-the-art results both for DNNs and RFs, reporting their accuracy and when available, the model size. As the WALK dataset was introduced in this work, no comparison is provided. Note that, as state-of-the-art papers implementing classic ML algorithms provide the actual memory footprint of the models, I compare my results only in terms of score. To provide a comparison in terms of memory, I perform a grid search on the depth $(1,20)$ and number of trees $(1,20)$ of an RF trained on each dataset, both using the raw data and the features proposed in [111, 112, 65] and reporting the memory footprint of the most accurate model.

The CNNs introduced in this work outperform the other solutions for three datasets out of four (UniMIB-SHAR, WISDM and WALK). Concerning UCI HAPT, the reference work focuses on a simplified version of the classification task, featuring only 8 classes instead of 12, making a direct comparison meaningless. However, the proposed CNNs still perform well in this case, featuring far fewer parameters than the comparison models.

In particular, the proposed CNNs achieve reductions in terms of model size by up to 2400x for UCI HAPT and 250x for UniMIB-SHAR and WISDM, obtaining also better performances in terms of prediction quality. More importantly, all the models introduced in this work fit in less than 256 kB of memory, 50% of the available size on Pulpissimo. On the contrary, previous DL solutions would not be deployable. Moreover, while Table 5.2 reports only the top-scoring sota models, even the less accurate architectures found in those works are too large, showing how this work is crucial for improving the state-of-the-art for HAR models deployed on cheap and low-power MCUs. Notably, this work is still relevant for less constrained devices, such as smartwatches, as the accurate yet small models proposed can be deployed for inexpensive inferences.

When compared to RFs, the proposed CNNs are more accurate and require lower memory (ranging from a 5x to a 41x reduction). Note that for UniMiB-SHAR and WISDM, the reference work [65] achieves up to 82.86% balanced accuracy with a k-NN and 81.48% with an RF. However, k-NNs require the on-device storage of the training data ($\geq$7 MB based on the feature reported by the paper and assuming a float representation), an unfeasible option for deployment on MCUs. The RF instead features 300 trees and since the number of nodes is not reported, estimating the memory is impossible. Nonetheless, even with 5 trees at depth 16, the memory becomes immediately higher than 23.16 kB. On WISDM, the reference work [112] achieves 99.8% accuracy with a k-NN. Nonetheless, accuracy is not a reliable metric for imbalanced datasets. Moreover, the memory requirements of a k-NN for WISDM would be of at least 12.6 MB. Concerning UCI HAPT, the reference work [111] achieves 88% accuracy on the 12-classes datasets with an RF. However, the authors provide no details on the hyperparameters of the RF, making the estimation of the memory unfeasible.

Detailed Deployment Results

Dataset	Config	Score [%]	Metric	Memory [kB]	Energy [μJ]	Latency [ms]
	Min	26.24		0.41	0.34	0.09
UniMiB-SHAR	Max - 5%	81.56	B.Acc.	9.28	29.33	7.7
	Max	86.24		23.17	61.59	16.2
	Min	44.53		0.42	0.44	0.12
UCI HAPT	Max - 5%	83.18	Acc.	4.6	30.52	8.01
	Max	85.63		7.54	23.27	6.11
	Min	67.6		0.22	0.3	0.08
WISDM	Max - 5%	94.74	F1	1.27	4.09	1.07
	Max	98.9		6.22	15.94	4.19
	Min	78.13		0.05	0.03	0.009
WALK	Max - 5%	91.81	B.Acc.	0.18	0.05	0.016
	Max	95.74		1.65	0.67	0.18

Table 5.3 Deployment results at different trade-off points in terms of score vs memory.

Table 5.3 reports the detailed metrics of some of the Pareto optimal CNNs found with the proposed search. Specifically, for each dataset, I report the most (*Max*) and least (*Min*) accurate models found with the exploration, and additionally, an intermediate one, i.e., the smallest model with a maximum drop of classification accuracy of 5% w.r.t to Max.

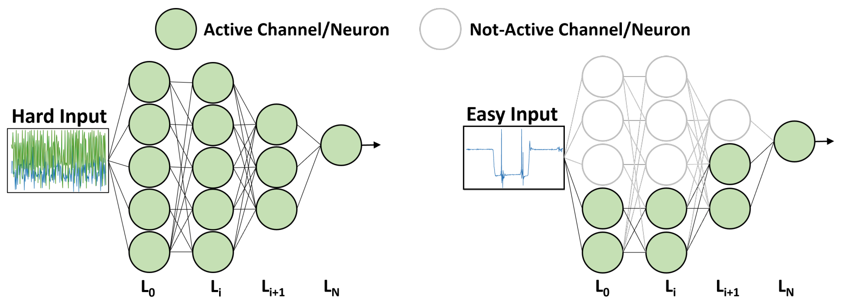


Fig. 5.7 Dynamic Inference with a variable width network.

Thanks to the proposed exploration, the models found spawn not only different orders of magnitude in terms of memory but also in terms of energy and latency. Notably, even the most accurate model is suitable for real-time inference, as its total latency is only 16 ms, a far shorter time than the one required to collect an entire window of samples for UniMiB-SHAR (1.28 s).

The $Max - 5\%$ CNNs often offer interesting trade-offs, significantly reducing the latency at the cost of a small accuracy drop. The most notable example is for WALK, where the Max-5% model is more energy and memory-efficient by an order of magnitude w.r.t to the Max model, while still achieving 91.8% accuracy.

5.3 Dynamic Slimmable Inference

5.3.1 Introduction

In this Section, I extend the results obtained from the previous flow (shown in Figure 5.1), introducing a dynamic inference mechanism for HAR networks.

5.3.2 Methodology

Figure 5.7 shows an overview of the proposed adaptive mechanism, based on [40] for 2D CNNs. For the sake of clarity, the image shows only FC layers, even if, in practice, we apply this approach to CNNs.

The main idea is to run a first inference using a subset of the channels/neurons of the whole network, that should be enough for easy inputs, thus saving computations.

In case of uncertain predictions, the whole network is run, still maintaining high accuracy on complex inputs.

The advantage of this approach is that parameters are shared between the adaptive models, resulting in no overhead in terms of memory. To train the networks, I refer to the procedure introduced in [113], as opposed to the original one introduced in [40], extending it to work for 1D CNNs. Specifically, at training time, the forward step is repeated for each supported width (i.e., the number of channels used by the reduced-size models) for each batch of inputs.

The optimization step is then performed after accumulating all the gradients from the forward steps. On the contrary, the original training algorithm introduced in [40], required performing incremental training on the network, starting from the smallest to the largest. Specifically, after training a network, its weights are frozen and the following one is trained.

Following the mechanism proposed in [113], I relaxed the constraints on the shared parameters, introducing the *Switchable Batch Norm*. Intuitively, using a subset of the channels of a convolution changes the mean and the standard deviation collected at training time by the batch norm, causing a degradation of accuracy as these statistics are misleading. Therefore, in [113] the authors add a BN layer per width after each convolution. Notably, these networks take the name of *slimmable* and they are not dynamic, as the user has to manually select which model should be run.

Concerning the adaptive policy to use, I selected the *Score Margin* mechanism, as it is lightweight yet accurate in estimating the prediction confidence.

A crucial point when deploying an adaptive model is selecting the best static architecture to be used as a baseline. For variable-width models, as the one proposed, this includes choosing the optimal hyperparameters for M_l and the number of channels to be used by reduced-width architecture M_s. Nonetheless, this design choice is generally not explored in the literature and is often assumed as a given.

In this work, I introduce a systematic way to select a promising architecture for dynamic inference. Importantly, selecting the most accurate model from the search space shown in Figure 5.6 is not ideal, as large CNNs often gain small amounts of accuracy but explode in terms of parameters. As a consequence, starting from such a model would lead to sub-optimal results, as even at reduced width, the obtained architecture would still be complex, achieving almost iso-accuracy w.r.t to the full-width model and nullifying the results obtained with the adaptive inference. On the other hand, starting from a model too small causes the reduced-width network to perform poorly, severely degrading the accuracy and forcing an almost constant use of the full-width architecture.

Therefore, in this work, I use the following metric to score the models on the Pareto front:

$$G_i = \frac{P(M_i) - P(M_{i-1})}{C(M_i) - C(M_{i-1}} \tag{5.4}$$

where $C(M_i)$ and $P(M_i)$ are respectively the inference cost and classification score of the i_{th} model of the Pareto architectures sorted by classification score. M_l is then selected using the architecture with the largest gain G_i and with at most an accuracy drop of 5% w.r.t to the most accurate Pareto-optimal model. The width used to run M_s is empirically selected between 25% and 50%.

5.3.3 Results

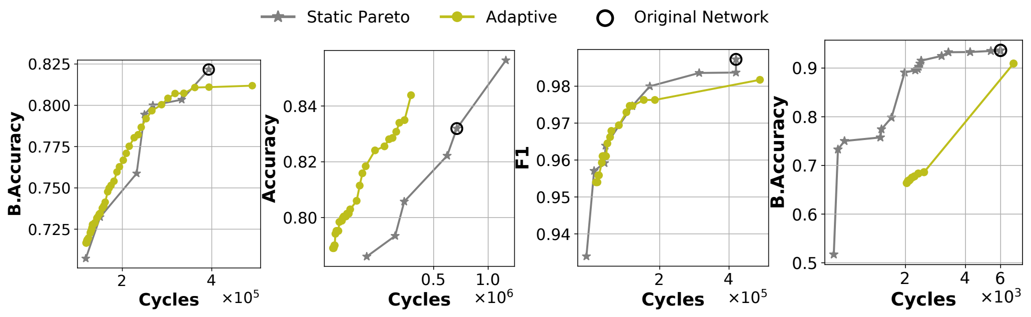
Experimental Setup

The experimental setup is kept identical to the one detailed in the previous section, with benchmarks on UniMiB-SHAR, UCI HAPT, WISDM and WALK. For UniMiB-SHAR and WALK, the width selected for M_s is 50%, while for WISDM and UCI HAPT is 25%.

Adaptive Inference

Figure 5.8 shows the results obtained when converting the model taken from Figure 5.6 that maximizes the gain from Equation 5.3.2 into its variable-width counterpart.

The x-axis reports the mean cycles per input over the whole test set. The gray line is the global Pareto front of Figure 5.6, with the starting static model highlighted in a black circle. The static Pareto front has been zoomed in the zone where the adaptive points lie for better visualization. The yellow points are obtained by the adaptive model varying the values of the confidence threshold th and represent different execution modes favoring either accuracy or energy efficiency.



(a) UniMiB-SHAR (b) UCI HAPT (c) WISDM (d) WALK

Fig. 5.8 Dynamic inference results. Gray points are different static models while yellow points belong to the same dynamic model.

Concerning UniMiB-SHAR and WISDM, the adaptive Pareto front lies close to the static one, except for the points featuring the largest value for th, as in that case, the overhead of performing an inference both with M_l and M_s is at its peak. Importantly, the class distribution of the test set of UniMiB-SHAR is significantly skewed towards the most complex tasks, with rare movements (e.g., falling backward) being more frequent than the samples of the *walking* class. With a more realistic class distribution (e.g., easier classes are generally far more frequent than complex ones), an adaptive approach would perform significantly better.

This is clearly shown for UCI HAPT, where the adaptive curve significantly outperforms its static counterpart. In this case, the adaptive approach reduces the average cycles per inference by up to 60% with no accuracy drops. To further explore how an adaptive approach improves when the frequency of the easy classes changes, I scored the adaptive model on two augmented versions of the test set of UCI HAPT, where the laying class (that is normally 15% of the test set) becomes 10x or 20x times more frequent. Notably, the augmented dataset matches more closely real-world scenarios, where easy classes such as *laying* are recorded for hours (e.g. while sleeping) and complex classes are rare (e.g. falling backward).

Figure 5.9 shows the obtained adaptive curves, clearly demonstrating that the input-dependent approach proposed becomes even more advantageous.

WALK represents the only dataset where variable-width networks did not achieve satisfying results. The reason is that the starting model for this dataset is a BNN having as the final layer a 1-bit FC layer. BNNs, while small and quite accurate, are generally not well calibrated [114], making a policy based on logits ineffective. On

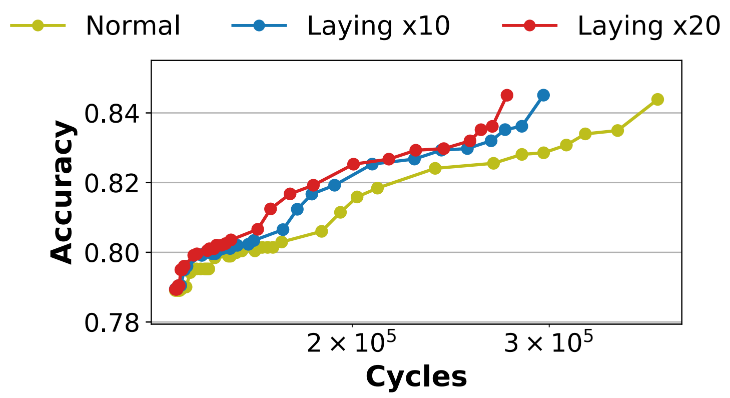


Fig. 5.9 Dynamic Inference scored on the original (Normal) and augmented (10x and 20x) versions of the UCI HAPT dataset.

the other hand, an adaptive model spawned from a higher precision architecture is not competitive, as it would be too far from the static Pareto curve.

Nonetheless, the main purpose of adaptive inference is not to outperform the static curve, but rather to offer multiple trade-off points in terms of accuracy versus cycles while deploying a single model.

Dataset	Score Range [%]	Approach	Cycles Range [$\cdot 10^5$]	N. of Points	Tot. Memory [kB]
UniMiB-SHAR	71:81 [BAcc.]	Static	1.6:3.1	5	50.6
		Adaptive	1.5:5.4	47	26.34
UCI HAPT	78:83 [Acc.]	Static	3.06:6.71	4	39.4
		Adaptive	1.38:3.75	34	14.63
WISDM	95:98 [F1]	Static	1.03:1.81	5	14.5
		Adaptive	1.05:5.44	21	11.96

Table 5.4 Detailed deployment results

Table 5.4 reports the memory requirements of a naive implementation of a runtime-dependent execution, that is deploying all the networks in the adaptive solution range. As shown, a static deployment would require up to 2.7x more memory, while providing significantly fewer trade-off points.

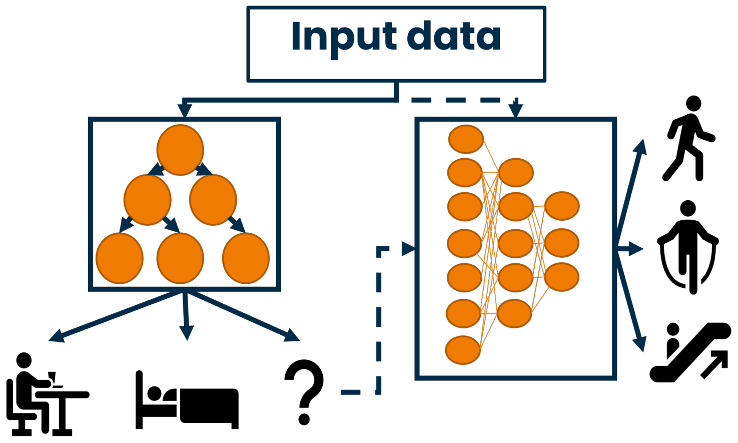


Fig. 5.10 Dynamic Hierarchical model composed by a DT and CNN. Easy classes are classified by the DT (left) and complex ones by the CNN (right).

5.4 Multi-Model Adaptive Hierarchical Inference

5.4.1 Introduction

In this section, I introduce a dynamic inference approach based on hierarchical classifiers, that leverages the different complexity of the activities commonly present in HAR applications. As for the other dynamic inference approaches, the goal is to reduce the average energy/latency per inference by running a reduced set of computations for easy classes. The approach illustrated in this section has been published in [115].

5.4.2 Dynamic Hierarchical Inference

Figure 5.10 shows an overview of the proposed flow, mixing a DT with a CNN, both working on different sub-tasks, i.e. recognizing a different set of activities. The DT is trained to recognize only the easy activities in the HAR dataset and on an additional fallback class. Only when predicting the latter, the CNN, trained only on the complex activities, is enabled. Intuitively, as easy activities such as *laying* and *walking* are more frequent in real-life scenarios, the CNN is rarely enabled.

Notably, this approach presents two main strengths w.r.t the dynamic inference introduced in Section 5.3. The reduced complexity of the sub-task enables the

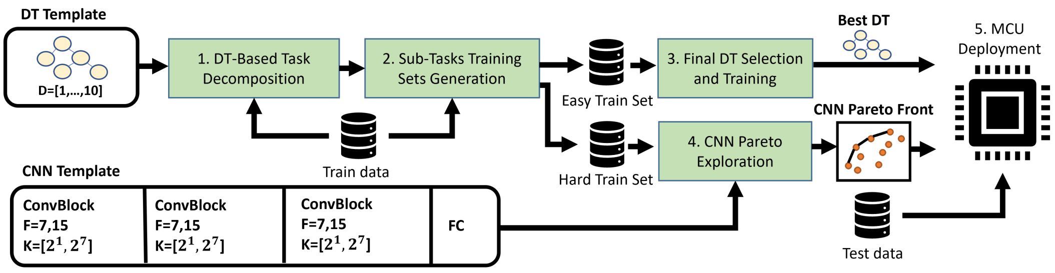

DT Template
D=[1,...,10]
1. DT-Based Task Decomposition
2. Sub-Tasks Training Sets Generation
Easy Train Set
Hard Train Set
Train data
3. Final DT Selection and Training
4. CNN Pareto Exploration
Best DT
CNN Pareto Front
5. MCU Deployment
Test data
CNN Template
ConvBlock F=7,15 K=[2^1, 2^7]
ConvBlock F=7,15 K=[2^1, 2^7]
ConvBlock F=7,15 K=[2^1, 2^7]
FC

Fig. 5.11 Hierarchical training flow.

deployment of a small initial model (as opposed to a reduced-width CNN) such as a DT. Moreover, as the CNN used to determine the complex activity operates on a limited number of classes, it generally becomes more accurate at iso-memory/latency than its counterpart trained on the whole set of activities.

Finally, there is no need to design and test the effectiveness of an early-stopping policy, such as the Score Margin, as this is handled automatically by the DT and the *fallback* class.

On the other hand, the complexity of this approach is moved to selecting the correct easy activities/classes to be recognized, a process detailed in the following Section.

5.4.3 Training a hierarchical classifier

Figure 5.11 shows an overview of the training and deployment flow proposed. It can be divided into five steps: *DT-Based Task decomposition*, *Sub-task training set generation*, *Final DT Selection and Training*, *CNN Pareto Exploration*, and *MCU deployment*.

DT-Based Task decomposition

Determining the activities to be handled by the DT, i.e. the easy classes M_{easy}, plays a fundamental role. Intuitively, a misclassification caused by the DT leads immediately to an error, as the CNN is not trained on the easy classes. On the contrary, the big model in a big/little approach can still correct a wrong output of the little model, as long as it is enabled (e.g. the little model outputs the wrong class but with low confidence).

Therefore, to determine the easy classes, I train several DTs featuring different hyper-parameters on the whole dataset, i.e. the original HAR task with all classes. In this case, I explore all DTs with depth in the interval $2, 10$. Then, I select as easy classes M_{easy}, the two activities featuring the highest F1 score. Notably, in this work, I explore only $M_{easy} = 2$, but this value becomes a hyperparameter that should be explored, as the concept of *easy* class changes with the dataset/task. The classes not selected with the aforementioned approach are considered hard.

Sub-task training sets generation

The training set for the DT is generated leaving the samples belonging to the easy classes unchanged. Samples belonging to hard classes are instead clustered into a single *fallback* class. Concerning the training set for the CNN, the easy classes and samples are entirely removed from the training data, as they are not considered in the second step of the inference.

Final DT Selection and Training

The final step consists of an additional grid search of the DT hyperparameters (i.e., depth) using the modified training set and then selecting the most accurate model.

CNN Pareto Exploration

Concerning the CNN, I extract the entire Pareto front in terms of accuracy vs energy of the hyperparameter search performed. The reason is that, as the CNN is the most computationally demanding part of the dynamic inference, starting from different architectures is the easiest model to obtain different trade-off points in terms of accuracy vs energy. Specifically, the architecture exploration is performed starting from the template shown in Figure 5.11, designed after LeNet [100]. It features three consecutive ConvBlocks (each composed of a Conv1D layer, a BN layer, and a MaxPooling layer) and a final FC layer. I explore all variants of the template by varying the number of output channels (K) and kernel size (F) of the convolutional layers. Both MaxPool's window and stride are equal to 2 and are kept fixed during the search. Networks are quantized at 8 bits with QAT [34].

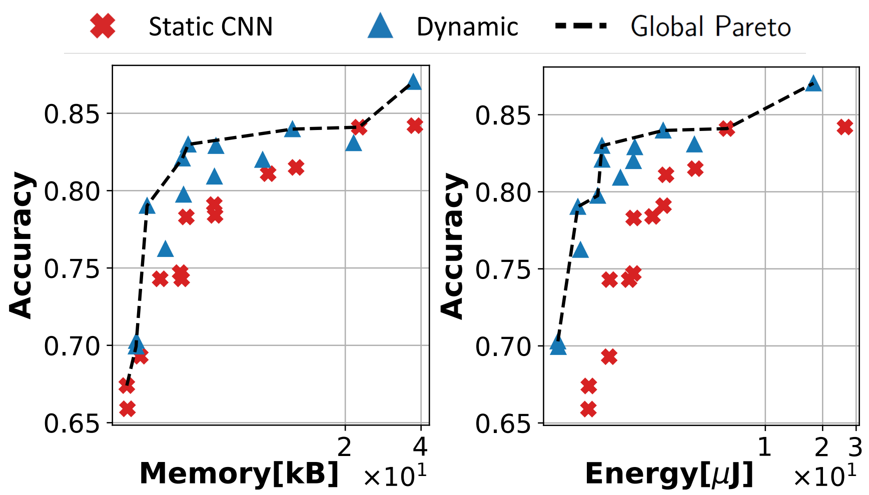


Fig. 5.12 Average energy vs accuracy and total memory vs accuracy results for the dynamic model (*Dynamic*) and the static CNN.

MCU Deployment

The CNNs and the DT obtained are deployed using the optimized libraries [109] for the target MCUs.

5.4.4 Results

Experimental Setup

I deploy the models on Pulpissimo using as a benchmark the UCI HAPT dataset. The easy classes are *sitting* and *laying*, having an F1-score above 85%. The samples belonging to these two activities are 30% of the total training and test set.

Deployment on MCU

Figure 5.12 shows the results in terms of memory and energy obtained when deploying the proposed hierarchical architecture. Each blue triangle represents a different dynamic model, composed of the same DT and a different Pareto-optimal CNN obtained from the architecture search (both shown in Figure 5.11). For comparison, I report also the Pareto fronts of the static CNNs, trained on the original task (i.e.,

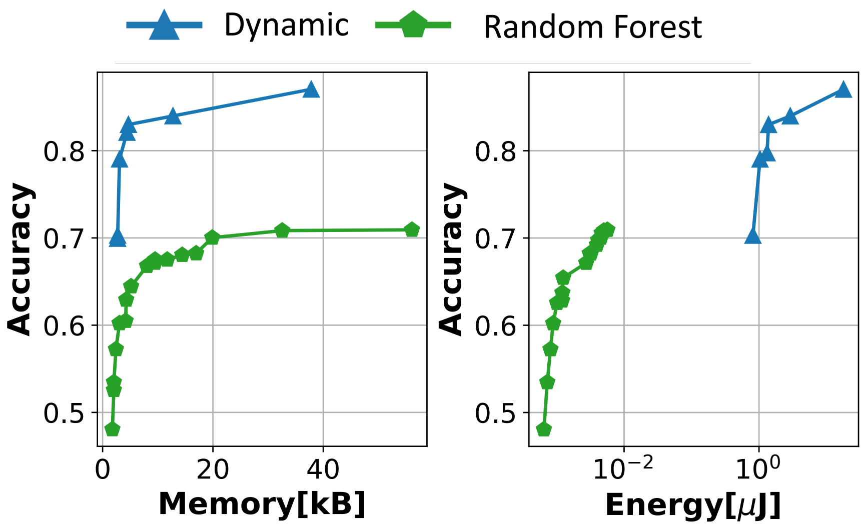


Fig. 5.13 Accuracy vs memory and accuracy vs average energy consumption for the dynamic model and a RF.

using all classes). The dashed line highlights the global Pareto front. Energy results refer to the average consumption per inference, over the entire test set.

Concerning energy consumption, the proposed approach is almost always Pareto optimal, aside from a single architecture at 85% accuracy. In particular, the hierarchical dynamic inference can improve the accuracy by up to 12.34% for the same energy, or reduce the energy per inference by up to 67.7% at iso-accuracy. Moreover, as detailed in the previous section, the simplification of the task performed by the CNN allows the proposed architecture to outperform by up to 3% accuracy the most accurate static model. Notably, the DT is also very accurate, yielding 92% accuracy on the easy task, while requiring 341x less energy per inference than the smallest CNN.

Regarding the memory, the size of the DT is 200 B. However, the deployed CNN in the hierarchical setup needs only M_{hard} output neurons. Therefore, the entire DT+CNN architecture introduces minimal overhead (9%) in the case of small CNNs ($\leq 3kB$). For large CNNs instead, the memory saved by reducing the size of the last FC layer outweighs the DT footprint. For this reason, together with the increased accuracy of the dynamic model, I achieve a reduction of up to 64% memory for the same accuracy.

	Mode	Acc.	Memory[kB]	Energy[μj]	Latency[ms]
Base	Ours	0.87	37.2	17.9	4.7
	Static	0.84	37.8	26.2	6.9
10x	Ours	0.89	37.2	6.2	1.6
	Static	0.85	37.8	26.2	6.9
20x	Ours	0.90	37.2	4.4	1.2
	Static	0.86	37.8	26.2	6.9

Table 5.5 Deployment results on the original and the synthetically augmented test set.

Figure 5.13 shows a comparison of an RF with the proposed approach both in terms of memory and energy. The RFs shown are the models with estimators in the interval $1,15$ and depth in $2,20$. As expected, the DL-based approach is far more accurate, rapidly becoming Pareto optimal in terms of memory. On the other hand, RFs are significantly more energy efficient at lower accuracy.

Finally, Table 5.5 reports the deployment details of the most accurate static CNN and dynamic architecture shown in Figure 5.12. Mirroring the idea of the synthetic augmentation shown in Section 5.3, I augment the test set by over-sampling the easy classes either by 10x or 20x. The reason for such an experiment is again to show the effectiveness of the proposed approach in a scenario closer to the real world, where activities like *laying* and *sitting* are far more frequent than the rest. Notably, compared to static CNNs, I reach an improved accuracy of up to 4%, while saving up to 76% or 83% energy/latency at iso-accuracy respectively with a 10x and 20x augmentation.

5.5 Dynamic Inference with Class-Dependent Confidence

5.5.1 Introduction

The standard Score Margin applies the same threshold independently on the class predicted. However, such an approach assumes that all the classes are equally complex to predict, a rare occurrence in embedded scenarios. Starting from this intuition, I propose an enhanced version of the Score Margin, that leverages a threshold for each class. Furthermore, I provide an automated approach to compute

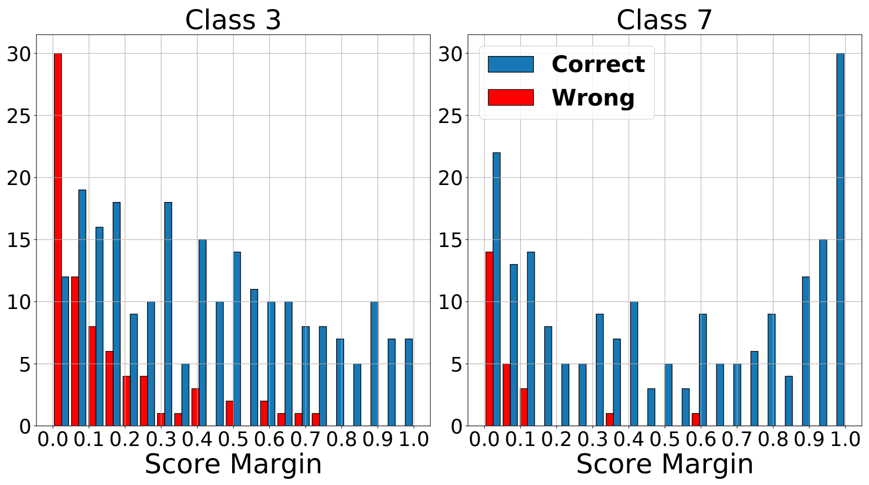


Fig. 5.14 Distribution of the SM depending on the class.

the optimal set of thresholds given a user-defined trade-off in terms of accuracy versus energy. This work has been published in [116].

5.5.2 Multi-threshold Score Margin

Single-threshold policies such as Score Margin, implicitly assume that the small model M_s in approaches such as the Big/Little is equally accurate on all the inputs. Nonetheless, this assumption rarely holds.

Figure 5.14 shows the SM distribution of all the samples belonging to classes 3 and 7 of the GTSRB validation set, obtained when using a single FC layer (i.e., a logistic regressor). Red bars in the histogram denote the samples misclassified by the model, while the blue ones denote the correct ones. If we considered only class 7 (right plot), we would have selected for the SM a th of 0.15, as it forces the activation of the big model M_l for almost all wrong samples. Nonetheless, considering class 3, we see that if we set $th = 0.15$ we enable M_l too few times, introducing a significant accuracy degradation. The insight is that when M_s predicts that an input belongs to class 3, the dynamic inference should treat the value with high skepticism, significantly raising th. On the other hand, when class 7 is predicted, a low th is sufficient to achieve high accuracy and should be selected to avoid unnecessary inferences with M_l.

As a consequence, I introduce a policy named *Class Score Margin* that features a threshold for each class, assigned depending on the class complexity on the validation set.

Specifically, I find the optimal class threshold th_c with the following equation:

$$th_c = argmin_{th_c}(FP_c(th_c) + \lambda E_c(th_c)) \tag{5.5}$$

Equation 5.5.2 features two contributions, the first depending on the classification accuracy and the second on the energy cost of the dynamic inference. Specifically, $FP_c(th_c)$ is the number of false positives generated per class and is computed as follows:

$$FP_c(th_c) = \sum_{j:true(j)\neq c}^{M_s^c} (SM_s(j) > th_c \vee M_l(j) \neq true(j)) \tag{5.6}$$

where M_s^c is the number of inputs that M_s predicts as belonging to class c, $true(j)$ is the true label of input j and $SM_s(j)$ is the score margin of M_s for j. The second contribution of Equation 5.5.2 instead can be computed as:

$$E_c(th_c) = \sum_{j=1}^{M_s^c} SM_s(j) \leq th_c \tag{5.7}$$

that is the number of times the big model is enabled. Noteworthy, the factor λ in Equation 5.5.2 balances the two contributions, giving more importance either to the accuracy or the energy.

The sets of thresholds obtained with a specific λ can be pre-computed offline and switched at runtime by the user. Intuitively, storing them on the device introduces a minimal overhead, as the number of classes in embedded tasks is generally limited.

Figure 5.15 shows the objective function of the two addends of Equation 5.5.2, with the minimum being highlighted with a black dot. Specifically, the plots depict two values of λ, showing that increasing it causes both an energy reduction and a small decrease in accuracy. Note that the minimization of Equation 5.5.2 requires a single inference pass on the validation set, storing the output logits of the network. Then, the desired th_c can be obtained with any minimization method, with no further executions of the models needed.

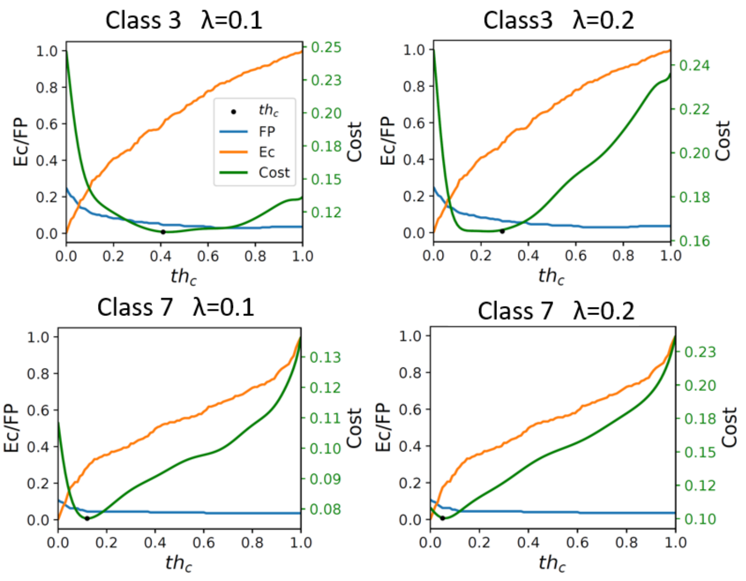


Fig. 5.15 Objective function for a LeNet5-like model for two values of alpha and two classes of the GTSRB dataset.

To conclude, the class score margin is an enhanced version of the Score Margin tailored for datasets presenting class imbalance. Computing several thresholds for each class requires a single offline inference on the validation set, storing then them on the device. Therefore, at runtime, a user can select the most suitable operating mode with little to no overhead. Moreover, this approach becomes identical to the classical Score Margin for datasets with identical class complexity, being outperformed by the latter only in case of mismatches between the test and validation set. Note also that this adaptive policy is orthogonal to the dynamic inference type, and can be used in other works with no modifications.

5.5.3 Results

Experimental Setup

I benchmark the class score margin on three datasets: CIFAR10, GTSRB, and GSP. For CIFAR10 and GSP, I used as little model a Lenet-5 [100] and as big model a MobilenetV1 [117]. On GTSRB, the little model is a logistic regressor, while the big model is a LeNet-5. The deployment platform is the STM32H743 MCU by STMicroelectronics, featuring an ARM Cortex-M7. Results have been obtained by deploying the floating-point models with X-CUBE-AI.

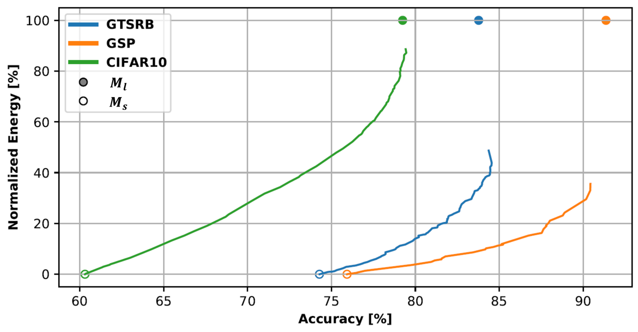


Fig. 5.16 Energy vs Accuracy results of the proposed methods.

Energy versus Accuracy Trade-off

Figure 5.16 shows the trade-offs between the average energy per input and accuracy obtained for the three datasets when using the proposed policy. Curves are obtained by changing the values of λ, with the dots representing the accuracy and energy of the small and big models when used statically.

	Accuracy [%]	Energy [mJ]
CIFAR10 [15]	79.46 [+0.22]	98.76 [- 11.4%]
GTSRB [17]	84.54 [+0.76]	10.98 [- 56.0%]
GSP [16]	90.43 [-1.01]	40.01 [- 64.4%]

Table 5.6 Maximum accuracy and energy of the proposed methods. Relative differences w.r.t. M_l are reported in brackets.

Table 5.6 reports the maximum accuracy achieved on each dataset, together with the energy required to obtain it. In square brackets, I report the relative difference w.r.t. an execution of M_l. Specifically, I obtain up to 0.76% additional accuracy, while reducing the energy by up to 56%. Even on GSP, where the dynamic approach introduced a minimal accuracy degradation w.r.t. the big model (-1.01%), the energy savings are outstanding, reducing the inference cost by up to 64.4%.

Comparison with state-of-the-art

Table 5.7 reports a comparison between the proposed policy and the classical SM. Specifically, I report the mean energy cost per inference at different trade-off points in terms of accuracy gained w.r.t. to M_s. The difference between the proposed approach and the SM is reported in square brackets. For instance, the column named *25%* reports the energy required to reach the accuracy of M_s plus 25% of the difference

	Energy [mJ] @ Normalized accuracy gain w.r.t. M_s				Max. reduction
	25%	50%	75%	100%	Acc/Energy
CIFAR10	12.37 [-9.6%]	27.46 [-4.5%]	46.76 [-2.0%]	94.26 [-3.2%]	63.00/8.98 [-14.1%]
GTSRB	2.85 [-4.4%]	4.08 [+1.0%]	5.70 [+6.6%]	8.98 [-26.4%]	84.40/10.05 [-59.3%]
GSP	5.44 [-24.1%]	10.81 [-14.88%]	17.95 [-12.53%]	34.51 [-17.28%]	79.8/5.44 [-24.1%]

Table 5.7 Energy consumption at different accuracy configurations. In brackets the difference w.r.t single-threshold SM.

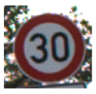
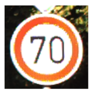
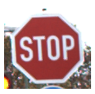
Class			
th$_c$	0.77	0.73	0.43
M_l calls [%]	54.9	63.3	3.2

Fig. 5.17 An example of *easy* and *complex* classes for the GTSRB dataset. % of M2 calls and th_c are for λ=0.05.

between the accuracy of M_l and M_s. The column named *100%* reports the energy value required to reach iso-accuracy with M_l, and so on. Finally, in the column *Max. reduction* I report the accuracy and the energy consumption of the point that yields the maximum gain in terms of energy w.r.t the classical SM. The class score margin policy outperforms the SM on almost all reported settings, achieving gains of up to 60% in terms of energy reduction. The few cases where the SM performs better can be attributed to mismatches between the validation and test set, leading to the derivation of slightly sub-optimal class thresholds. Moreover, the results obtained on CIFAR10 are the least impressive as the balanced nature of the task featured in this dataset makes the proposed approach less effective (i.e., closer to a classical SM).

Finally, in Figure 5.17, I report an example extracted from GTSRB, showing how the proposed approach assigns larger th_c to similar classes. The two speed-limit signs are similar to each other, confusing M_s, and thus are assigned a high threshold. On the other hand, the easily recognizable stop sign is assigned a lower th_c, as it is significantly different from other traffic signs present in the dataset.

Effects on imbalanced datasets

The proposed approach is effective not only when classes of different complexity are present in the dataset, but also when the class distribution is different between training and validation/test sets. This is a common occurrence when employing a

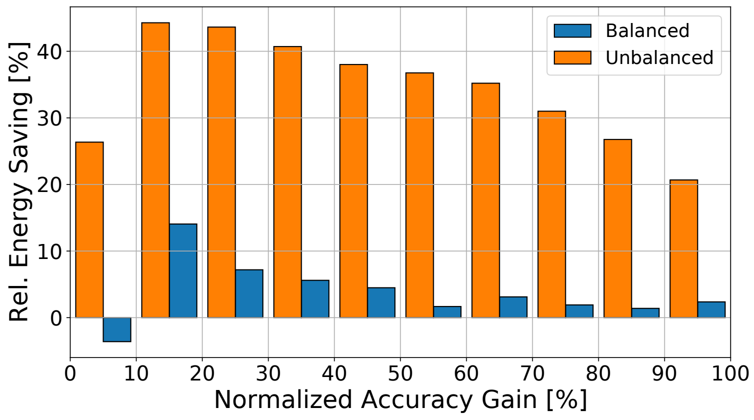


Fig. 5.18 Energy gains on the unbalanced and standard CIFAR10 dataset obtained with the class score margin.

pre-trained model on a different task and has a negative impact on the SM, as the model is not well calibrated.

The class score margin handles this imbalance, assigning the class threshold not according to the training data, but to the validation data. Differently from other approaches, it requires no time-consuming operations (e.g., retraining, fine-tuning), needing only an inference pass on the validation set.

To prove this, I artificially unbalanced the training set of CIFAR10, undersampling 8 random classes to 1/5 of the original images. After performing the training on the unbalanced training set, I computed the class thresholds on the (still balanced) validation set, finally evaluating the class score margin on the test set.

Figure 5.18 reports the gain in terms of energy obtained at different accuracy points (w.r.t. M_l) both for the Score Margin and the Class Score Margin. As shown, the proposed method becomes significantly dominant in an unbalanced setting (even if it was already performing well in the standard version of CIFAR10), with energy reductions of more than 40%.

Chapter 6

Conclusions

This dissertation introduced several optimization algorithms and approaches tailored for efficient deployments of Machine Learning models on MCUs. These techniques trade-off little to no accuracy to significantly reduce the memory footprint or the energy consumption of the models. Moreover, they are often orthogonal to each other, making them a suitable option for multiple deployment scenarios where other optimizations have been already introduced.

In Chapter 4, I focused on tree ensembles, a popular type of ML model for constrained IoT devices. In particular, I show that with a compact representation of the ensemble paired with quantization, the memory requirements can be reduced by more than 2x with almost no accuracy drops. Noteworthy, quantization, while widely explored in DL applications, has been far less explored in classical ML algorithms. Then, thanks to a dynamic inference mechanism, I showed that I can reduce the average inference cost in terms of energy/latency by up to 60%, with an accuracy drop lower than 1%. Differently from previous works providing only theoretical energy savings, I benchmarked the dynamic tree ensembles on multi-core IoT devices. The parallel execution of the decision trees is a challenging problem, that has not yet been tackled for dynamic inference in the literature. Noteworthy, the optimizations introduced have been collected and included in an open-source library, automatically exporting tree ensembles into optimized C code for MCUs.

In Chapter 5, I focused on optimizations for deep learning models. First, I introduced an automated flow for deploying small yet accurate DNNs for Human Activity Recognition at the edge, thanks to sub-byte and mixed-precision quantization. The compressed DNNs achieve up to 98.9% accuracy and require at most 23.16 kB of memory, while also satisfying real-time latency constraints. Then, I focused on dynamic inference techniques to further improve the energy efficiency of DNNs. First, I introduce a single DNN architecture for HAR that is run dynamically at variable widths. That is, depending on the input complexity, only a subset of the channels/neurons in the DNN is executed. This approach leads to savings of up to 60% in terms of average energy per inference, with only an overhead in terms of memory of up to 9% and negligible loss of accuracy. Then, I introduce a dynamic hierarchical inference mechanism that exploits the different complexities of the classes in a dataset. The first model, a lightweight DT, recognizes only easy classes, while the second, an accurate CNN, only the complex classes. As easy classes are the most common, often we need only to perform an inference with the DT. This approach saves up to 67.7% energy on average per inference, with no accuracy drops w.r.t. deploying a single DNN trained on all the classes. Finally, I show a dynamic inference approach tailored for datasets featuring classes of different complexity, where state-of-the-art approaches for early stopping generally underperform. In this case, the savings in terms of energy range from 10-60%, with negligible accuracy drops.

In summary, this work pushes the boundary of lightweight yet accurate models that can be deployed on edge devices, enabling increasingly complex applications to be moved on ultra-low power devices. Furthermore, it highlights the effectiveness of dynamic inference approaches, showing their flexibility and outstanding performances even in the most resource-constrained deployment scenarios.

List of publications

In this appendix, I report a complete list of publications that I authored or co-authored during my PhD. This list includes publications that are not presented in this dissertation, but that are still tied to the optimization of Machine Learning algorithms for edge ML.

International Peer-Reviewed Journals

- **F. Daghero**, et al., “Dynamic decision tree ensembles for energy-efficient inference on IoT edge nodes,” IEEE Internet of Things Journal, vol. 11, no. 1. pp. 742–757, 2024. doi: 10.1109/JIOT.2023.3286276.
- C. Xie, A. Burrello, **F. Daghero**, et al., “Reducing the energy consumption of sEMG-Based gesture recognition at the edge using transformers and dynamic inference,” Sensors, vol. 23, no. 4. 2023. doi: 10.3390/s23042065.
- C. Xie, **F. Daghero**, et al., “Efficient deep learning models for privacy-preserving people counting on low-resolution infrared arrays,” IEEE Internet of Things Journal, vol. 10, no. 15. pp. 13895–13907, 2023. doi: 10.1109/JIOT.2023.3263290.
- **F. Daghero**, et al., “Human activity recognition on microcontrollers with quantized and adaptive deep neural networks,” ACM Transactions on Embedded Computing Systems, vol. 21, no. 4. 2022. doi: 10.1145/3542819.

International Peer-Reviewed Conferences

- K. S. Sidahmed Alamin, **F. Daghero**, et al., “Model-driven dataset generation for data-driven battery SOH models,” vol. 2023-August. in Proceedings of the International Symposium on Low Power Electronics and Design, vol. 2023-August. 2023. doi: 10.1109/ISLPED58423.2023.10244587.

- **F. Daghero**, et al., “Two-stage human activity recognition on microcontrollers with decision trees and CNNs.” in PRIME 2022 - 17th International Conference on Ph.D Research in Microelectronics and Electronics, Proceedings. pp. 173–176, 2022. doi: 10.1109/PRIME55000.2022.9816745.

- C. Xie, **F. Daghero**, et al., “Privacy-preserving social distance monitoring on microcontrollers with low-resolution infrared sensors and CNNs,” vol. 2022-May. in Proceedings - IEEE International Symposium on Circuits and Systems, vol. 2022- May. pp. 1332–1336, 2022. doi: 10.1109/ISCAS48785.2022.9937837.

- **F. Daghero** et al., “Ultra-compact binary neural networks for human activity recognition on RISC-V processors.” in Proceedings of the 18th ACM International Conference on Computing Frontiers 2021, CF 2021. pp. 3–11, 2021. doi: 10.1145/3457388.3458656.

- **F. Daghero**, et al., “Adaptive random forests for energy-efficient inference on microcontrollers,” vol. 2021-October. in IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC, vol. 2021- October. 2021. doi: 10.1109/VLSI-SoC53125.2021.9606986.

- **F. Daghero**, et al., “Energy-efficient adaptive machine learning on IoT end-nodes with class-dependent confidence.” in ICECS 2020 - 27th IEEE International Conference on Electronics, Circuits and Systems. 2020. doi: 10.1109/ICECS49266.2020.9294863.

References

[1] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[2] Shouvik Chakraborty and Kalyani Mali. An overview of biomedical image analysis from the deep learning perspective. *Research Anthology on Improving Medical Imaging Techniques for Analysis and Intervention*, pages 43–59, 2023.

[3] Maryam Parsa, Priyadarshini Panda, Shreyas Sen, and Kaushik Roy. Staged inference using conditional deep learning for energy efficient real-time smart diagnosis. In *2017 39th annual international conference of the IEEE engineering in medicine and biology society (EMBC)*, pages 78–81. IEEE, 2017.

[4] Zeki Murat Çınar, Abubakar Abdussalam Nuhu, Qasim Zeeshan, Orhan Korhan, Mohammed Asmael, and Babak Safaei. Machine learning in predictive maintenance towards sustainable smart manufacturing in industry 4.0. *Sustainability*, 12(19):8211, 2020.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[6] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[7] Alessio Burrello, Moritz Scherer, Marcello Zanghieri, Francesco Conti, and Luca Benini. A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–6. IEEE, 2021.

[8] Ambuj Mehrish, Navonil Majumder, Rishabh Bharadwaj, Rada Mihalcea, and Soujanya Poria. A review of deep learning techniques for speech processing. *Information Fusion*, page 101869, 2023.

[9] Wevolver Staff. 2023 Edge AI Technology Report — wevolver.com. https://www.wevolver.com/article/2023-edge-ai-technology-report. [Accessed 20-12-2023].

[10] Joao Pereira and Margarida Silveira. Learning representations from healthcare time series data for unsupervised anomaly detection. In *2019 IEEE international conference on big data and smart computing (BigComp)*, pages 1–7. IEEE, 2019.

[11] Francesco Daghero, Alessio Burrello, Chen Xie, Luca Benini, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Adaptive random forests for energy-efficient inference on microcontrollers. In *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2021.

[12] Daniele Jahier Pagliari, Matteo Ansaldi, Enrico Macii, and Massimo Poncino. Cnn-based camera-less user attention detection for smartphone power management. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.

[13] GAP9 processor | GreenWaves Technologies — greenwaves-technologies.com. https://greenwaves-technologies.com/gap9_processor/. [Accessed 25-03-2024].

[14] Justin Ker, Lipo Wang, Jai Rao, and Tchoyoson Lim. Deep learning applications in medical image analysis. *Ieee Access*, 6:9375–9389, 2017.

[15] lsdm6dsox sensor. https://www.st.com/en/mems-and-sensors/lsm6dsox.html.

[16] Zhong Chen, CB Sivaparthipan, and BalaAnand Muthu. Iot based smart and intelligent smart city energy optimization. *Sustainable Energy Technologies and Assessments*, 49:101724, 2022.

[17] Matthew Veres and Medhat Moussa. Deep learning for intelligent transportation systems: A survey of emerging trends. *IEEE Transactions on Intelligent transportation systems*, 21(8):3152–3168, 2019.

[18] Lefteris Benos, Aristotelis C Tagarakis, Georgios Dolias, Remigio Berruto, Dimitrios Kateris, and Dionysis Bochtis. Machine learning in agriculture: A comprehensive updated review. *Sensors*, 21(11):3758, 2021.

[19] Joao Pereira and Margarida Silveira. Learning representations from healthcare time series data for unsupervised anomaly detection. In *2019 IEEE international conference on big data and smart computing (BigComp)*, pages 1–7. IEEE, 2019.

[20] Oded Z Maimon and Lior Rokach. *Data mining with decision trees: theory and applications*, volume 81. World scientific, 2014.

[21] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[22] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[24] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[25] Francisco Javier Ordóñez and Daniel Roggen. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1):115, 2016.

[26] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[27] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[28] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[29] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[31] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[32] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[33] Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–21, 2019.

[34] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2018.

[35] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, pages 1–1, 2021.

[36] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[37] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[38] Simon Bernard, Laurent Heutte, and Sebastien Adam. On the selection of decision trees in random forests. In *2009 International joint conference on neural networks*, pages 302–307. IEEE, 2009.

[39] Eunhyeok Park, Dongyoung Kim, Soobeom Kim, Yong-Deok Kim, Gunhee Kim, Sungroh Yoon, and Sungjoo Yoo. Big/little deep neural network for ultra low power inference. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 124–132, 2015.

[40] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. Runtime configurable deep neural networks for energy-accuracy trade-off. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2016.

[41] Yu-Gang Jiang, Changmao Cheng, Hangyu Lin, and Yanwei Fu. Learning layer-skippable inference network. *IEEE Transactions on Image Processing*, 29:8747–8759, 2020.

[42] Francesco Daghero, Alessio Burrello, Chen Xie, Marco Castellano, Luca Gandolfi, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Human activity recognition on microcontrollers with quantized and adaptive deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 21(4), aug 2022.

[43] Serena Wang, Maya Gupta, and Seungil You. Quit when you can: efficient evaluation of ensembles by optimized ordering. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(4):1–20, 2021.

[44] Yacine Jernite, Edouard Grave, Armand Joulin, and Tomas Mikolov. Variable computation in recurrent neural networks. *arXiv preprint arXiv:1611.06188*, 2016.

[45] Minjoon Seo, Sewon Min, Ali Farhadi, and Hannaneh Hajishirzi. Neural speed reading via skim-rnn. *arXiv preprint arXiv:1711.02085*, 2017.

[46] Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. *arXiv preprint arXiv:1708.06834*, 2017.

[47] Adams Wei Yu, Hongrae Lee, and Quoc V Le. Learning to skim text. *arXiv preprint arXiv:1704.06877*, 2017.

[48] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[49] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. Sbnet: Sparse blocks network for fast inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 8711–8720, 2018.

[50] Xuanyi Dong, Junshi Huang, Yi Yang, and Shuicheng Yan. More is less: A more complicated network with less inference complexity. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5840–5848, 2017.

[51] Shijie Cao, Lingxiao Ma, Wencong Xiao, Chen Zhang, Yunxin Liu, Lintao Zhang, Lanshun Nie, and Zhi Yang. Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11216–11225, 2019.

[52] Shu Kong and Charless Fowlkes. Pixel-wise attentional gating for scene parsing. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1024–1033. IEEE, 2019.

[53] Zhenda Xie, Zheng Zhang, Xizhou Zhu, Gao Huang, and Stephen Lin. Spatially adaptive inference with stochastic feature sampling and interpolation. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*, pages 531–548. Springer, 2020.

[54] Thomas Verelst and Tinne Tuytelaars. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, pages 2320–2329, 2020.

[55] Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. Dynamic capacity networks. In *International Conference on Machine Learning*, pages 2549–2558. PMLR, 2016.

[56] Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2369–2378, 2020.

[57] Huiyu Wang, Aniruddha Kembhavi, Ali Farhadi, Alan L Yuille, and Mohammad Rastegari. Elastic: Improving cnns with dynamic scaling policies. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2258–2267, 2019.

[58] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D. Lane. Adaptive inference through early-exit networks: Design, challenges and directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, EMDL'21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 186–191. IEEE, 2020.

[60] Stm32h7. https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html. [Accessed 22-12-2023].

[61] Nxp lpc4300. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc4300-arm-cortex-m4-m0:MC_1403790133078#/. [Accessed 22-12-2023].

[62] 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)*, pages 1–4. IEEE, 2018.

[63] Jennifer R Kwapisz, Gary M Weiss, and Samuel A Moore. Activity recognition using cell phone accelerometers. *ACM SigKDD Explorations Newsletter*, 12(2):74–82, 2011.

[64] Jorge-L Reyes-Ortiz, Luca Oneto, Albert Samà, Xavier Parra, and Davide Anguita. Transition-aware human activity recognition using smartphones. *Neurocomputing*, 171:754–767, 2016.

[65] Daniela Micucci, Marco Mobilio, and Paolo Napoletano. Unimib shar: A dataset for human activity recognition using acceleration data from smartphones. *Applied Sciences*, 7(10):1101, 2017.

[66] Manfredo Atzori, Arjan Gijsberts, Claudio Castellini, Barbara Caputo, Anne-Gabrielle Mittaz Hager, Simone Elsig, Giorgio Giatsidis, Franco Bassetto, and Henning Müller. Electromyography data for non-invasive naturally-controlled robotic hand prostheses. *Scientific data*, 1(1):1–13, 2014.

[67] Hard Drive Test Data — backblaze.com. https://www.backblaze.com/cloud-storage/resources/hard-drive-test-data. [Accessed 20-12-2023].

[68] Alessio Burrello, Daniele Jahier Pagliari, Andrea Bartolini, Luca Benini, Enrico Macii, and Massimo Poncino. Predicting hard disk failures in data centers using temporal convolutional neural networks. In *Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020, Revised Selected Papers 26*, pages 277–289. Springer, 2021.

[69] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[70] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32:323–332, 2012.

[71] TensorFlow Speech Recognition Challenge. https://www.kaggle.com/c/tensorflow-speech-recognition-challenge. [Accessed 18-12-2023].

[72] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: An ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3. IEEE, 2018.

[73] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: a highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416. IEEE, 2021.

[74] Alfio Di Mauro, Francesco Conti, Pasquale Davide Schiavone, Davide Rossi, and Luca Benini. Always-on 674μ w@ 4gop/s error resilient binary neural networks with aggressive sram voltage scaling on a 22-nm iot end-node. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(11):3905–3918, 2020.

[75] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot end-point devices. *IEEE transactions on very large scale integration (VLSI) systems*, 25(10):2700–2713, 2017.

[76] Enrico Tabanelli, Giuseppe Tagliavini, and Luca Benini. Dnn is not all you need: Parallelizing non-neural ml algorithms on ultra-low-power iot processors. *ACM Transactions on Embedded Computing Systems*, 22(3):1–33, 2023.

[77] Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *Information Fusion*, 81:84–90, 2022.

[78] Francesco Daghero, Alessio Burrello, Chen Xie, Luca Benini, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Low-overhead early-stopping policies for efficient random forests inference on microcontrollers. In *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*, pages 25–47. Springer, 2021.

[79] Francesco Daghero, Alessio Burrello, Enrico Macii, Paolo Montuschi, Massimo Poncino, and Daniele Jahier Pagliari. Dynamic decision tree ensembles for energy-efficient inference on iot edge nodes. *IEEE Internet of Things Journal*, 11(1):742–757, 2024.

[80] Heping Zhang and Minghui Wang. Search for the smallest random forest. *Statistics and its Interface*, 2(3):381, 2009.

[81] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng-zhong Xu. Dynamic channel pruning: Feature boosting and suppression. *CoRR*, abs/1810.05331, 2018.

[82] Tianshi Gao and Daphne Koller. Active classification based on value of classifier. *Advances in neural information processing systems*, 24, 2011.

[83] Alexander G Schwing, Christopher Zach, Yefeng Zheng, and Marc Pollefeys. Adaptive random forest—how many "experts" to ask before making a decision? In *CVPR 2011*, pages 1377–1384. IEEE, 2011.

[84] Darius Morawiec. sklearn-porter. Transpile trained scikit-learn estimators to C, Java, JavaScript and others.

[85] Hyunsu Cho and Mu Li. Treelite: toolbox for decision tree deployment. 2018.

[86] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[87] Enrico Tabanelli, Giuseppe Tagliavini, and Luca Benini. Optimizing random forest-based inference on risc-v mcus at the extreme edge. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4516–4526, 2022.

[88] Akram Bayat, Marc Pomplun, and Duc A Tran. A study on human activity recognition using accelerometer data from smartphones. *Procedia Computer Science*, 34:450–457, 2014.

[89] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L Reyes-Ortiz. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In *International workshop on ambient assisted living*, pages 216–223. Springer, 2012.

[90] Nils Y Hammerla, Shane Halloran, and Thomas Plötz. Deep, convolutional, and recurrent models for human activity recognition using wearables. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1533–1540, 2016.

[91] Nguyen Thi Hoai Thu and Dong Seog Han. Hihar: A hierarchical hybrid deep learning architecture for wearable sensor-based human activity recognition. *IEEE Access*, 9:145271–145281, 2021.

[92] George Vavoulas, Charikleia Chatzaki, Thodoris Malliotakis, Matthew Pediaditis, and Manolis Tsiknakis. The mobiact dataset: Recognition of activities of daily living using smartphones. In *International Conference on Information and Communication Technologies for Ageing Well and e-Health*, volume 2, pages 143–151. SCITEPRESS, 2016.

[93] Yin Tang, Qi Teng, Lei Zhang, Fuhong Min, and Jun He. Layer-wise training convolutional neural networks with smaller filters for human activity recognition using wearable sensors. *IEEE Sensors Journal*, 21(1):581–592, 2021.

[94] Valentina Bianchi, Marco Bassoli, Gianfranco Lombardo, Paolo Fornacciari, Monica Mordonini, and Ilaria De Munari. Iot wearable sensor and deep learning: An integrated approach for personalized human activity recognition in a smart home environment. *IEEE Internet of Things Journal*, 6(5):8553–8562, 2019.

[95] Antonio De Vita, Alessandro Russo, Danilo Pau, Luigi Di Benedetto, Alfredo Rubino, and Gian Domenico Licciardo. A partially binarized hybrid neural network system for low-power and resource constrained human activity recognition. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 3893–3904, 2020.

[96] Preeti Agarwal and Mansaf Alam. A lightweight deep learning model for human activity recognition on edge devices. *Procedia Computer Science*, 167:2364–2373, 2020.

[97] Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. Dynamic Bitwidth Reconfiguration for Energy-Efficient Deep Learning Hardware. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, pages 47:1—-47:6. ACM, 2018.

[98] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi.

In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247, 2017.

[99] Francesco Daghero, Chen Xie, Daniele Jahier Pagliari, Alessio Burrello, Marco Castellano, Luca Gandolfi, Andrea Calimera, Enrico Macii, and Massimo Poncino. Ultra-compact binary neural networks for human activity recognition on risc-v processors. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 3–11, New York, NY, USA, 2021. Association for Computing Machinery.

[100] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[101] Matteo Risso, Alessio Burrello, Daniele Jahier Pagliari, Francesco Conti, Lorenzo Lamberti, Enrico Macii, Luca Benini, and Massimo Poncino. Pruning In Time (PIT): A Lightweight Network Architecture Optimizer for Temporal Convolutional Networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1015–1020, 2021.

[102] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, 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*, pages 12965–12974, 2020.

[103] 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 Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[104] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[105] Alessio Burrello, Alberto Dequino, Daniele Jahier Pagliari, Francesco Conti, Marcello Zanghieri, Enrico Macii, Luca Benini, and Massimo Poncino. TCN Mapping Optimization for Ultra-Low Power Time-Series Edge Inference. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2021.

[106] Nazareno Bruschi, Angelo Garofalo, Francesco Conti, Giuseppe Tagliavini, and Davide Rossi. Enabling mixed-precision quantized neural networks in extreme-edge devices. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 217–220, 2020.

[107] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. Bmxnet: An open-source binary neural network implementation based on mxnet. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1209–1212, 2017.

[108] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. dabnn: A super fast inference framework for binary neural networks on arm devices. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2272–2275, 2019.

[109] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philosophical Transactions of the Royal Society A*, 378(2164):20190155, 2020.

[110] Zhaowei Cai and Nuno Vasconcelos. Rethinking differentiable search for mixed-precision neural networks. In *CVPR*, 2020.

[111] Wesllen Sousa, Eduardo Souto, Jonatas Rodrigres, Pedro Sadarc, Roozbeh Jalali, and Khalil El-Khatib. A comparative analysis of the impact of features on human activity recognition with smartphone sensors. In *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*, pages 397–404, 2017.

[112] KH Walse, Rajiv V Dharaskar, and Vilas M Thakare. Performance evaluation of classifiers on wisdm dataset for human activity recognition. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, pages 1–7, 2016.

[113] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018.

[114] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On Calibration of Modern Neural Networks. *CoRR*, abs/1706.0, 2017.

[115] Francesco Daghero, Daniele Jahier Pagliari, and Massimo Poncino. Two-stage human activity recognition on microcontrollers with decision trees and cnns. In *2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pages 173–176, 2022.

[116] Francesco Daghero, Alessio Burrello, Daniele Jahier Pagliari, Luca Benini, Enrico Macii, and Massimo Poncino. Energy-efficient adaptive machine learning on iot end-nodes with class-dependent confidence. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2020.

[117] 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. *CoRR*, abs/1704.04861, 2017.