POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

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

Laurea Magistrale in Mathematical Engineering - Ingegneria Matematica

Author: Andrea Santamaria

Advisor: Prof. Marco Marcon

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

Academic year: 2020-2021

1. Introduction

In recent years, Deep Learning (DL) has achieved great success and state-of-the-art performance in numerous applications providing an opportunity for integrating Machine Learning (ML) into low power resource constrained devices such as sensors and microcontrollers (MCUs). Indeed, the opportunity of deploying Neural Network (NN) models as close as possible to the sensing device to process acquired data, is key to develop autonomous Internet of Things (IoT) systems and opens up to many benefits for lots of real-world embedded applications [1]. However, the deployment of accurate NN models on tiny devices represents a critical challenge due to the aggressive computational and memory constraints. The problem of reducing the complexity of DL solutions to match the technological constraints of IoT systems is becoming more and more relevant from both the scientific and technological perspective [1, 2]. Solutions present in the literature address such a problem using different approximation and optimization techniques (i.e., parameter pruning and sharing, quantization and knowledge distillation). Unfortunately, these approaches often require to re-design or modify the models' topology to be deployed and may 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 devices. In this context, we propose the *Neural Network Splitter*, a software tool whose goal is to automatically distribute a given pre-trained NN on multiple MCUs in order to optimize an objective function while satisfying the memory and computational constraints of the devices themselves as well as preserving the model architecture and its level of accuracy. With respect to the literature, the novelties of this work can be summarized as follow:

- the methodology adopted to solve the problem, which takes into account the devices' memory and computation capabilities, as well as the communication requirements;
- a detailed mathematical formulation for the problem definition which extends the formulation of [2] in order to solve *overlapping problems*.

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

2. Problem Definition and Assumptions

The problem of how to optimally distribute the execution of a pre-trained NN on multiple tiny devices can be modeled as an Optimization Problem (OP). Two objective functions have been taken into account:

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

The solution of the problem assigns each layer to a MCU in order to optimize the chosen objective function while satisfying the technological constrains imposed by the IoT system, as well as preserving the behavior of the original model (same architecture and accuracy). Despite the formalization is general enough to work with any NN topology, we assumed to use sequential Convolutional Neural Networks (CNNs).

2.1. Mathematical Formulation

Let a CNN with n layers to be deployed on d devices. Without loss of generality, we can represent the CNN as a computational graph where the nodes are the layers, while the edges correspond to the input/output tensors of a layer. Assuming to assign a sequential number to the layers sorted in topological order, let us define the layers set $N = \{1, \dots, n\}$ and let us also define $D = \{1, \dots, d\}$ as the set of the available devices each one represented by a unique ID. Therefore, there are d^n candidate solutions expressed by $P_p = \{(layer_1, dev_1), \dots, (layer_n, dev_d)\}$ for $p = 1, \dots, d^n$, where each tuple is a layer-device assignment and let $P = \{P_1, \dots, P_{d^n}\}$ be the candidates' set. Given the solution of the problem, a sub-model is obtained by grouping together all the consecutive layers assigned to the same device. By enumerating the sub-models in topological order, let us define $M = \{1, \dots, m\}$ as the set of sub-models, where $1 \leq \mathbf{card}(M) \leq n$. Let $N^{(m)}$ $\forall m \in M$, a partition of N, be the set of the consecutive layers belonging to the m^{th} sub-model. Similarly, let $M^{(i)} = \{m^i_{first}, \dots, m^i_{last}\}$ $\forall i \in D$, a partition of M, be the set of the sub-models (in topological order) processed by the i^{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. Each layer is characterized 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 (FLASH and embedded RAM memory size), and performance properties, i.e. operating CPU clock frequency (CPUFREQ) and average number of cycles per MAC (CpM).

A candidate solution must satisfy these memory constraints in order to deploy the obtained sub-models on the corresponding devices:

- there must be at least one device on which to deploy the most memory demanding layer in terms of FLASH and RAM size;
- for each device, its FLASH (RAM) 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.

On the other hand, the performance properties are used in (1) to define the *layer computational latency* as the time (in seconds) required to process a layer $j \in N$ by a device $i \in D$. It is worth noting that if j not assigned to i, then L^{ij} is equal to zero.

$$L^{ij} = \frac{MAC_j \; CpM_i}{CPUFREQ_i} \tag{1}$$

Furthermore, let us define in (2) the *total computational latency*, one of the two elements needed to compute the total inference latency.

$$L = \sum_{i \in D} \sum_{m \in M^{(i)}} \sum_{j \in N^{(m)}} L^{ij} \tag{2}$$

Similarly, let us define in (3) 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 given baud rate. It could be equal to zero where there is no communication between the devices, i.e. there is no pair of layers assigned to them, one assigned to i and one to h, such that it contains two consecutive layers.

$$T^{ih} = \frac{Bytes\ to\ transfer}{baud\ rate} \tag{3}$$

Then, is it possible to define in (4) the *total communication latency* and, finally, in (5) the *total inference latency.*

$$T = \sum_{i,h \in D} T^{ih} \tag{4}$$

$$I = L + T \tag{5}$$

(5) represents the time (in seconds) needed to perform an entire inference and shall be minimized when adopting the first policy to solve the problem. The other policy we considered is the maximization of the throughput, which is basically the inverse of the waiting time (6) to process the next input when having multiple input samples to process one at a time.

$$throughput = \frac{1}{waiting\ time} \tag{6}$$

Without loss of generality, let us identify the unique device with the highest computational latency:

$$\bar{i} = \arg\max_{i \in D} \sum_{m \in M^{(i)}} \sum_{j \in N^{(m)}} L^{ij} \tag{7}$$

and the set $R^{(\bar{i})} = range(m^{\bar{i}}_{first}, m^{\bar{i}}_{last}) \setminus M^{(\bar{i})}$ which contains all the sub-models made of *"intermediate"* layers, namely those models including the layers that are not assigned to $\bar{i}$ in the schedule, but their processing must be done between the layers of the first and the last sub-models assigned to $\bar{i}$. It is worth noting that to define this set we took advantage of the *range(a,b)* function that generates the sequence of numbers starting from the given start integer a to the stop integer b.
Finally, it is possible to define the waiting time as a sum of three contributions (in seconds):

- the time required by the device $\bar{i}$ to process all the sub-models assigned to it;
- the sum of the times required by the device $\bar{i}$ to transfer its output data to the next devices;
- the time needed to process all the *"intermediate"* layers.

2.2. Improvement of the Waiting Time Definition

We improved the definition of the waiting time with respect to the one presented in [2], which is defined as the computational latency of the device $\bar{i}$ plus the time to transmit the data. It is worth noting that if using the definition of [2], we might incur into what we called an *"overlapping problem"*: after n inputs a device has to simultaneously process two different layers belonging to two different inferences. For example, as shown in Figure 2, given a 4-layers CNN and its optimal partitioning on two arbitrary MCUs (orange and blue, where the orange one is $\bar{i}$), 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 our definition. The key term to add in the formulation is the processing time of the *"intermediate"* layers ($layer_2$ in the above example).

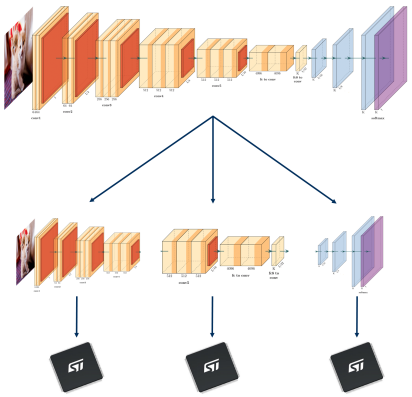
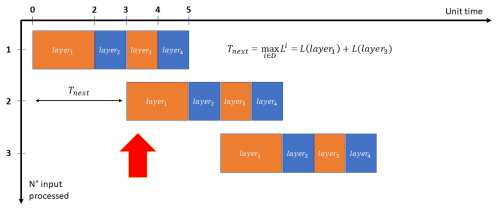
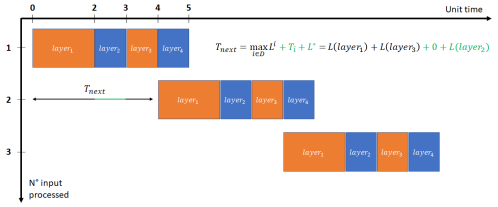


Figure 1: Example of a CNN partitioning.



(a) Overlapping



(b) Correct scheduling

Figure 2: Schedule obtained using the two definitions of the throughput.

3. Proposed Work

To solve the OP, three algorithms have been used, namely Full Search (FS), Dichotomic Search (DS) and Branch-and-Bound (B&B),

which are described in the follow and compared together in Table 1.

3.1. Full Search

This is the basic approach among the others and consists in exploring all the candidate solutions, one after the other, by checking at each step whether the current candidate P_p 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. However, its drawback is the exponential complexity in the number of layers, which implies that it is impractical when dealing with very deep NNs.

3.2. Dichotomic Search

The DS is a recursive algorithm that produces a bisection tree, which is explored in a depth-first search (DFS) fashion. It starts from an initial candidate solution (root node) that assigns all the layers to the same MCU. 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. 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.

3.3. Branch-and-Bound

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 the problem using the B&B we modeled it as a Constraint Satisfaction Optimization Problems (CSOP) [3]. 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. In our context, we decided to apply the B&B 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 (5), 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* (1) w.r.t 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. Finally, even though the time complexity of the B&B is exponential in the number of layers, 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. By construction, this algorithm guarantees to always find the optimal solution.

Table 1: Algorithms comparisons.

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

4. Experimental Results

To evaluate the three algorithms, we carried out a detailed experimental campaign considering eight CNNs models (whose properties are summarized in Table 2) and ten STM32 MCUs (see Table 3) characterized by heterogeneous memory and computational properties.

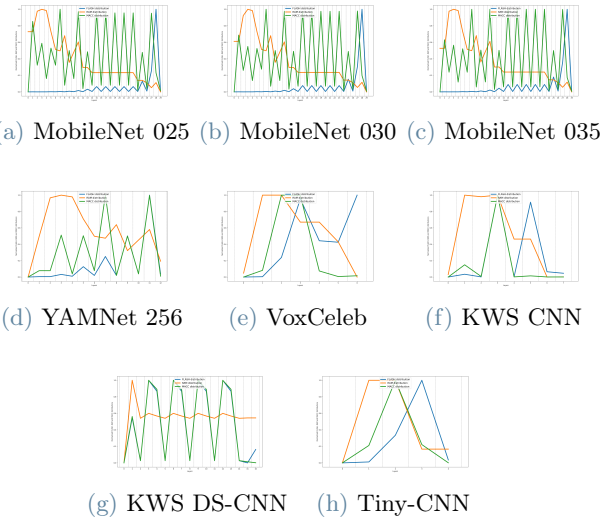
As for the CNN models, we used three MobileNets v1 which take as input a 128×128 RGB image, using different values for the α parameter (0.25, 0.30 and 0.35) to control the width of the network. YAMNet 256 is a modified version of the YAMNet model obtained by taking the first six convolution blocks of the original model, while VoxCeleb is a proprietary model trained on the VoxCeleb dataset 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 convolu-

Table 2: CNNs used in the experimental campaign.

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

Table 3: MCUs memory and computing properties.

MCU	FLASH (KB)	RAM (KB)	CPU Freq. (MHz)	CpM
STM32H743ZI	2048	1024	480	6
STM32H723ZG	1024	564	550	6
STM32F446RE	512	128	180	9
STM32F401RE	512	96	84	9
STM32F401RB	128	64	84	9
STM32L4R5ZI	2048	640	120	9
STM32L452RE	512	128	80	9
STM32L433RC	256	64	80	9
STM32L412KB	128	40	80	9
STM32G071RB	128	36	64	307



(a) MobileNet 025 (b) MobileNet 030 (c) MobileNet 035

(d) YAMNet 256 (e) VoxCeleb (f) KWS CNN

(g) KWS DS-CNN (h) Tiny-CNN

Figure 3: FLASH (blue), RAM (orange) and MACC (green) normalized profiles of the CNNs.

tion 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, 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 two models for keyword spotting (CNN and DS-CNN), whose weights and activations are in *float32* format.
The data reported in Table 2 have been obtained using the X-CUBE-AI tool v 7.1.0.
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 have been used. In particular, as for the minimization of the inference latency we used all the algorithm described in Section 3, while regarding the maximization of the throughput, we considered only FS and DS.
The obtained results are summarized in Table 4.

5. Porting a Neural Network model to a real IoT System

The methodology has been also applied to distribute the 5-layer Tiny-CNN on a real technological scenario comprising two STM32G071RB MCUs (see Figure 4). The goal of this experiment is to first validate the the CNN placement provided by the *Neural Network Splitter* on physical devices, and second by comparing the *total inference latency* estimated by our tool with the one measured when deploying the model on the IoT system. In particular, the figures of merit L, T and I reported in Table 5 are the *total computational latency* (2), the *total communication latency* (4) and the *total inference latency* (5), respectively. The optimal solution of the minimization problem assigned the first three layers of the CNN to one MCU and the last two layers to the other one.
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 measured processing time L is 8% smaller than the estimated one. This is justified by the fact the used CpM was estimated through another similar CNN model, but the CpM is actually NN architecture dependent.

6. Conclusions

The aim of this work was to introduce the *Neural Network Splitter*, a software tool that allows to automatically partition a given pre-trained NN model over multiple devices without affecting the model's architecture or its accuracy. The partitioning problem has been modeled through

Table 4: Summary results of the experimental evaluations (* when an algorithm did not reach optimality).

Model	Used MCUs	Latency (s)	# Splits	Throughput (s^{-1})	# Splits	FS steps	DS steps	BB steps
MobileNet v1 025	STM32H743ZI, STM32L4R5ZI	0.268	2	4.034	2	2^{30}	236	66
MobileNet v1 030	STM32H743ZI, STM32F401RE	1.839	3	0.544 (↓ 13.5%)	3	2^{30}	236	62
MobileNet v1 035	STM32H743ZI, STM32L4R5ZI	0.448	2	2.379	2	2^{30}	236	66
YAMNet 256	STM32H743ZI, STM32L4R5ZI	4.331	2	0.278	2	2^{13}	100*	73
VoxCeleb	STM32L452RE, STM32F446RE	0.684	2	1.492	2	2^{7}	52	13
VoxCeleb	STM32F446RE, STM32H723ZG	0.208	2	4.955	2	2^{7}	52	13
KWS CNN	STM32L433RC, STM32L412KB	0.822	3	1.216 (↓ 1%)	3	2^{8}	46	17
KWS DS-CNN	STM32F401RB, STM32F401RB	2.74	2	0.431	2	2^{17}	132	80
Tiny-CNN	STM32G071RB, STM32G071RB	4.10	2	0.341	3	2^{5}	18	13

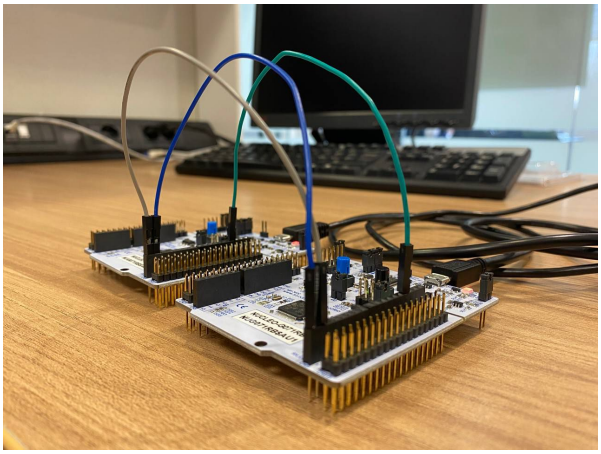


Figure 4: CNN placement on a real IoT system.

Table 5: Experimental benchmark results of Tiny-CNN and two STM32G071RB MCUs.

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

a detailed mathematical formulation and solved using three different algorithms, namely FS, DS and B&B. To evaluate the proposed methodology an exhaustive experimental campaign has been carried out on several CNN architectures and heterogeneous MCUs. The obtained results showed that, among the considered algorithms, the B&B achieved the optimal solution in the lowest number of steps in all the experiments. Concerning FS, it turned out to have limitations with deeper network architectures since exploring all the possible candidates requires significant computational effort and may 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, the power consumption could be included in the constraints as well as in the objective function. From the algorithmic point of view, DS can be improved by introducing an adaptive bisection which splits the current candidate based on the network's profiles instead of fixed points. Furthermore, the B&B can be improved by introducing an innovative admissible heuristic for selecting the next layer-device assignment in the throughput maximization problem. Finally, the last open point consists in evaluating this methodology on multi-branches NNs models.

References

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

[2] 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.

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