POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Neural Network Splitter: Optimal Decomposition of a Neural Network and its Distribution on Multiple Microcontrollers

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica


**Andrea Santamaria, 10523667**

**Advisor:**
Prof. Marco Marcon

**Co-advisors:**
Biagio Montaruli
(STMicroelectronics)
Danilo Pietro Pau
(STMicroelectronics)



**Academic year:**
2020-2021



**Abstract:** The deployment of Neural Network (NN) models on low-power and resource-constrained devices represents a critical bottleneck in the development of intelligent and autonomous Internet of Things (IoT) systems due to the aggressive computational and memory constraints. For this reason, Machine Learning (ML) solutions addressing tiny devices must be designed having in mind constraints on memory and processing capabilities characterizing such devices. In this thesis, we introduce a novel design methodology based on a distributed approach, which aims at automatically partitioning the execution of a NN over multiple heterogeneous tiny devices. Such a methodology is formalized as an optimization problem where either the inference latency is minimized or the throughput is maximized, within the devices' memory and computing capabilities. The methodology is evaluated over different NN architectures and microcontrollers (MCUs) by using three algorithms, namely Full Search (FS), Dichotomic Search (DS), and Branch-and-Bound (B&B). The obtained results showed that the B&B outperformed the others as it was able to find the optimal solution in the lowest number of computing steps in all the experiments. With this work, we aim at enabling novel ML solutions that offer low decision-latency, autonomy, and high energy efficiency.



**Key-words:** Machine Learning, Neural Networks, Distributed Systems, Optimization Problem, Microcontrollers


## 1. Introduction

In recent years, Deep Learning (DL) has achieved great success and state-of-the-art performance in numerous applications such as computer vision and natural language processing. The *TinyML* community [1–3] is set to provide a unique opportunity for integrating ML into low-power (mW range and below) resource-constrained devices, such as sensors and MCUs. Indeed, the opportunity of deploying NN models as close as possible to the sensing device to process acquired data is the key to develop autonomous IoT systems, and opens up to many benefits for several real-world embedded applications. Among those, there is the reduction in terms of latency and communication bandwidth [4], as well as the possibility of enhancing user data privacy [5].
However, the deployment of accurate NN models on tiny devices represents a critical challenge due to the aggressive computational and memory constraints. This problem has been addressed by the *TinyML* research and industry communities using different approximation and optimization techniques, such as parameter pruning and sharing, quantization, and knowledge distillation [4, 6]. Unfortunately, these approaches often require to

re-design or modify the models' topology to be deployed, which requires significant effort and time and might imply a reduction in terms of accuracy. An alternative approach consists in adopting a distributed computing paradigm by partitioning a NN model into sub-models to be deployed on several tiny devices, which could be heterogeneous in their properties. This would also opens up to additional advantages such as improving scalability, more productive deployment, and the possibility of distributed learning from a wide and differentiated amount of data [5]. Moreover, it is worth noting that to further reduce the memory and computing footprint, this distributed approach can be jointly combined with the state-of-the-art compression techniques [5, 7].

## 1.1. Thesis Topic

In this context, this work proposes the *Neural Network Splitter*, a software tool whose goal is to automatically distribute a given pre-trained NN model on multiple heterogeneous tiny devices in order to optimize an objective function, while satisfying the memory and computational constraints of the devices themselves as well as preserving the model's topology and accuracy.
With respect to the literature, the novelties of this work can be summarized as follows:

- the methodology adopted to solve the problem, which takes into account the devices' memory and computational capabilities, as well as the communication requirements;
- a detailed mathematical formulation for the problem definition.

To validate the proposed methodology an extensive experimental campaign has been carried out, considering several heterogeneous NN architectures and multiple MCUs.

## 1.2. Structure

The thesis can be divided into four major blocks. The first one, presented in Section 2, illustrates the related works in the literature. In particular, we focused on papers that either rely on the distributed approach or deal with approximation techniques, summarizing the positive aspects as well as the negative ones for each work.
The second block defines the problem by introducing the mathematical formulation adopted to model it and its assumptions. It is developed in Section 3 and, at the end of it, we show through a counterexample how correctly defining the quantities to be optimized (i.e., throughput) is crucial, otherwise, the obtained solution might not be relevant.
Section 4 explains in details the algorithms developed to solve the problem and offers an overview of the performance of each algorithm in terms of time complexity, number of steps to find the solution, and if they guarantee the optimum. Finally, in the last part of the thesis, composed of Sections 5, 6, and 7, the results of the experimental campaign are described in details, showing both the strengths and the limitations of the algorithms. We also present the performance evaluation of a practical case of the NN deployment over few devices comparing it with the theoretical estimates. In the end, we carry out a summary of the obtained results, along with possible future extensions.

# 2. Related Literature

The problem of distributing a NN over several embedded devices has been addressed from several points of view and different complementary approaches have been proposed.
The authors of [4] proposed a methodology formalized as a quadratic optimization problem aiming at distributing some Convolutional Neural Networks (CNNs) over several IoT devices by minimizing the "data production to decision making"-latency.
[8] presented a framework called DeepThings that allows to partition the execution of CNN layers vertically in a grid fashion, thus resulting into independently parallelizable tasks that can be distributed among several devices. This approach allows to minimize the memory footprint by reducing the sizes of input and output activations. However, the main drawback is the replication of the network's parameters on all the devices, thus increasing the memory footprint at system level.
[9] presented a framework called *Pipe-it* whose goal was to partition CNN layers across clusters while limiting parallelization of their respective kernels to the assigned cluster. However, one of the main limitations of this approach consisted in the strong assumption about the network topology. In particular, it assumed as reference models CNNs having initial convolutional layers that are more compute-intensive than the other layers in the remaining part of the pipeline.
[6, 7] introduced *Network of Neural Networks*, a communication-aware distributed learning framework based on Knowledge Distillation to compress a large pre-trained Deep Neural Network (DNN) into several independent smaller networks that fit the memory and performance budgets of the devices and result in a negligible loss in

terms of accuracy. A further improvements to this work could be to evaluate the solution on tiny low-power MCUs since the authors have carried out the experimental evaluation on more powerful Edge devices (Raspberry Pi-3 and Odroid-XU4S boards).
[5] introduced a framework for distributing the CNNs computation between the Edge and the Cloud with the possibility of classifying samples at early exit points in a CNN using a confidence level threshold. The major limitation of the solution proposed by the paper consists in the predominant usage of Cloud which may significantly impact the communication latency and expose the deployed models to privacy issues. Moreover, using heterogeneous end-devices is not taken into consideration (having different requirements in terms of memory and/or computational power could imply different exit points).
A different point of view consists in reducing the computational and memory demand of DL solutions to match the technological constraints of IoT systems and it is provided by several approximation techniques. Such approaches allow to reduce the memory and/or computational demand of DL solutions at the expense of a decrease in the accuracy. [10] proposed layers' weights quantization which can be done also during the model training phase [11–13]. Compression techniques such as compressing the weights [14], pruning whole or part of layers [15] or adopting Huffman coding [16] could be a way to fit memory constraints. Another approach is reducing inference time on pooling and normalization layers [17]. Gate-Classification CNNs [18] and Adaptive Early Exit CNNs [19] showed that the classification output can be provided at intermediate layers, skipping the remaining computation.
In summary, even though all the above related works offer very interesting solutions and methodologies, almost all these solutions do not use the original NN architecture, but rely on a modified version to match the hardware and physical constraints of the devices and they are not evaluated on low-power tiny devices, which on the other hand, is the main focus of this work.

# 3. Problem Definition and Assumptions

The problem addressed by this work is how to optimally distribute the execution of a pre-trained NN on multiple heterogeneous tiny devices such as MCUs and, at the same time, preserving the model architecture and its level of accuracy while satisfying the MCUs' memory and computational constraints. This problem can be modeled as an Optimization Problem (OP) and two objective functions have been taken into account:

- minimization of the total inference latency;
- maximization of the throughput.

The solution of the problem has to both satisfy the technological constrained imposed by the IoT system, as well as preserve the behavior of the original model, that is the output generated by the original model given an input must be the same of the output generated by the sub-models executed in-order fashion considering the same input sample. Despite the formalization is general enough to work with any NN model, we assumed to use sequential CNNs (where inputs are processed only in the forward direction like in Feed-Forward Neural Networks [20]), because dealing with branches is still an open point. A trivial solution to manage multi-branches NNs is to group together the root node and all the layers inside the branches as a single layer (*"super-node"*) in order to obtain a sequential network. The main drawback of this approach is that a generated *"super-node"* might not fit the memory and/or computational requirements of any device.

## 3.1. Mathematical Formulation

Let a CNN with $n$ layers to be deployed on $d$ heterogeneous devices. Without loss of generality, we can represent the CNN as a computational graph [21] where the nodes are the layers, while the edges correspond to the input/output tensors of a layer. The goal is to assign each layer to a MCU in order to optimize the chosen objective function without violating the constraints. Assuming to assign a sequential number to each layer in order to sort them in topological order, let us define the layers set $N = \{1, \ldots, n\}$. Moreover, let us also define $D = \{1, \ldots, d\}$ as the set of the available MCU devices (each one represented by a unique ID). Therefore, there are $d^n$ candidate solutions in total, but not all of them might be feasible (that is they satisfy the constraints). Let $P_p = \{(layer_1, dev_1), \ldots, (layer_n, dev_d)\}$ for $p = 1, \ldots, d^n$ a candidate solution where each tuple is a layer-device assignment and let $P = \{P_1, \ldots, P_{d^n}\}$ be the candidates' set. Given the solution of the problem, let us define the set of sub-models, where each sub-model is obtained by grouping together all the consecutive layers assigned to the same device. Moreover, by enumerating the sub-models by topological order, let us define $M = \{1, \ldots, m\}$ as the set of sub-models, where $1 \leq \mathbf{card}(M) \leq n$. Let $N^{(m)} = \{n^m_{first}, \ldots, n^m_{last}\}$ $\forall m \in M$, a partition of $N$, be the set of the consecutive layers belonging to the same $m^{\text{th}}$ sub-model. Similarly, let $M^{(i)} = \{m^i_{first}, \ldots, m^i_{last}\}$ $\forall i \in D$, a partition of $M$, be the set of the sub-models (in topological order) processed by the $i^{\text{th}}$ device. It is worth noting that a device can be assigned to one or more non-consecutive layers, which means that it can process one or more sub-models.

In the proposed OP, let us characterize each layer with three properties: FLASH memory size, RAM memory size, and number of Multiply-Accumulate (MAC) operations. As for the devices, they are abstracted through their memory properties, namely FLASH and embedded RAM memory sizes (which are usually limited up to few Mbytes and up to hundreds of Kbytes, respectively), and performance properties, i.e. operating CPU clock frequency (CPUFREQ) and cycles per MAC (CpM).
The memory properties are used to define the following (global) constraints:

$$\sum_{j \in N} FLASH_j \leq \sum_{i \in D} FLASH_i \tag{1}$$

$$\max_{j \in N} FLASH_j \leq \max_{i \in D} FLASH_i \tag{2}$$

$$\max_{j \in N} RAM_j \leq \max_{i \in D} RAM_i \tag{3}$$

Eq. (1) states that the total FLASH size of the original network, computed as the sum of the FLASH size of each layer, must be less or equal than sum of the available MCUs' FLASH size. While (2) and (3) impose that the layer with the highest FLASH/RAM size cannot exceed the memory resources of the least constrained MCU, which basically means that there must be at least one device on which to deploy the most memory demanding layer. If one of the above is violated, then the CNN can not be deployed on the available devices.
Moreover, in order to deploy the sub-models on the corresponding devices, the following additional (local) constraints must be satisfied:

$$\sum_{m \in M^{(i)}} \sum_{j \in N^{(m)}} FLASH_j \leq FLASH_i \qquad \forall i \in D \tag{4}$$

$$\max_{m \in M^{(i)}} \max_{j \in N^{(m)}} RAM_j \leq RAM_i \qquad \forall i \in D \tag{5}$$

These constraints allow a candidate solution to be feasible in the sense that for each device $i$, its FLASH (RAM) memory size has to be always greater or equal than the total FLASH (maximum RAM) size required to store all the sub-models assigned to it. It is worth noting that the RAM memory is not additive, which means that in a CNN only the layer with the maximum RAM size represents the bottleneck.
On the other hand, the performance properties are used in (6) to define the *layer computational latency* as the time (in seconds) required to process a layer $j$ by a device $i$.

$$L^{ij} = \begin{cases} 0 & \text{if } j \text{ not assigned to } i \\ \frac{MAC_j \; CpM_i}{CPUFREQ_i} & \text{otherwise} \end{cases} \qquad \forall i \in D, \forall j \in N \tag{6}$$

Similarly, let us define in (7) the *sub-model computational latency* as the time (in seconds) required to process a sub-model $m$ by a device $i$:

$$L^i_m = \sum_{j \in N^{(m)}} L^{ij} \qquad \forall i \in D, \forall m \in M \tag{7}$$

and define in (8) the *device computational latency* as the time (in seconds) that a device $i$ spends to process all the sub-models assigned to it:

$$L^i = \sum_{m \in M^{(i)}} L^i_m \qquad \forall i \in D \tag{8}$$

Without loss of generality, let us identify the unique device with the highest computational latency, as reported in (9), because this will be useful later on.

$$\bar{i} = \arg\max_{i \in D} L^i \tag{9}$$

Moreover, it is possible to define the *total computational latency* as the sum of the latencies defined in (8) over all the devices.

$$L = \sum_{i \in D} L^i \tag{10}$$

Now, another important aspect to introduce is the time required by two devices to transfer data, since this quantity is present in the formula to compute the total inference latency. Let us define the *communication latency* as the time (in seconds) required by a device $i$ to send a certain amount of bytes to a device $h$ at a previously selected baud rate. Here the latency could be equal to zero when there is no pair of layers assigned

to them (one assigned to $i$ and one to $h$) such that it contains two consecutive layers, which means that there is no communication between the devices.

$$T^{ih} = \begin{cases} 0 & \text{if } i, h \text{ do not communicate} \\ \frac{\mathit{Bytes\ to\ transfer}}{\mathit{baud\ rate}} & \text{otherwise} \end{cases} \qquad \forall i, h \in D \tag{11}$$

Then, is it possible to define in (12) the *total communication latency* as the sum of all the communication latencies:

$$T = \sum_{i,h \in D} T^{ih} \tag{12}$$

Finally, let us define in (13) the *total inference latency* as the sum of the *total computational latency* and the *total communication latency*. This quantity represents the time (in seconds) needed to perform an entire inference taking into account the communication time to transfer data from one device to another, and this shall be minimized when adopting the first policy to solve the problem.

$$I = L + T \tag{13}$$

The other policy we considered in our work to solve the OP is the maximization of the throughput, which is basically the inverse of the waiting time to process the next input when having multiple input sample to process one at a time. Hence, the waiting time will be minimized when the workload in terms of computational latency among the devices is equally balanced, and that's the reason to have previously introduced the device $\bar{i}$ with the highest computational latency. If all the devices have the same *device computational latency* and only one sub-model is assigned to each device, then the throughput achieves the highest possible value and there is a significant speed up in the processing of the data since each device can immediately process a new input of the next run once its task is finished regardless if the current run is not finished yet.
First of all, let us define in (14) the *communication latency* as the sum of the times (in seconds) required by the device $\bar{i}$ to transfer its output (receive its input) data to (from) a generic device $h$.

$$T_{\bar{i}} = \sum_{h \in D} T^{\bar{i}h} \tag{14}$$

Furthermore, is important to introduce a new set which contains specific sub-models that are not assigned to $\bar{i}$ because it will simplify the notation. For instance, let us assume to split an ideal network in three sub-models (in order A, B and C) between two devices (D1 and D2) with this schedule: sub-models A and C assigned to D1 and sub-model B assigned to D2. Let D1 be the device with the highest computational latency. Thus, we called *"intermediate"* layers those layers that are not assigned to $\bar{i}$ in the schedule, but their processing must be done between the first and the last sub-models assigned to $\bar{i}$, which means in this example that the *"intermediate"* layers are all the layers of the sub-model B (see Figure 1).

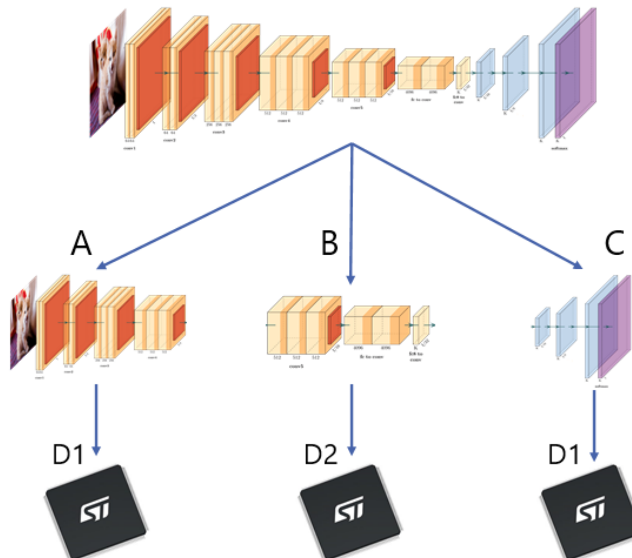



Figure 1: Example of a CNN partitioning.

So, the new set introduced in (15) contains all the sub-models made of *"intermediate"* layers, where the *range(a,b)* function generates the sequence of numbers starting from the given start integer $a$ to the stop

integer $b$.

$$R^{(\bar{i})} = range(m^{\bar{i}}_{first}, m^{\bar{i}}_{last}) \setminus M^{(\bar{i})} \tag{15}$$

The last term to insert in the mathematical formulation before defining the throughput is the time (in seconds) needed to process the *"intermediate"* layers defined as the sum of each *sub-model computational latency* that belongs to (15) all over the devices different from $\bar{i}$.

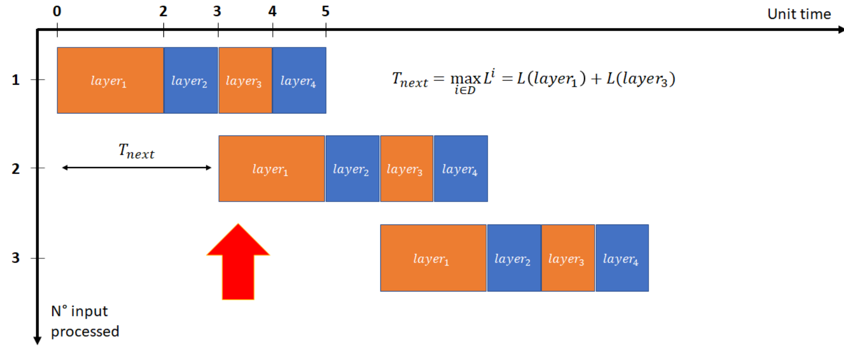
$$L^* = \sum_{i \in D \setminus \{\bar{i}\}} \sum_{m \in R^{(\bar{i})}} L^i_m \tag{16}$$

Finally, we have all the ingredients to introduce the throughput, stated in (17), as the inverse of the waiting time to process the next input.

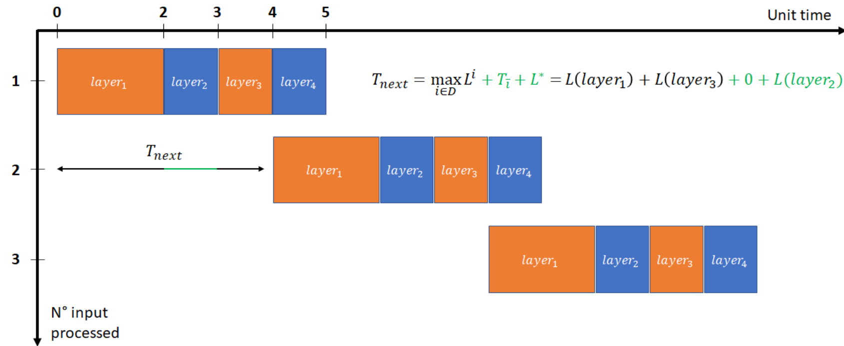
$$throughput = \frac{1}{waiting\ time} = \frac{1}{\max_{i \in D} L^i + T_{\bar{i}} + L^*} \tag{17}$$

## 3.2. Improvement of the Waiting Time Definition

In our work we improved the definition of the waiting time with respect to the one presented in [9], which is defined as maximum device computational latency, that is $\max_{i \in D} L^i$ using our notation, plus the time to transmit the data. It is worth noting that if using the definition described in [9], it may happen that after $n$ inputs a device has to simultaneously process two layers belonging to two different inferences (*overlapping problem*). For example, as shown in Figure 2, given a 4-layers CNN and its optimal partition on two arbitrary MCUs (orange and blue, where the orange device is the one with the highest computational latency), assuming communication time equals to zero, there will be a certain time in which the orange device has to simultaneously process the third and the first layer of two different inferences (see Figure 2a). Assuming the MCUs can process only one layer at a time, this will lead to an additional delay in the run in order to execute sequentially the two layers. Figure 2b shows instead the scheduling with no conflicts when throughput is defined using (17). The key term added in the initial formulation is the processing time $L^*$ of the layers ($layer_2$ in the example below) that have not been assigned to the device with the highest computational latency $\bar{i}$, and fall inside the range of the layers (sorted in topological order) assigned to $\bar{i}$.




(a) Overlapping




(b) Correct scheduling

Figure 2: Schedule obtained using the two definitions of the throughput, 2a uses definition in [9], while 2b uses (17).

# 4. Proposed Work

To solve the OP, three algorithms have been used, namely FS, DS, and B&B, which are described in the follow and compared together in Table 1 in terms of time complexity, number of steps to find the solution, and optimality.

## 4.1. The Full Search Algorithm

This is the basic approach among the others and consists in exploring all the candidate solutions $P_p$ defined in Subsection 3.1, one after the other, by checking at each step whether the current candidate is feasible and is better than the best solution found so far. If so, that candidate becomes the current best solution. This algorithm always guarantees to find the optimal solution since it evaluates all the possible candidate solutions. However, its drawback is the exponential complexity in the number of layers, which implies that it is impractical when dealing with very deep CNNs.
Three functions compose the pseudo-code reported in Algorithm 1: *new_configuration()*, which takes in input the current iteration number $p$, the set $N$ of the CNN layers, and the set $D$ of the available MCUs and returns the unique candidate solution $P_p$ related to the current iteration number (since there exists a bijection between integer numbers and the $d^n$ possible configuration for a candidate). The function *check_feasibility()*, by taking in input the candidate $P_p$, checks if it violates any constraints described in Subsection 3.1 and returns *True* or *False*. In the end, the function *better_solution()* evaluates the current configuration $P_p$ with respect to the stored solution $P_{\overline{p}}$ based on the chosen objective function $f$. If the candidate $P_p$ is better in terms of maximizing the throughput or minimizing the total inference latency, then the solution is updated.
FS emerged to be useful with shallow CNNs, regardless of the number of available MCUs.

**Algorithm 1** Full Search Algorithm

**Input:** $N, D, f$
**Output:** optimal $P_{\overline{p}}$
*Initialization:* $P_{\overline{p}} = \emptyset$

```
1: for p ∈ [1, d^n] do
2:   P_p ← new_configuration(p, N, D)
3:   if check_feasibility(P_p) then
4:     if better_solution(P_p, P_p̄, f) then
5:       P_p̄ ← P_p
6:     end if
7:   end if
8: end for
9: return P_p̄
```

## 4.2. The Dichotomic Search Algorithm

Adopting the DS was another way to solve the problem. The DS is a recursive algorithm that produces a bisection tree, which is explored in a depth-first search (DFS) fashion, which means that the algorithm starts at the root node and explores as far as possible along each branch before backtracking [22]. The DS algorithm starts from an initial candidate solution (*root node*) that assigns all the layers to the same MCU. It is worth noting that the root node may not be a feasible configuration and, in that case, it just represents a starting point for the tree exploration. New candidate solutions (*child nodes*) are generated by assigning $(n/2)^t$ consecutive layers in the parent node to a different device, where $n$ is the number of layers and $t$ is the depth level of the node. In case the CNN has an odd number of layers, we assumed to round up the fraction that counts how many layers have to change their assignment. Moreover, for each new candidate, the DS checks whether it is feasible and better than the best solution found so far. If so, it updates the current best solution with that candidate. With respect to FS, DS has linear complexity in the number of layers, but due to the way it generates new child nodes it does not guarantee to find the optimum.
Algorithm 2 reflects the DS's implementation: for each device $dev$ in the set $D$, the function *initial_configuration()* is called to create the root node that assigns all the layers to $dev$. Then, the function *recursion()*, which takes the current candidate $P_p$, the solution $P_{\overline{p}}$ found so far, and the chosen objective function $f$ as inputs, is applied to generate and explore the tree. This function is detailed in Algorithm 3: it first checks potential constraints violations and saves the current pipeline $P_p$ if better than the stored solution with the functions *check_feasibility()*

and *better_ solution()*, respectively. In particular, under certain circumstances such as having a number of CNN layers which is not a power of 2, or when the number of layers assigned to the device *dev* is odd, these two functions not only evaluate the candidate $P_p$, but also its "reversed" version in which the list of the devices that appears in the assignment is put in the reverse order. This procedure aims to find few candidate solutions which cannot be generate otherwise and produces an increase in the number of explored nodes. Then, either the algorithm reaches the exit condition from the recursion loop if the function *is_ leaf()* returns *True*, or it generates a set $C$ of new child nodes through the function *create_ child_ nodes()* and continues the exploration. Both the functions take in input the current candidate $P_p$ and the device *dev*. The former returns *True* if and only if it runs into a candidate with only one layer assigned to *dev* (*leaf node*), while the latter generates new child nodes with the procedure described above and, for each one, the recursion loop starts again.
Even if DS does not guarantee to find the optimal solution, achieving linear complexity in the number of layers could help to deal with very deep CNNs, which are too time-consuming to be analyzed by FS.

**Algorithm 2** Dichotomic Search Algorithm

**Input:** $N, D, f$
**Output:** $optimal P_{\overline{p}}$
*Initialization:* $P_{\overline{p}} = \emptyset$

```
1: for dev ∈ D do
2:     P_p ← initial_configuration(dev, N)
3:     recursion(P_p, P_p̄, f, dev)
4: end for
5: return P_p̄
```

**Algorithm 3** function *recursion* pseudo-code

**Input:** $P_p, P_{\overline{p}}, f, dev$

```
1: if check_feasibility(P_p) then
2:     if better_solution(P_p, P_p̄, f) then
3:         P_p̄ ← P_p
4:     end if
5: end if
6: if is_leaf(P_p, dev) then
7:     return
8: else
9:     C ← create_child_nodes(P_p, dev)
10:    for c ∈ C do
11:        recursion(c, P_p̄, f, dev)
12:    end for
13: end if
14: return
```

## 4.3. The Branch-and-Bound Algorithm

The B&B algorithm is a general search algorithm for finding an optimal solution and relies on the availability of good heuristics for estimating the *best* values (*"best"* according to the optimization function) of all the leaves under the current branch of the search tree. In order solve our problem using this algorithm, as described in [23], we modeled the problem introduced in Section 3 as a Constraint Satisfaction Optimization Problems (CSOP), since all optimization problems studied in operation research are Constraint Satisfaction Problems (CSPs) in the general sense, where the constraints are normally numerical. In this context, the CNN layers corresponds to the variables of the CSOP, while the domain of each variable is the set of available devices.
Moreover, when using the B&B it is important to define how to compute the *f-value* and to choose an admissible heuristic to estimate the *h-value* for every feasible assignment of some variables. By saying that the chosen heuristic must be admissible, as explained in [23], it means that the *h-value* must be an upper (lower) bound for the *f-value* in a maximization (minimization) problem.
In our context, we decided to apply the B&B algorithm to solve only the latency minimization problem. In particular, the *f-value* is computed, when a new candidate solution is found, by using the formula of the

*total inference latency* defined in (13), while the *h-value* is computed as the sum between the partial *layer computational latency* (computed by taking into account only the variables assigned so far) and the estimated remaining latency (computed, for each unassigned variable, as the sum of the minimum *layer computational latency* (6) with respect to the devices). It is worth noting that, this heuristic is admissible because it returns an underestimation of any f-values. In fact, it does not consider the communication latency for the unassigned variables.
As for the throughput maximization problem, designing an admissible heuristic to estimate the *h-value* is a difficult task due to the computation of the throughput itself. Indeed, selecting a value for the unassigned variables in order to find an upper bound of the *f-value* actually becomes a scheduling sub-problem inside the CSOP we want to solve, and that's the reason we preferred to apply the B&B algorithm to solve only the latency minimization problem.
Finally, even though the time complexity of the B&B is exponential in the number of layers because its worst-case scenario takes $\frac{d^{n+1}-d}{d-1}$ iterations to explores all the tree, in practice, it can be significantly reduced by the pruning mechanism adopted by the B&B during the exploration as well as by using additional heuristics for selecting the next variable and sorting the set of values to be assigned [24, 25]. In particular, we developed seven proprietary heuristics to select the next layer as well as seven proprietary heuristics to sort the device's set. We decided to anonymize these heuristics by naming them *Method 1, Method 2, ... Method 7* in order to protect STMicroelectronics' intellectual properties. Furthermore, we added to this list a device heuristic, called *Arbitrary*, which allows to choose a value from the domain by hand (it could be chosen at random if not specified) and three well-known strategies explained in details in [26]:

- *Degree heuristic* (DH): it selects the variable that is involved in the largest number of constraints on other unassigned variables (layer heuristic);
- *Minimum Remaining Values* (MRV or *"fail-first"*): it selects the most costrained variable, namely the one with the fewest "legal" remaining values in its domain (layer heuristic);
- *Least constraining value* (LCV): it prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph (device heuristic).

By construction, this algorithm always guarantees to find the optimal solution. For this reason, the B&B algorithm can be considered as the best trade-off between the FS (it always find the optimal solution but has exponential time complexity) and DS (finding the optimal solution is not guaranteed but has linear time complexity). The flowchart of the B&B algorithm is depicted below in Figure 3.

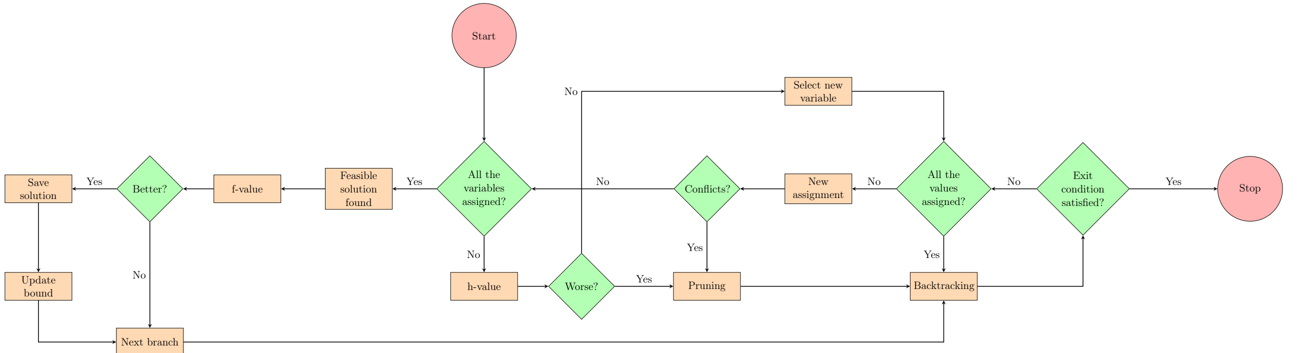



Figure 3: Flowchart of the B&B algorithm.

Table 1: Algorithms comparison.

| | **Full Search** | **Dichotomic Search** | **Branch-and-Bound** |
|---|---|---|---|
| *Time Complexity* | $O(e^n)$ | $O(n)$ | $O(e^n)$ |
| *Explored nodes* | $d^n$ | $\beta n$ | $\leq \frac{d(d^n-1)}{d-1}$ |
| *Optimality* | Yes | Not guaranteed | Yes |

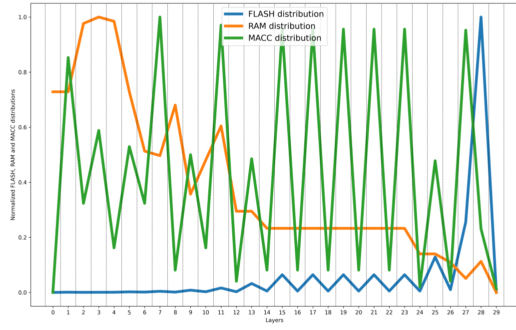
## 5. Experimental Results

To evaluate the FS, the DS, and the B&B algorithms presented in Section 4, we carried out a detailed experimental campaign considering eight CNNs models (whose properties are summarized in Table 3) and ten STM32 MCUs (see Table 2) characterized by heterogeneous memory and computational properties.
As for the CNN models, we used three MobileNets v1 [27] which take as input a $128 \times 128$ RGB image, using different values for the $\alpha$ parameter to control the width of the network (width multiplier): 0.25, 0.30, and 0.35. YAMNet 256 is a modified version of the YAMNet[1] model obtained by taking the first six convolution blocks of the original model, while VoxCeleb is a proprietary model trained on the VoxCeleb dataset [28] and consists of four convolution blocks (made by a convolution using ReLu non-linearity followed by a max pooling layer) followed by a separable convolution block (made by a separable convolution followed by a global average pooling and a dropout layer), and two fully-connected layers having 128 and 256 output neurons, respectively. Then, we created a small CNN, named Tiny-CNN and trained on the MNIST dataset [29, 30], whose architecture consists in three convolution blocks (like VoxCeleb's ones) followed by a fully-connected layer of 48 output neurons. Finally, we also considered the CNN and DS-CNN models introduced in [31] for keyword spotting, with the only difference that in this work the weights and activations are in *float32* format. For more details, see the Appendix A where we reported a list of tables, where each row is a layer of the CNN and each column represents a feature to describe the layer. Moreover, CNNs' complexity profiles are drawn in Figure 4.
The data reported in Table 3 has been obtained using the X-CUBE-AI tool v 7.1.0 [32, 33]. It is worth noting that the second column (*Depth*) contains the number of layers of the *optimized model* generated through X-CUBE-AI, while the next columns (*FLASH, RAM*, and *MAC*) contains the sum of the FLASH memory sizes, the max of the RAM memory sizes, and the sum of the MACs. In simple terms, the *optimized model* is an oriented graph that collects all the necessary information related to the model (*i.e.* input/output tensors and shapes, memory demand, name and type for each layer, etc.) and is obtained through a surjective map that potentially merges consecutive layers of the given CNN into a single node.
To handle data communication between partitioned sub-models deployed on different MCUs, a mesh network topology was assumed and the UART transmission protocol (baud rate set to 115200 bps, asynchronous mode, one stop bit, eight data bits and no parity bits) was used.
In the experimental evaluation, both the optimization policies described in Subsection 3.1 have been used. In particular, as for the minimization of the inference latency, we used all the algorithm described in Section 4, while regarding the maximization of the throughput, we considered only FS and DS. Tables 4 and 5 report a summary of the obtained results, which are further analyzed in the follow.

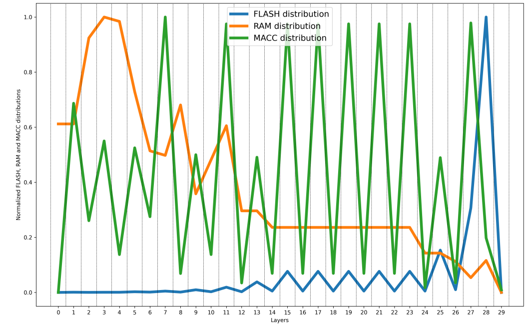
Table 2: MCUs memory and computing properties.

| MCU | FLASH (KB) | RAM (KB) | CPU Freq. (MHz) | CpM |
|---|---|---|---|---|
| STM32H743ZI [34] | 2048 | 1024 | 480 | 6 |
| STM32H723ZG [35] | 1024 | 564 | 550 | 6 |
| STM32F446RE [36] | 512 | 128 | 180 | 9 |
| STM32F401RE [37] | 512 | 96 | 84 | 9 |
| STM32F401RB [38] | 128 | 64 | 84 | 9 |
| STM32L4R5ZI [39] | 2048 | 640 | 120 | 9 |
| STM32L452RE [40] | 512 | 128 | 80 | 9 |
| STM32L433RC [41] | 256 | 64 | 80 | 9 |
| STM32L412KB [42] | 128 | 40 | 80 | 9 |
| STM32G071RB [43] | 128 | 36 | 64 | 307 |

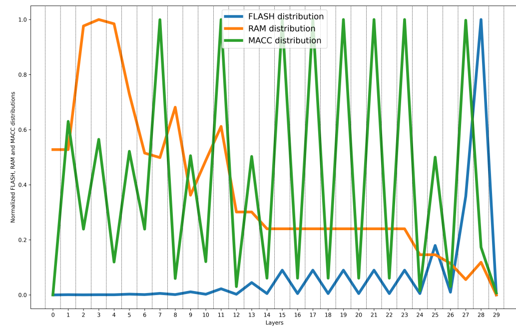
[1]YAMNet source code and related tutoral available at `https://www.tensorflow.org/hub/tutorials/yamnet` (Accessed on $28^{th}$ February 2022)



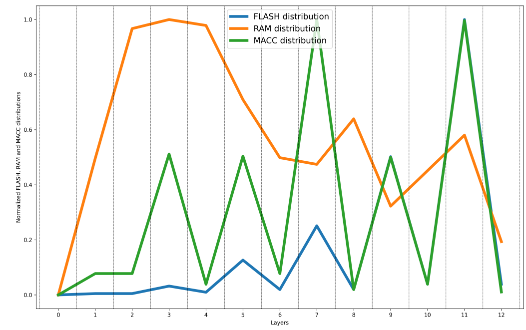

(a) MobileNet v1 025



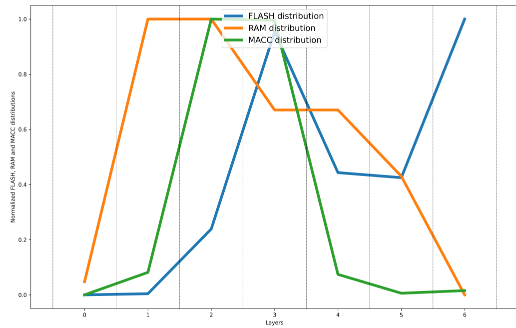

(b) MobileNet v1 030



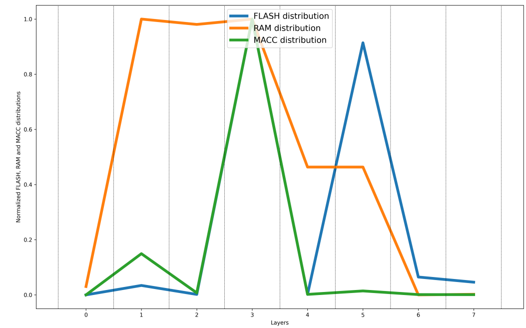

(c) MobileNet v1 035



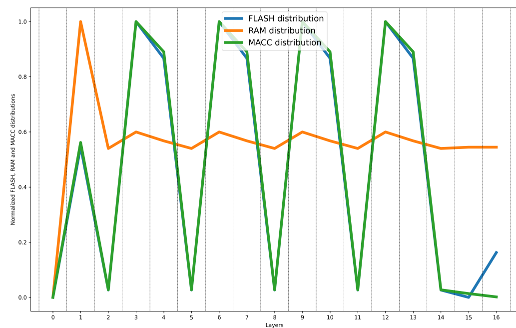

(d) YAMNet 256



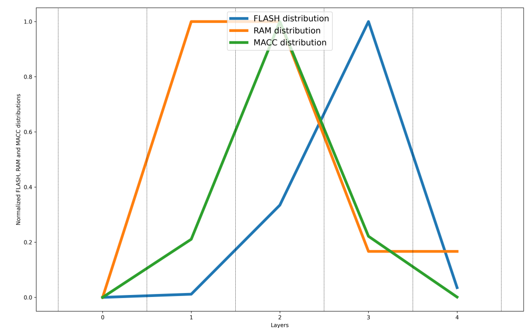

(e) VoxCeleb




(f) KWS CNN




(g) KWS DS-CNN




(h) Tiny-CNN

Figure 4: FLASH (blue), RAM (orange), and MACC (green) normalized profiles of the CNN models.

Table 3: CNN models used in the experimental evaluation.

| Model | Depth | Tot FLASH (KB) | Max RAM (KB) | Tot MAC ($10^6$) |
|---|---|---|---|---|
| MobileNet v1 025 | 30 | 1825.53 | 262.06 | 14.4 |
| MobileNet v1 030 | 30 | 2366.15 | 311.32 | 19.6 |
| MobileNet v1 035 | 30 | 2976.80 | 360.34 | 26.0 |
| YAMNet 256 | 13 | 526.25 | 396.25 | 24.4 |
| VoxCeleb | 7 | 926.84 | 39.75 | 12.1 |
| KWS CNN | 8 | 270.91 | 31.21 | 2.53 |
| KWS DS-CNN | 17 | 155.80 | 56.25 | 4.83 |
| Tiny-CNN | 5 | 74.85 | 11.31 | 0.81 |

## 5.1. Evaluation of MobileNet v1 025

This CNN model has been evaluated on the STM32H743ZI and STM32L4R5ZI MCUs, and for both of them the FLASH memory was set to 75% of the available one in order to potentially split the network in at least two parts. Concerning the maximization of the throughput, DS was successfully able to obtain the optimal solution in 236 iterations (0.000022% w.r.t FS). The original model has been split into two sub-models, one consisting of the last two layers of the original model and was assigned to the STM32L4R5ZI, while the other sub-model (consisting of all the other layers) was assigned to the more powerful STM32H743ZI. The estimated throuhput was 4.034 $s^{-1}$ using our innovative formula and, in this case, it coincides with the other definition since the set defined in (15) was empty and consequently (16) was zero.
As for the other optimization policy, namely the minimization of the total inference latency, all the algorithms (FS, DS and B&B) provided the same optimal solution previously found and the estimated total latency was 0.268 $s$. In addition to this result, we want to underline the fact that the estimated throughput coincides with the inverse of just a part of the total inference latency, that is the time required by the STM32H743ZI to process the first sub-model plus the time needed to transmit its output to the other device. Here, the DS took again 236 iterations to find the optimum, while the B&B ranged between 66 and 439857 explored nodes, depending on which pair of heuristics is chosen to order the layers and the devices. Figure 5 shows that two of our methods were the worst heuristics to organize the devices, while *DH, MRV* and *Method 1* were the layers heuristics that achieved the minimum number of iterations. By analyzing the result in more detail according to the best heuristics, we found that they actually proposed the same ordering for the variables. This means that by following the ordering of the heuristics presented in literature produces, in this case, the best result in a tremendously lower number of iteration w.r.t FS and DS.

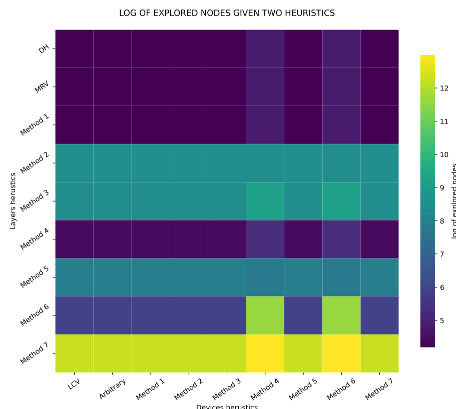



Figure 5: B&B heatmap for MobileNet v1 025.

## 5.2. Evaluation of MobileNet v1 030

For this test, the STM32H743ZI and the STM32F401RE MCUs were used as target devices at full capacities. For both the optimization policies, the optimal solution consisted in three sub-models: the first and the last sub-models were assigned to the STM32H743ZI, while the second one (consisting of just a single layer) was processed by the less powerful STM32F401RE. The estimated inference latency was 1.839 $s$, while the estimated throughput was 0.544 $s^{-1}$. It is worth noting that, the throughput estimated with the formula used in [9] was 0.629 $s^{-1}$. However, in a real-world application after 1.59 $s$ from the beginning of the first inference there would be an overlapping issue (see Section 3) in the schedule: the STM32H743ZI would have to process the last sub-model and simultaneously the first sub-model of following inference.
Because of the number of layers was not different from the MobileNet v1 025, FS and DS took the same amount of steps of the previous test to reach optimality. On the other hand, B&B presented a slightly difference in terms of iterations as shown in the heatmap in Figure 6: the *Method 2* for ordering the devices does not find the optimal solution with the minimum number of iterations. Again, by maintaining the ordering provided by *DH, MRV* or *Method 1* for layers, the optimal solution was achieved in 62 steps when combined with *Methods 1, 3, 5* or *7* for ordering the devices. In the worst-case scenario, the B&B algorithm explored 15769 nodes of the tree (0.0015% w.r.t FS) before finding the optimal solution, which is a clear proof of how efficient pruning techniques are even without choosing the best heuristics.

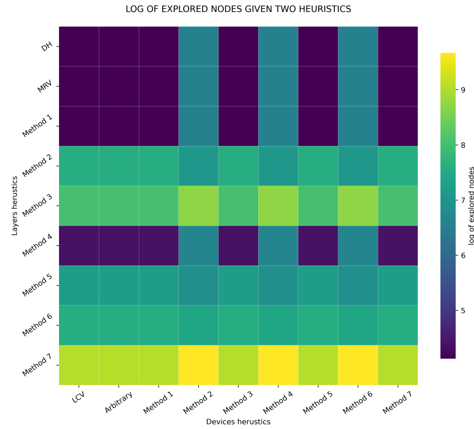



Figure 6: B&B heatmap for MobileNet v1 030.

## 5.3. Evaluation of MobileNet v1 035

To evaluate this model, STM32H743ZI and STM32L4R5ZI were used, as for the MobileNet v1 025, but in this case without restricting the size of the FLASH memories. The optimal solution was the same w.r.t the evaluation of MobileNet v1 025 for both the optimization policies: a single split that that assigned the first sub-model to the more powerful STM32H743ZI and the ending part of the model, made of two layers, assigned to the STM32L4R5ZI. The obtained inference latency and throughput were 0.448 $s$ and 2.379 $s^{-1}$, respectively. Clearly we obtained a higher inference latency due to the more computational efforts required by the network w.r.t the MobileNet v1 025 (higher FLASH memroy size, RAM memory size and MAC, see Table 3). As a consequence of that, the throughput is reduced. Once again, DS reached the optimal split in 236 iterations, while B&B ranged between 66 and 483094 explored nodes verifying that for the MobileNet v1 architecture the best pairs of heuristics are the ones reported above (check darker squares in the heatmap drawn in Figure 7).
It is worth noting that replacing the devices and adopting almost the same architecture involves changes in the heatmap, which basically proves that the efficiency of the algorithm is both layer heuristic and device heuristic dependent.

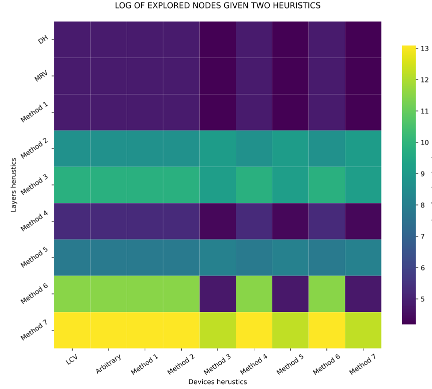



Figure 7: B&B heatmap for MobileNet v1 035.

## 5.4. Evaluation of YAMNet 256

This was a challenging as well as interesting test due to the high RAM size requirement in contrast to the low FLASH one, but the result was a failure in terms of DS's performance. STM32H743ZI and STM32L4R5ZI were used for the evaluation, but the FLASH to be used for both the MCUs was set to 25% of the nominal FLASH size. This model resulted in a sub-optimal solution of the DS for both the optmization policies found in 100 steps (almost 1.2% w.r.t the FS). In fact, as for the policy that maximize the throughput, the DS differs from the FS in both the number of sub-models (3 instead of 2) and the throughput estimation (0.146 $s^{-1}$ instead of 0.278 $s^{-1}$). The optimal solution assigns the first nine layers to the most powerful device STM32H743ZI, leaving the second sub-model made of the last four layers to the STM32L4R5ZI. On the other hand, the sub-optimal solution found by DS assigns the first and the last sub-model to the STM32H743ZI, while the middle one, made of only one layer, to the less powerful device. It is worth noting that the optimal configuration produces the same throughput independently from the used formula since (15) is empty. Concerning the total inference latency minimization, the FS obtained a two sub-models configuration that assigned the last four layers to the STM32L4R5ZI with an estimated total inference latency equal to 4.331 $s$. On the other hand, DS's solution assigned only the last two layers to the STM32L4R5ZI and its estimated latency was 7.531 $s$. We obtained a much higher latency estimation due to the increase in the amount of data to transfer from STM32H743ZI to STM32L4R5ZI that raises the communication latency (see Table 12 in Appendix A). As showed in Figure 8, the best layers heuristic is our *Method 4* that managed to reach optimality in 73 steps when combined with our devices heuristics *Methods 3, 5* and *7*. On the other hand, in the worst-case scenario the B&B algorithm achieved optimality in 1144 iteration when adopting *Method 7* for ordering the layers.

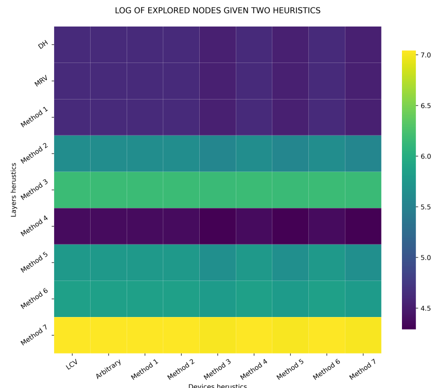
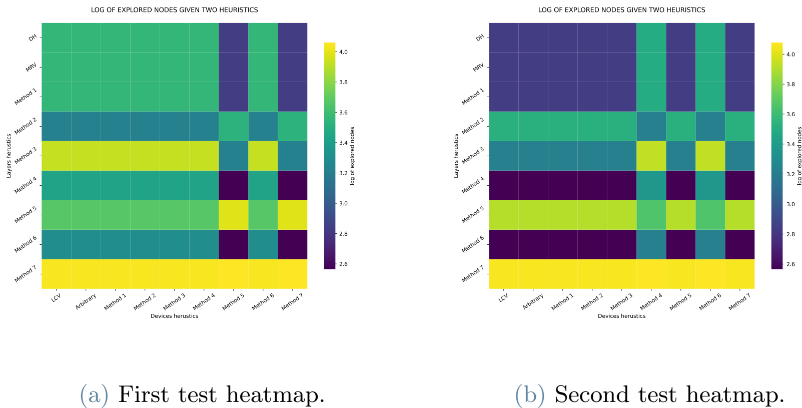



Figure 8: B&B heatmap for YAMNet 256.

## 5.5. Evaluation of VoxCeleb

To evaluate this model we performed two tests. In the former, STM32L452RE and STM32F446RE were used, which differ only in the CPU clock frequency: 80MHz and 180MHz, respectively (see Table 2). For both the objective functions, the optimal solution found by FS assigned the last two layers to the less powerful MCU (STM32L452RE) obtaining 0.684 $s$ as total inference latency and 1.492 $s^{-1}$ as throughput. DS managed to reach optimality in both problems by exploring the 41% of all the possible assignments. As for the B&B algorithm, the best pairs of heuristics were our *Method 4* and *6* for ordering the layers paired with *Methods 5* and *7* for selecting the devices, achieving the optimal solution by exploring barely 13 nodes (see Figure 9a).
The second test involved a different pair of boards (STM32F446RE and STM32H723ZG, where the FLASH size of the second MCU was set to 50% of the available one) but achieved the same sub-models as optimal solution. The first sub-model, made of the first 5 layers, was assigned to the more powerful STM32H723ZG. In this case, the estimated latency was 70% lower than the first case (0.208 $s$), while the the throughput was 70% higher (4.955 $s^{-1}$) since the STM32H723ZG has a higher clock frequency and lower CpM w.r.t the STM32L452RE. Again, DS managed to reach optimality in 52 iterations. Concerning the B&B, the performance was identical w.r.t the first test: only 13 explored nodes to reach the optimum, but as shown in Figure 9b, almost all the devices heuristics were successful.




(a) First test heatmap.  (b) Second test heatmap.

Figure 9: B&B heatmaps for VoxCeleb.

## 5.6. Evaluation of KWS CNN

This model has been evaluated on STM32L433RC and STM32L412KB, two low power MCUs that differ only in FLASH and RAM memories.
As for the latency minimization, all the algorithms found the optimal solution with an inference latency equal to 0.822 $s$ and divided the original network in three sub-models, where the first and last sub-models were assigned to the less restricted MCU (STM32L433RC) and the remaining one, made of a single layer, was assigned to STM32L412KB. DS managed to achieve the optimal solution in 46 steps (18% w.r.t FS), while B&B took 17 iterations when following our *Method 4* as layers heuristic. It is worth noting that, as showed in Figure 10, the worst possible devices heuristics are *Methods 2* and *4* which have to explore 116 nodes before finding the optimum.
As for the maximization throughput policy, both FS and DS produced an optimal solution made of three sub-models, but this time the second one has doubled the number of layers. The computed throughput was 1.216 $s^{-1}$ when using (17), because adopting the other throughput definition would lead to the *overlapping problem* with a 1% higher throughput (1.229 $s^{-1}$). In this case, the correct throughput estimation was exactly the inverse of the total inference latency since there was no possibility to speed up the execution: the next input could be processed only after an entire inference of the network.

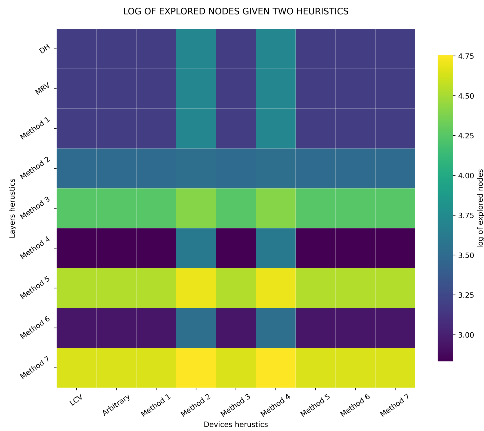



Figure 10: B&B heatmap for KWS CNN.

## 5.7. Evaluation of KWS DS-CNN

This model has been evaluated on two identical MCUs, namely the STM32F401RB. Since there was no difference in MCUs w.r.t layer's processing time, the solution in this setup relied on the total communication latency (12). As for the maximization throughput policy, the optimal solution split the original network in two sub-models (the first one made of the earlier four layers) and the estimated throughput was 0.431 $s^{-1}$. Both FS and DS achieved optimality in $2^{17}$ and 132 iterations, respectively.
As for the latency minimization, the three algorithms provided three different solutions, but all achieved the same total inference latency equal to 2.74 $s$. This is an unusual situation in which there were multiple optimal solutions due to the use of two identical MCUs and due to the model architecture of the DS-CNN model. In particular, this model includes the repetition of four identical blocks (each one consisting in SeparableConv2D - Conv2D - BatchNorm layers, see Table 7), which implies that there were multiple splitting points leading to multiple optimal solutions since the computational latency and/or the communication latency were identical for each block. DS was able to find the minimum latency by exploring only the 0.1% of the possible assignments, while the B&B took 80 steps (0.06% w.r.t FS) when adopting our *Method 4* for layers ordering. It is worth noting that in this case, the B&B algorithm's efficiency is independent from the devices heuristics and this is reflected in Figure 11: having the same MCUs on which to deploy the sub-models makes any device ordering completely useless.

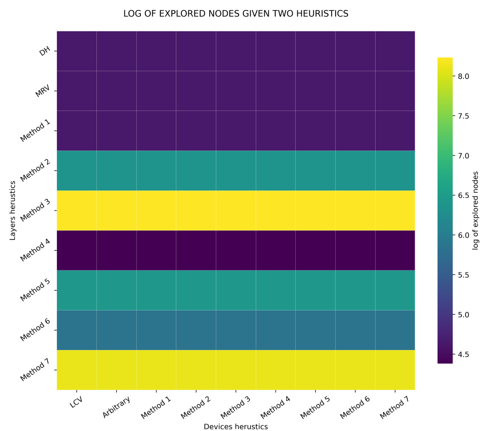



Figure 11: B&B heatmap for KWS DS-CNN.

## 5.8. Evaluation of Tiny-CNN

To evaluate this model, two identical MCUs were used again, but this time we decided to select a super restricted device: the STM32G071RB MCU. In order to be able to deploy the sub-models over the microcontrollers (storing the C code that handles the network) and to perform the split, we reduced the available FLASH size of the devices by 70 KBytes (see Table 2 to recall the device's properties).

The optimal solution of the maximization policy was different from the minimization one: in the first case, FS and DS split the model in three parts, where the middle one consisted in only one convolution layer (more precisely, the third one following the topological order in Table 13). It is worth noting that the second sub-model had the highest *model computational latency* (7), which implied that even if the optimal solution provided three sub-models, in this case, the computed throughput did not coincide with the inverse of the total inference latency (as happened in the tests reported in Subsection 5.2 and 5.6), but it was equal to 0.341 $s^{-1}$, which basically is the inverse of the sum between the *layer computational latency* (6) of the third layer of the network and its *communication latency* (14).

As for the minimization policy, each algorithm managed to reach optimality with an estimated total inference latency equal to 4.1 $s$. In particular, DS took 18 iterations (56% w.r.t FS) while B&B took only 13 steps (41% w.r.t FS). Here, the optimal solution divides the CNN in two sub-models: the first three layers were assigned to a microcontroller, while the two remaining layers were assigned to the other MCU. In Figure 12 we reported the B&B heatmap that shows the devices heuristic independence due to the usage of two identical MCUs.

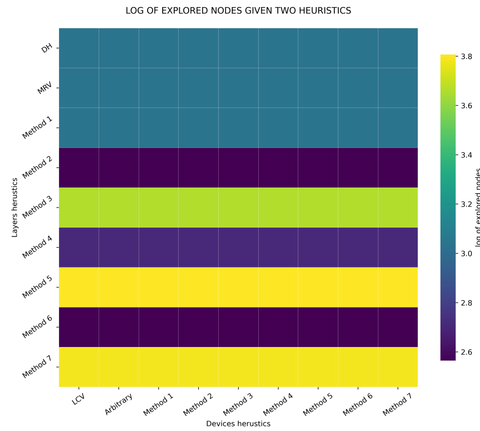



Figure 12: B&B heatmap for Tiny-CNN.

Table 4: Summary results of the minimization latency policy.

| Model | Used MCUs | Latency (s) | # Splits | FS steps | DS steps | BB steps |
|---|---|---|---|---|---|---|
| MobileNet 025 | STM32H743ZI<br>STM32L4R5ZI | 0.268 | 2 | $2^{30}$ | 236 | 66 |
| MobileNet 030 | STM32H743ZI<br>STM32F401RE | 1.839 | 3 | $2^{30}$ | 236 | 62 |
| MobileNet 035 | STM32H743ZI<br>STM32L4R5ZI | 0.448 | 2 | $2^{30}$ | 236 | 66 |
| YAMNet 256 | STM32H743ZI<br>STM32L4R5ZI | 4.331 | 2 | $2^{13}$ | 100 | 73 |
| VoxCeleb | STM32L452RE<br>STM32F446RE | 0.684 | 2 | $2^{7}$ | 52 | 13 |
| VoxCeleb | STM32F446RE<br>STM32H723ZG | 0.208 | 2 | $2^{7}$ | 52 | 13 |
| KWS CNN | STM32L433RC<br>STM32L412KB | 0.822 | 3 | $2^{8}$ | 46 | 17 |
| KWS DS-CNN | STM32F401RB<br>STM32F401RB | 2.74 | 2 | $2^{17}$ | 132 | 80 |
| Tiny-CNN | STM32G071RB<br>STM32G071RB | 4.10 | 2 | $2^{5}$ | 18 | 13 |

Table 5: Summary results of the maximization throughput policy.

| Model | Used MCUs | Throughput ($s^{-1}$) | # Splits | FS steps | DS steps |
|---|---|---|---|---|---|
| MobileNet 025 | STM32H743ZI<br>STM32L4R5ZI | 4.034 | 2 | $2^{30}$ | 236 |
| MobileNet 030 | STM32H743ZI<br>STM32F401RE | 0.544 ($\downarrow$ 13.5%) | 3 | $2^{30}$ | 236 |
| MobileNet 035 | STM32H743ZI<br>STM32L4R5ZI | 2.379 | 2 | $2^{30}$ | 236 |
| YAMNet 256 | STM32H743ZI<br>STM32L4R5ZI | 0.278 | 2 | $2^{13}$ | 100 |
| VoxCeleb | STM32L452RE<br>STM32F446RE | 1.492 | 2 | $2^{7}$ | 52 |
| VoxCeleb | STM32F446RE<br>STM32H723ZG | 4.955 | 2 | $2^{7}$ | 52 |
| KWS CNN | STM32L433RC<br>STM32L412KB | 1.216 ($\downarrow$ 1%) | 3 | $2^{8}$ | 46 |
| KWS DS-CNN | STM32F401RB<br>STM32F401RB | 0.431 | 2 | $2^{17}$ | 132 |
| Tiny-CNN | STM32G071RB<br>STM32G071RB | 0.341 | 3 | $2^{5}$ | 18 |

# 6. Porting a Neural Network model to a real IoT System

The methodology has been applied to the placement of the 5-layer CNN tested in Subsection 5.8 on a real technological scenario comprising two STM32G071RB MCUs depicted in Figure 14. We used the software STM32CubeMX [32] to generate the corresponding C code of the CNN model.

The chosen transmission technology is the UART transmission protocol with this parameters setting: baud rate set to 115200 bps, asynchronous mode, one stop bit, eight data bits and no parity bits. In particular, the jumpers shown in Figure 14b are responsible for linking both the microcontrollers to the ground (green) and allowing the communication between them (blue and beige). The goal of this experiment is to first validate the CNN placement provided by the *Neural Network Splitter* on physical devices, and second by comparing the *total inference latency* estimated by our tool (theoretical estimation) with the one measured when deploying the model on the IoT system (physical estimation). In particular, the figures of merit $L, T$, and $I$ reported in Table 6 are the *total computational latency* (10), the *total communication latency* (12) and the *total inference latency* (13), respectively. We recall that the optimal solution of the minimization problem found by the three algorithms assigned the first three layers of the CNN to one MCU and the last two layers to the other one, as shown in Figure 13 through a graphical representation of the Tiny-CNN and its sub-models by using the software Netron, *"a Visualizer for neural network, deep learning and machine learning models"* [44]. The measured transmission and processing times are particularly interesting, showing that the measured transmission time $T$ is almost equal to the estimated one, whereas the experimental processing time $L$ is 8% smaller than the estimated one, as reported in Table 6. This is justified by the fact the CpM parameter we used to compute the latency of each layer was estimated using X-CUBE-AI by deploying a similar CNN on the microcontroller providing us a CpM equal to 307 (such a high value is due to the fact that the data format of the Tiny-CNN model is *float32* and the STM32G071RB does not have a floating point unit (FPU)). Unfortunately, the CpM estimation is actually NN architecture dependent. In fact, after the deployment of the sub-models we estimated again the CpM for each device and we obtained two different values: as for the MCU assigned to the first heavier sub-model, its CpM was equal to 294, while for the other device its estimation was equal to 216. Since the latency is directly proportional to the CpM parameter, then the lower the parameter is, the lower the estimated latency becomes, which is reflected in a difference between theoretical and practical estimations.

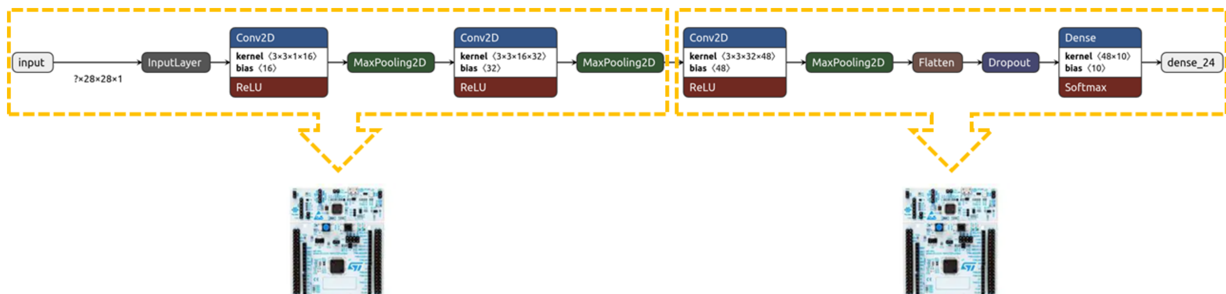
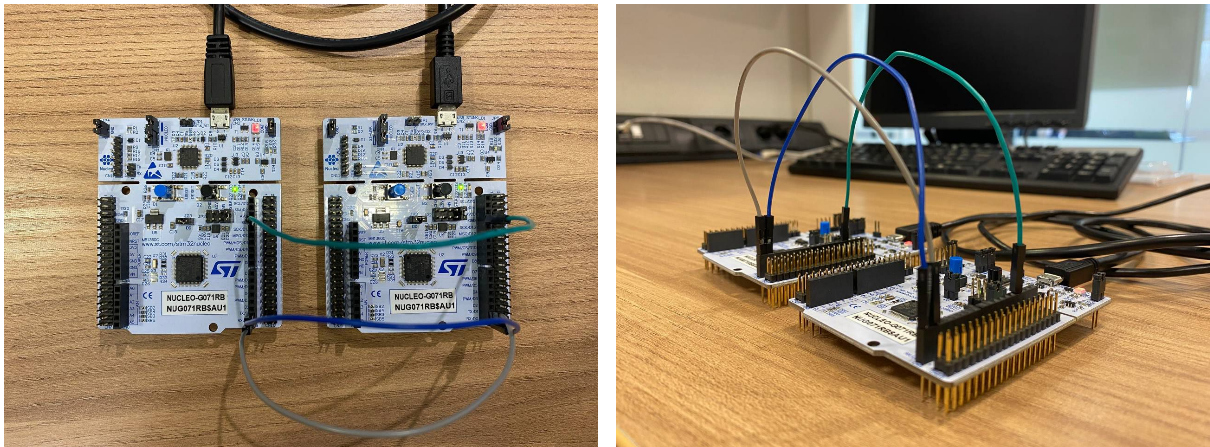



Figure 13: Visualization of the CNN and its optimal decomposition.



(a) Front view of the CNN placement on the MCUs. (b) Side view of the CNN placement on the MCUs.

Figure 14: Photos of the CNN placement on a real technological scenario.

Table 6: Experimental benchmark results of Tiny-CNN and two STM32G071RB devices. The figure of merit is the total inference latency $I$ (processing $L$ plus transmission $T$) and is measured in seconds.

| **Case** | $L$ (s) | $T$ (s) | $I = L + T$ (s) |
|---|---|---|---|
| Model | 3.88 | 0.22 | 4.1 |
| Experimental | 3.56 | 0.25 | 3.81 |

## 7. Conclusions

This thesis aimed to introduce the *Neural Network Splitter*, a software tool that allows to automatically partition a given pre-trained NN model over multiple heterogeneous tiny devices without affecting the architecture of the model or its accuracy. The partitioning problem has been modeled through a detailed mathematical formulation and solved using three different algorithms, namely FS, DS, and B&B. To evaluate the proposed methodology a detailed experimental campaign has been carried out on several CNN architectures and heterogeneous MCUs in memory and computing properties. The obtained results showed that, among the considered algorithms, the B&B obtained the best results because not only it always finds the optimum but it also achieved the lowest number of steps to find it. Concerning FS, it turned out to have limitations with deeper network architectures since exploring all the possible candidates requires significant computational efforts and might be also impractical. On the other hand, since DS's complexity is linear in the number of layers, this algorithm can be an alternative to the FS. However, its main downside is that the optimality is not guaranteed due to its searching strategy which could not explore the whole search space.
As future work possibilities, one could be to extend the mathematical formulation by including the power consumption in the constraints as well as in the objective function. The former could affect the solution by making some candidates not admissible anymore, while the latter would provide a different optimization problem where the optimal solution prefers lower power consumption than minimizing the inference latency or the waiting time between two consecutive CNN executions.
From the algorithmic point of view, DS can be improved by introducing an adaptive bisection that splits the current candidate solution based on the network's profiles (FLASH, RAM, and/or MAC) instead of applying bisection at fixed points regardless of its computational properties in order to make this algorithm more consistent in terms of finding the optimal solution. Furthermore, the B&B can be improved by introducing innovative admissible heuristics for selecting the next layer-device assignment in the throughput maximization problem.
Finally, the last open point consists in developing a non-trivial method in order to apply this methodology to multi-branches neural network models. By the way, we plan to extend our tool to better handle networks with branches in order to evaluate other state-of-the-art NNs such as [45–48].

## References


[1] Ramon Sanchez-Iborra and Antonio F Skarmeta. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020.

[2] Bharath Sudharsan, Simone Salerno, Duc-Duy Nguyen, Muhammad Yahya, Abdul Wahid, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. Tinyml benchmark: Executing fully connected neural networks on commodity microcontrollers. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pages 883–884. IEEE, 2021.

[3] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9):2533, 2020.

[4] Simone Disabato, Manuel Roveri, and Cesare Alippi. Distributed deep convolutional neural networks for the internet-of-things. *IEEE Transactions on Computers*, 2021.

[5] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 328–339. IEEE, 2017.

[6] Kartikeya Bhardwaj, Naveen Suda, and Radu Marculescu. Edgeal: A vision for deep learning in the iot era. *IEEE Design & Test*, 38(4):37–43, 2019.

[7] Kartikeya Bhardwaj, Ching-Yi Lin, Anderson Sartor, and Radu Marculescu. Memory-and communication-aware model compression for distributed deep learning inference on iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019.

[8] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[9] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2019.

[10] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.

[11] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[13] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[14] 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.

[15] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

[16] Shaohui Lin, Rongrong Ji, Chao Chen, Dacheng Tao, and Jiebo Luo. Holistic cnn compression via low-rank decomposition with knowledge transfer. *IEEE transactions on pattern analysis and machine intelligence*, 41(12):2889–2905, 2018.

[17] Dawei Li, Xiaolong Wang, and Deguang Kong. Deeprebirth: Accelerating deep neural network execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[18] Simone Disabato and Manuel Roveri. Reducing the computation load of convolutional neural networks through gate classification. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[19] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.

[20] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

[21] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[22] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[23] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in cognitive science. Academic Press, 1993. ISBN 9780127016108.

[24] Malek Mouhoub and Bahareh Jafari. Heuristic techniques for variable and value ordering in csps. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 457–464, 2011.

[25] Igor Razgon. Complexity analysis of heuristic csp search algorithms. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 88–99. Springer, 2005.

[26] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Pearson, 3 edition, 2009.

[27] 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.

[28] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. Voxceleb: a large-scale speaker identification dataset. *arXiv preprint arXiv:1706.08612*, 2017.

[29] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs. Available online: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[30] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[31] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *CoRR*, abs/1711.07128, 2017.

[32] *X-CUBE-AI - AI expansion pack for STM32CubeMX.* STMicroelectronics, 2022. Available online: `www.st.com/en/embedded-software/x-cube-ai`.

[33] *UM2526 - Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI).* STMicroelectronics, 2022. Available online: `https://tinyurl.com/3ssemzfv`.

[34] *Datasheet - STM32H742xI/G STM32H743xI/G.* STMicroelectronics, 2022. Available online: `https://www.st.com/resource/en/datasheet/stm32h743zi.pdf`.

[35] *Datasheet - STM32H730AB STM32H730IB STM32H730VB STM32H730ZB.* STMicroelectronics, 2021. Available online: `https://www.st.com/resource/en/datasheet/stm32h723zg.pdf`.

[36] *Datasheet - STM32F446xC/E.* STMicroelectronics, 2021. Available online: `https://www.st.com/resource/en/datasheet/stm32f446re.pdf`.

[37] *Datasheet - STM32F401xD STM32F401xE.* STMicroelectronics, 2015. Available online: `https://www.st.com/resource/en/datasheet/stm32f401re.pdf`.

[38] *Datasheet - STM32F401xB STM32F401xC.* STMicroelectronics, 2019. Available online: `https://www.st.com/resource/en/datasheet/stm32f401rb.pdf`.

[39] *Datasheet - STM32L4R5xx STM32L4R7xx STM32L4R9xx.* STMicroelectronics, 2020. Available online: `https://www.st.com/resource/en/datasheet/stm32l4r5zi.pdf`.

[40] *Datasheet - STM32L452xx.* STMicroelectronics, 2020. Available online: `https://www.st.com/resource/en/datasheet/stm32l452re.pdf`.

[41] *Datasheet - STM32L433xx.* STMicroelectronics, 2021. Available online: `https://www.st.com/resource/en/datasheet/stm32l433rc.pdf`.

[42] *Datasheet - STM32L412xx.* STMicroelectronics, 2020. Available online: `https://www.st.com/resource/en/datasheet/stm32l412kb.pdf`.

[43] *Datasheet - STM32G071x8/xB.* STMicroelectronics, 2021. Available online: `https://www.st.com/resource/en/datasheet/stm32g071rb.pdf`.

[44] Lutz Roeder. Netron, Visualizer for neural network, deep learning, and machine learning models (version 1.2.0), 2017. `https://github.com/lutzroeder/netron`.

[45] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[48] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

# A. Appendix A

At the beginning of Section 5, the models used for the experimental campaign have been briefly introduced with a description of their goal and sometimes giving info in their architecture. This appendix is thus entirely dedicated to an in-depth analysis of these NNs from a technical point of view. In particular, each model is represented by a table where rows are the optimized layers (nodes of the computational graph) obtained through X-CUBE-AI and per each layer: the name, the shape of the input/output tensors, and the complexities are proposed.

Table 7: DS-CNN-KWS computational graph's characteristics.

| **Layer name** | **Input shape** | **Output shape** | **FLASH (KB)** | **RAM (KB)** | **MAC ($10^3$)** |
|---|---|---|---|---|---|
| Input | (49, 10, 1) | (49, 10, 1) | 0.0 | 1.914 | 0.0 |
| Conv2D | (49, 10, 1) | (25, 5, 64) | 10.25 | 56.25 | 328.064 |
| BatchNorm | (25, 5, 64) | (25, 5, 64) | 0.5 | 31.25 | 16.0 |
| SepConv2D | (25, 5, 64) | (25, 5, 64) | 18.75 | 34.5 | 584.128 |
| Conv2D | (25, 5, 64) | (25, 5, 64) | 16.25 | 32.75 | 520.064 |
| BatchNorm | (25, 5, 64) | (25, 5, 64) | 0.5 | 31.25 | 16.0 |
| SepConv2D | (25, 5, 64) | (25, 5, 64) | 18.75 | 34.5 | 584.128 |
| Conv2D | (25, 5, 64) | (25, 5, 64) | 16.25 | 32.75 | 520.064 |
| BatchNorm | (25, 5, 64) | (25, 5, 64) | 0.5 | 31.25 | 16.0 |
| SepConv2D | (25, 5, 64) | (25, 5, 64) | 18.75 | 34.5 | 584.128 |
| Conv2D | (25, 5, 64) | (25, 5, 64) | 16.25 | 32.75 | 520.064 |
| BatchNorm | (25, 5, 64) | (25, 5, 64) | 0.5 | 31.25 | 16.0 |
| SepConv2D | (25, 5, 64) | (25, 5, 64) | 18.75 | 34.5 | 584.128 |
| Conv2D | (25, 5, 64) | (25, 5, 64) | 16.25 | 32.75 | 520.064 |
| BatchNorm | (25, 5, 64) | (25, 5, 64) | 0.5 | 31.25 | 16.0 |
| AVGPool2D | (25, 5, 64) | (1, 1, 64) | 0.0 | 31.5 | 8.0 |
| Dense | (1, 1, 64) | (1, 1, 12) | 3.047 | 31.5 | 0.96 |

Table 8: MobileNet v1 025 computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (128, 128, 3) | (128, 128, 3) | 0.0 | 192.0 | 0.0 |
| Conv2D | (128, 128, 3) | (64, 64, 8) | 0.875 | 192.0 | 950.28 |
| Conv2D_dw | (64, 64, 8) | (64, 64, 8) | 0.313 | 256.0 | 360.456 |
| Conv2D_pw | (64, 64, 8) | (64, 64, 16) | 0.563 | 262.063 | 655.376 |
| Conv2D_dw | (64, 64, 16) | (32, 32, 16) | 0.625 | 258.063 | 180.24 |
| Conv2D_pw | (32, 32, 16) | (32, 32, 32) | 2.125 | 192.0 | 589.856 |
| Conv2D_dw | (32, 32, 32) | (32, 32, 32) | 1.25 | 136.375 | 360.48 |
| Conv2D_pw | (32, 32, 32) | (32, 32, 32) | 4.125 | 132.125 | 1114.144 |
| Conv2D_dw | (32, 32, 32) | (16, 16, 32) | 1.25 | 179.5 | 90.144 |
| Conv2D_pw | (16, 16, 32) | (16, 16, 64) | 8.25 | 96.0 | 557.12 |
| Conv2D_dw | (16, 16, 64) | (16, 16, 64) | 2.5 | 128.0 | 180.288 |
| Conv2D_pw | (16, 16, 64) | (16, 16, 64) | 16.25 | 160.0 | 1081.408 |
| Conv2D_dw | (16, 16, 64) | (8, 8, 64) | 2.5 | 80.0 | 45.12 |
| Conv2D_pw | (8, 8, 64) | (8, 8, 128) | 32.5 | 80.0 | 540.8 |
| Conv2D_dw | (8, 8, 128) | (8, 8, 128) | 5.0 | 64.0 | 90.24 |
| Conv2D_pw | (8, 8, 128) | (8, 8, 128) | 64.5 | 64.0 | 1065.088 |
| Conv2D_dw | (8, 8, 128) | (8, 8, 128) | 5.0 | 64.0 | 90.24 |
| Conv2D_pw | (8, 8, 128) | (8, 8, 128) | 64.5 | 64.0 | 1065.088 |
| Conv2D_dw | (8, 8, 128) | (8, 8, 128) | 5.0 | 64.0 | 90.24 |
| Conv2D_pw | (8, 8, 128) | (8, 8, 128) | 64.5 | 64.0 | 1065.088 |
| Conv2D_dw | (8, 8, 128) | (8, 8, 128) | 5.0 | 64.0 | 90.24 |
| Conv2D_pw | (8, 8, 128) | (8, 8, 128) | 64.5 | 64.0 | 1065.088 |
| Conv2D_dw | (8, 8, 128) | (8, 8, 128) | 5.0 | 64.0 | 90.24 |
| Conv2D_pw | (8, 8, 128) | (8, 8, 128) | 64.5 | 64.0 | 1065.088 |
| Conv2D_dw | (8, 8, 128) | (4, 4, 128) | 5.0 | 40.0 | 22.656 |
| Conv2D_pw | (4, 4, 128) | (4, 4, 256) | 129.0 | 40.0 | 532.736 |
| Conv2D_dw | (4, 4, 256) | (4, 4, 256) | 10.0 | 32.0 | 45.312 |
| Conv2D_pw | (4, 4, 256) | (1, 1, 256) | 257.0 | 17.0 | 1061.12 |
| Conv2D_preds | (1, 1, 256) | (1, 1, 1000) | 1003.91 | 33.0 | 257.0 |
| Predictions | (1, 1, 1000) | (1, 1, 1000) | 0.0 | 3.91 | 15.0 |

Table 9: MobileNet v1 030 computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (128, 128, 3) | (128, 128, 3) | 0.0 | 192.0 | 0.0 |
| Conv2D | (128, 128, 3) | (64, 64, 9) | 0.984 | 192.0 | 1069.065 |
| Conv2D_dw | (64, 64, 9) | (64, 64, 9) | 0.352 | 288.0 | 405.513 |
| Conv2D_pw | (64, 64, 9) | (64, 64, 19) | 0.742 | 311.324 | 856.083 |
| Conv2D_dw | (64, 64, 19) | (32, 32, 19) | 0.742 | 306.449 | 214.035 |
| Conv2D_pw | (32, 32, 19) | (32, 32, 38) | 2.969 | 228.0 | 817.19 |
| Conv2D_dw | (32, 32, 38) | (32, 32, 38) | 1.484 | 161.945 | 428.07 |
| Conv2D_pw | (32, 32, 38) | (32, 32, 38) | 5.789 | 156.898 | 1556.518 |
| Conv2D_dw | (32, 32, 38) | (16, 16, 38) | 1.484 | 213.156 | 107.046 |
| Conv2D_pw | (16, 16, 38) | (16, 16, 76) | 11.578 | 114.0 | 778.316 |
| Conv2D_dw | (16, 16, 76) | (16, 16, 76) | 2.969 | 152.0 | 214.092 |
| Conv2D_pw | (16, 16, 76) | (16, 16, 76) | 22.86 | 190.0 | 1517.644 |
| Conv2D_dw | (16, 16, 76) | (8, 8, 76) | 2.969 | 95.0 | 53.58 |
| Conv2D_pw | (8, 8, 76) | (8, 8, 153) | 46.02 | 95.0 | 763.929 |
| Conv2D_dw | (8, 8, 153) | (8, 8, 153) | 5.977 | 76.5 | 107.865 |
| Conv2D_pw | (8, 8, 153) | (8, 8, 153) | 92.039 | 76.5 | 1517.913 |
| Conv2D_dw | (8, 8, 153) | (8, 8, 153) | 5.977 | 76.5 | 107.865 |
| Conv2D_pw | (8, 8, 153) | (8, 8, 153) | 92.039 | 76.5 | 1517.913 |
| Conv2D_dw | (8, 8, 153) | (8, 8, 153) | 5.977 | 76.5 | 107.865 |
| Conv2D_pw | (8, 8, 153) | (8, 8, 153) | 92.039 | 76.5 | 1517.913 |
| Conv2D_dw | (8, 8, 153) | (8, 8, 153) | 5.977 | 76.5 | 107.865 |
| Conv2D_pw | (8, 8, 153) | (8, 8, 153) | 92.039 | 76.5 | 1517.913 |
| Conv2D_dw | (8, 8, 153) | (8, 8, 153) | 5.977 | 76.5 | 107.865 |
| Conv2D_pw | (8, 8, 153) | (8, 8, 153) | 92.039 | 76.5 | 1517.913 |
| Conv2D_dw | (8, 8, 153) | (4, 4, 153) | 5.977 | 47.813 | 27.081 |
| Conv2D_pw | (4, 4, 153) | (4, 4, 307) | 184.68 | 47.813 | 761.667 |
| Conv2D_dw | (4, 4, 307) | (4, 4, 307) | 11.992 | 38.375 | 54.339 |
| Conv2D_pw | (4, 4, 307) | (1, 1, 307) | 369.359 | 20.387 | 1523.027 |
| Conv2D_preds | (1, 1, 307) | (1, 1, 1000) | 1203.125 | 39.574 | 308.0 |
| Predictions | (1, 1, 1000) | (1, 1, 1000) | 0.0 | 3.906 | 15.0 |

Table 10: MobileNet v1 035 computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (128, 128, 3) | (128, 128, 3) | 0.0 | 192.0 | 0.0 |
| Conv2D | (128, 128, 3) | (64, 64, 11) | 1.203 | 192.0 | 1306.635 |
| Conv2D_dw | (64, 64, 11) | (64, 64, 11) | 0.43 | 352.0 | 495.627 |
| Conv2D_pw | (64, 64, 11) | (64, 64, 22) | 1.031 | 360.336 | 1171.478 |
| Conv2D_dw | (64, 64, 22) | (32, 32, 22) | 0.86 | 354.836 | 247.83 |
| Conv2D_pw | (32, 32, 22) | (32, 32, 44) | 3.953 | 264.0 | 1081.388 |
| Conv2D_dw | (32, 32, 44) | (32, 32, 44) | 1.719 | 187.516 | 495.66 |
| Conv2D_pw | (32, 32, 44) | (32, 32, 44) | 7.734 | 181.672 | 2072.62 |
| Conv2D_dw | (32, 32, 44) | (16, 16, 44) | 1.719 | 246.813 | 123.948 |
| Conv2D_pw | (16, 16, 44) | (16, 16, 89) | 15.645 | 133.0 | 1048.153 |
| Conv2D_dw | (16, 16, 89) | (16, 16, 89) | 3.477 | 178.0 | 250.713 |
| Conv2D_pw | (16, 16, 89) | (16, 16, 89) | 31.289 | 222.0 | 2073.433 |
| Conv2D_dw | (16, 16, 89) | (8, 8, 89) | 3.477 | 111.25 | 62.745 |
| Conv2D_pw | (8, 8, 89) | (8, 8, 179) | 62.93 | 111.25 | 1042.675 |
| Conv2D_dw | (8, 8, 179) | (8, 8, 179) | 6.992 | 89.5 | 126.195 |
| Conv2D_pw | (8, 8, 179) | (8, 8, 179) | 125.86 | 89.5 | 2073.715 |
| Conv2D_dw | (8, 8, 179) | (8, 8, 179) | 6.992 | 89.5 | 126.195 |
| Conv2D_pw | (8, 8, 179) | (8, 8, 179) | 125.86 | 89.5 | 2073.715 |
| Conv2D_dw | (8, 8, 179) | (8, 8, 179) | 6.992 | 89.5 | 126.195 |
| Conv2D_pw | (8, 8, 179) | (8, 8, 179) | 125.859 | 89.5 | 2073.715 |
| Conv2D_dw | (8, 8, 179) | (8, 8, 179) | 6.992 | 89.5 | 126.195 |
| Conv2D_pw | (8, 8, 179) | (8, 8, 179) | 125.859 | 89.5 | 2073.715 |
| Conv2D_dw | (8, 8, 179) | (8, 8, 179) | 6.992 | 89.5 | 126.195 |
| Conv2D_pw | (8, 8, 179) | (8, 8, 179) | 125.859 | 89.5 | 2073.715 |
| Conv2D_dw | (8, 8, 179) | (4, 4, 179) | 6.992 | 55.938 | 31.683 |
| Conv2D_pw | (4, 4, 179) | (4, 4, 358) | 251.719 | 55.938 | 1037.126 |
| Conv2D_dw | (4, 4, 358) | (4, 4, 358) | 13.984 | 44.75 | 63.366 |
| Conv2D_pw | (4, 4, 358) | (1, 1, 358) | 502.039 | 23.773 | 2068.166 |
| Conv2D_preds | (1, 1, 358) | (1, 1, 1000) | 1402.344 | 46.148 | 359.0 |
| Predictions | (1, 1, 1000) | (1, 1, 1000) | 0.0 | 3.906 | 15.0 |

Table 11: CNN-KWS computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (49, 10, 1) | (49, 10, 1) | 0.0 | 1.914 | 0.0 |
| Conv2D | (49, 10, 1) | (40, 7, 28) | 4.484 | 31.211 | 321.468 |
| BatchNorm | (40, 7, 28) | (40, 7, 28) | 0.219 | 30.625 | 15.68 |
| Conv2D | (40, 7, 28) | (16, 4, 30) | 131.367 | 31.211 | 2152.35 |
| BatchNorm | (16, 4, 30) | (1, 1, 1920) | 0.234 | 15.0 | 3.84 |
| Dense | (1, 1, 1920) | (1, 1, 16) | 120.063 | 15.0 | 30.736 |
| Dense | (1, 1, 16) | (1, 1, 128) | 8.5 | 1.0 | 2.304 |
| Dense | (1, 1, 128) | (1, 1, 12) | 6.047 | 1.063 | 1.728 |

Table 12: YAMNet 256 computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (96, 64, 1) | (96, 64, 1) | 0.0 | 24.0 | 0.0 |
| Conv2D | (96, 64, 1) | (48, 32, 32) | 1.25 | 208.5 | 491.552 |
| Conv2D_dw | (48, 32, 32) | (48, 32, 32) | 1.25 | 384.0 | 491.552 |
| Conv2D_pw | (48, 32, 32) | (48, 32, 64) | 8.25 | 396.25 | 3244.096 |
| Conv2D_dw | (48, 32, 64) | (24, 16, 64) | 2.5 | 388.25 | 245.824 |
| Conv2D_pw | (24, 16, 64) | (24, 16, 128) | 32.5 | 288.0 | 3195.008 |
| Conv2D_dw | (24, 16, 128) | (24, 16, 128) | 5.0 | 209.5 | 491.648 |
| Conv2D_pw | (24, 16, 128) | (24, 16, 128) | 64.5 | 200.5 | 6340.736 |
| Conv2D_dw | (24, 16, 128) | (12, 8, 128) | 5.0 | 262.0 | 123.008 |
| Conv2D_pw | (12, 8, 128) | (12, 8, 256) | 129.0 | 144.0 | 3170.56 |
| Conv2D_dw | (12, 8, 256) | (12, 8, 256) | 10.0 | 192.0 | 246.016 |
| Conv2D_pw | (12, 8, 256) | (12, 8, 256) | 257.0 | 240.0 | 6316.288 |
| Conv2D_dw | (12, 8, 256) | (1, 1, 256) | 10.0 | 96.0 | 67.84 |

Table 13: Tiny-CNN computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (28, 28, 1) | (28, 28, 1) | 0.0 | 3.0625 | 0.0 |
| Conv2D | (28, 28, 1) | (13, 13, 16) | 0.625 | 11.313 | 118.992 |
| Conv2D | (13, 13, 16) | (5, 5, 32) | 18.125 | 11.313 | 564.672 |
| Conv2D | (5, 5, 32) | (1, 1, 48) | 54.188 | 4.438 | 125.088 |
| Dense | (1, 1, 48) | (1, 1, 10) | 1.914 | 4.438 | 0.64 |

Table 14: VoxCeleb computational graph's characteristics.

| Layer name | Input shape | Output shape | FLASH (KB) | RAM (KB) | MAC ($10^3$) |
|---|---|---|---|---|---|
| Input | (100, 13, 1) | (100, 13, 1) | 0.0 | 5.078 | 0.0 |
| Conv2D | (100, 13, 1) | (50, 6, 32) | 1.25 | 39.75 | 454.432 |
| Conv2D | (50, 6, 32) | (25, 3, 64) | 72.25 | 39.75 | 5568.064 |
| Conv2D | (25, 3, 64) | (12, 1, 128) | 288.5 | 27.75 | 5545.472 |
| SepConv2D | (12, 1, 128) | (1, 1, 256) | 134.0 | 27.75 | 413.568 |
| Dense | (1, 1, 256) | (1, 1, 128) | 128.5 | 19.0 | 33.024 |
| Dense | (1, 1, 128) | (1, 1, 600) | 302.344 | 3.344 | 86.4 |

# List of Figures


1 Example of a CNN partitioning. . . . 5
2 Example of a schedule using two throughput's definitions. . . . 6
3 Flowchart of the B&B algorithm. . . . 9
4 FLASH (blue), RAM (orange), and MACC (green) normalized profiles of the CNN models. . . . 11
5 B&B heatmap for MobileNet v1 025. . . . 12
6 B&B heatmap for MobileNet v1 030. . . . 13
7 B&B heatmap for MobileNet v1 035. . . . 14
8 B&B heatmap for YAMNet 256. . . . 14
9 B&B heatmaps for VoxCeleb. . . . 15
10 B&B heatmap for KWS CNN. . . . 16
11 B&B heatmap for KWS DS-CNN. . . . 16
12 B&B heatmap for Tiny-CNN. . . . 17
13 Visualization of the CNN and its optimal decomposition. . . . 19
14 Photos of the CNN placement on a real technological scenario. . . . 19


# List of Tables


1 Algorithms comparison. . . . 9
2 MCUs memory and computing properties. . . . 10
3 CNN models used in the experimental evaluation. . . . 12
4 Summary results of the minimization latency policy. . . . 18
5 Summary results of the maximization throughput policy. . . . 18
6 Experimental benchmark results of Tiny-CNN and two STM32G071RB devices. . . . 20
7 DS-CNN-KWS computational graph's characteristics. . . . 23
8 MobileNet v1 025 computational graph's characteristics. . . . 24
9 MobileNet v1 030 computational graph's characteristics. . . . 25
10 MobileNet v1 035 computational graph's characteristics. . . . 26
11 CNN-KWS computational graph's characteristics. . . . 27
12 YAMNet 256 computational graph's characteristics. . . . 27
13 Tiny-CNN computational graph's characteristics. . . . 27
14 VoxCeleb computational graph's characteristics. . . . 28

# Abstract in lingua italiana

L'implementazione di una rete neurale (NN) su dispositivi a bassa potenza e con risorse limitate rappresenta un problema critico nello sviluppo di sistemi IoT intelligenti ed autonomi a causa degli aggressivi vincoli computazionali e di memoria. Per questo motivo, le soluzioni di Machine Learning (ML) rivolte a piccoli dispositivi devono essere progettate tenendo presente i vincoli legati alla memoria e alla capacità di elaborazione che caratterizzano tali dispositivi. In questa tesi, introduciamo una nuova metodologia di progettazione basata su un approccio distribuito, il quale ha come obiettivo partizionare automaticamente l'esecuzione di una NN su più dispositivi eterogenei molto limitati. Tale metodologia è formalizzata come un problema di ottimizzazione in cui o la latenza di inferenza è minimizzata oppure il throughput è massimizzato, tenendo in considerazione le capacità di memoria e di calcolo dei dispositivi. La metodologia è valutata su diverse architetture di reti neurali e su microcontrollori (MCUs) utilizzando tre algoritmi, vale a dire il Full Search (FS), il Dichotomich Search (DS) ed il Branch-and-Bound (B&B). I risultati ottenuti hanno mostrato che il B&B ha performato in modo di gran lunga migliore rispetto agli altri, in quanto è stato sempre in grado di trovare la soluzione ottima nel minor numero di iterazioni. Con questo lavoro, cerchiamo di proporre nuove soluzioni di ML che offrano una bassa decision-latency, autonomia ed un'elevata efficienza energetica.



**Parole chiave:** Machine Learning, Reti Neurali, Sistemi Distribuiti, Problema di Ottimizzazione, Microcontrollori



# Acknowledgements

Ammetto che di discorsi conclusivi ne ho scritti un po' nel corso degli anni, ma di veri e propri ringraziamenti non ne sono sicuro. Credo dunque che farò a modo mio. In primis, e qui sì per importanza, vorrei ringraziare i miei genitori che mi hanno permesso di iniziare e concludere gli studi senza ulteriori preoccupazioni. Ho avuto la fortuna di nascere in una famiglia che non mi ha mai fatto mancare nulla, ecco forse qualche volta un po' di silenzio non sarebbe guastato, ma me lo sono fatto andare bene ugualmente.
I FRAH, amici di una vita, pensare che alcuni li conosco da praticamente vent'anni mi fa impressione. Quando necessitavo di sfogarmi per lo stress accumulato nelle sessioni, loro erano pronti per farmi fare serata, anche se di alcune non ricordo alcunché. Ma non si tratta solo delle feste, è qualcosa che va oltre, che non è facile da spiegare, semplicemente sono fortunato ad avere un gruppo di amici così. Biagio, anche conosciuto come Boss, capoBiagio, il mio tutor in ST. Dire che mi ha dato una mano è riduttivo, mi ha seguito fino all'ultimo giorno, ci teneva moltissimo al mio percorso e voleva che le cose fossero fatte bene e con i tempi giusti. Felice di averlo avuto come supervisor, ma ancora di più come amico. Il Professor Marcon, perché è uno di quei casi in cui la persona viene prima della professione. Lasciamo stare che è il mio relatore, quando ho seguito il suo corso mi sono appassionato. Ho persino chiesto del materiale aggiuntivo per approfondire un argomento e questa cosa non mi era mai capitata. Fu lui a propormi lo stage in azienda e di conseguenza è anche merito suo di quello che mi è accaduto negli ultimi mesi. E come lui, vorrei ringraziare altri professori incontrati nel mio percorso accademico: la Professoressa Paganoni, perché era un po' la mamma di noi studenti, il perfetto ponte tra liceo ed università. Il Professor Campi, perché mi ha trasmesso che fuori dalle mura universitarie c'è molto di più. Ancora oggi ricordo il suo discorso ispirazionale nei 10 minuti finali dell'ultima lezione del corso, ci fu una standing ovation. Il Professor Dario Andrea Nicola Natali (grazie ancora Beep per questi nomi aggiuntivi che crearono lo sdoppiamento tra insegnante ed esercitatore), per la sua genuina simpatia ma sopratutto per la maglia del gatto fronte-retro. Io non dimentico. Il Professor Micheletti, che dire, non era un professore universitario, era un one man show. Sono grato di aver potuto partecipare a più di un suo spettacolo live. La Professoressa Guglielmi, per avermi fatto scoprire ed innamorare della statistica bayesiana. Magari non sarà il docente più adorato dagli studenti, ma di sicuro *Bayesians do it better*. Il Professor Marzocchi, di cui sono tutt'ora un iscritto su Youtube, perché quale altro Professore riuscirebbe a spiegare un argomento con un video in cui taglia carote e patate? Il suo corso mi ha davvero divertito e mi ha insegnato un sacco di cose, sono orgoglioso di averlo seguito. Infine, la Professoressa Carrera, che ai tempi del liceo, nonostante qualche animato scambio di idee sui miei inconsueti modi di fare, mi ha indirizzato a scegliere questa facoltà.
Cambiando discorso, tre anni fa sono diventato un Clover, i sogni che avevo da ragazzino si sono in parte avverati perché ho finalmente trovato un gruppo con il quale mi diverto in campo e fuori. Vi ringrazio per come mi avete accolto e per tutto quello che passeremo negli anni futuri.
Come ultimi, vorrei ringraziare i miei gatti, Stella e Fulmine. Sono quasi certo di poter affermare che sono gli esseri viventi con cui ho fisicamente passato più tempo in questi anni. Mi scandivano le pause dallo studio, mi facevano compagnia e mi accompagnavano a letto. Onestamente non so come avrei fatto senza di voi. Grazie.