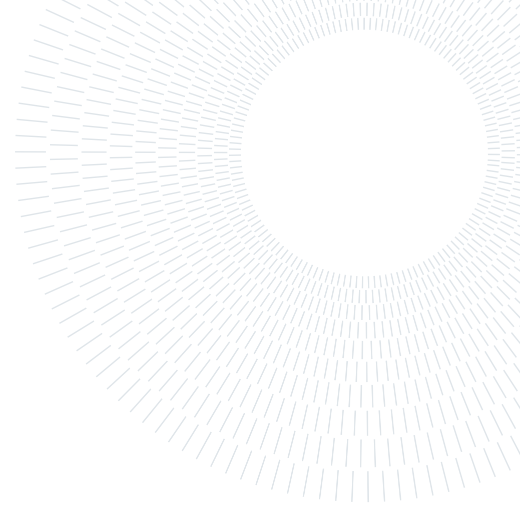


POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Neural Architecture Search for Tiny Incremental On-Device Learning

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Marco Lacava, 962725

Advisor:
Prof. Manuel Roveri

Co-advisors:
Matteo Gambella
Massimo Pavan

Academic year:
2022-2023

Abstract: Incremental on-device learning is a promising research field in TinyML which consists in the ability of models to adapt to new data without constant reliance on cloud connectivity. TyBox is a toolbox for the automatic design and code-generation of incremental on-device TinyML models that addresses this challenge effectively. In parallel, Neural Architecture Search is a powerful AutoML approach which aims to automate the design of a neural architecture optimized on its accuracy and computational requirements for a given task and dataset. Constrained NAS expands on the concept of Neural Architecture Search, incorporating TinyML-inspired constraints to strike a balance between performance and resource efficiency. However, it primarily targets mobile devices with greater computational resources, making it unsuitable for MCUs. The aim of this work is to design a methodology to integrate Neural Architecture Search with tiny incremental on-device learning. At the time of delivery, this is the first time NAS has been considered for incremental on-device learning. The approach involves compressing CNAS-designed networks with structured filter pruning and extending TyBox with full-integer quantization to create tiny incremental models that conforms with the restrictions of tiny devices. Performance tests on multi-image classification tasks demonstrate the effectiveness of the proposed methodology, showcasing the competitiveness of the compressed incremental models compared to the original static CNAS models and alternative incremental toolboxes.

Key-words: ML, TinyML, incremental learning, Constrained NAS, embedded AI

Contents

1 Introduction **2**
1.1 Problem and motivation . . . 2
1.2 Thesis structure . . . 3

2 Background **3**
2.1 Machine Learning overview . . . 3
2.1.1 Deep Learning and Convolutional Neural Networks . . . 4
2.2 TinyML . . . 4
2.2.1 Deep Compression techniques . . . 5
2.2.2 Incremental on-device learning . . . 6

2.3 AutoML . . . 7
2.3.1 Neural Architecture Search . . . 7
2.3.2 One-shot NAS . . . 8
2.3.3 MSUNAS - Evolutionary Multi Objective Surrogate-Assisted NAS . . . 9
2.3.4 MobileNetV3 . . . 11

3 Related Literature **12**
3.1 The TyBox toolbox . . . 13
3.2 Constrained NAS . . . 14

4 Proposed Solution **16**
4.1 PyTorch to Tensorflow model conversion . . . 17
4.2 Structured pruning . . . 17
4.3 Full-integer quantization . . . 18

5 Experimental results **20**
5.1 Dataset . . . 20
5.1.1 CIFAR-10 . . . 20
5.1.2 Imagenette . . . 21
5.2 Baseline model conversion . . . 22
5.3 Proposed methodology . . . 22
5.3.1 Structured pruning . . . 22
5.3.2 TyBox full-integer quantization . . . 24
5.4 Application scenarios . . . 24
5.4.1 Concept drift . . . 26
5.4.2 Incremental learning . . . 26
5.4.3 Transfer learning . . . 30

6 Conclusions **30**

1. Introduction

1.1. Problem and motivation

Machine Learning (ML) is a sub-field of Artificial Intelligence that has revolutionized the way computers learn and make predictions or decisions by extracting patterns and insights from data, without the need for explicit human intervention [46]. Traditional ML algorithms often rely on high computational power and memory resources to process and store the data. While these requirements are feasible in resource-rich environments, they pose substantial difficulties when applied to energy-efficient solutions with constrained resources.
The field of Tiny Machine Learning (TinyML) has emerged as a response to these challenges [47]. TinyML focuses on deploying energy-efficient ML algorithms on small, low-power microcontrollers units (MCUs), which are commonly found in tiny devices, such as embedded systems and Internet-of-Things (IoT) units. One of the most promising research fields in TinyML is incremental on-device learning [17], which consists in the ability of a TinyML model to learn and adapt to new data directly on the device itself, without constant reliance on cloud connectivity or the need to retrain the entire model offline. Given the limited resources available in tiny devices, incremental on-device learning presents a particularly challenging subject. A state-of-the-art solution that successfully addresses this challenge is TyBox, a toolbox for the automatic design and code-generation of incremental on-device TinyML models [38].

Alongside Machine Learning, Automated Machine Learning (AutoML) has emerged as a powerful technique that streamlines the ML pipeline by enabling to automatically build ML applications without much requirement for statistical and ML knowledge [26]. A powerful AutoML approach is Neural Architecture Search (NAS) [41], which aims to automate the design of a neural architecture optimized on its accuracy and computational requirements for a given task and dataset.
Constrained Neural Architecture Search (CNAS) [22] is a novel AutoML technique that expand upon the concept of NAS by integrating TinyML inspired technological constraints. CNAS facilitates the discovery of architectures that balance performance and resource efficiency, thus enabling the development of models that can effectively operate on devices and systems with limited computational power or energy availability. However, the models produced by CNAS predominantly target mobile devices, which possess significantly greater computational resources compared to MCUs. As CNAS was not designed to accommodate the strict technological

requirements of MCUs, the technique is not suitable for deployment on tiny devices.

This work aims at addressing the aforementioned problems by combining CNAS with the severe constraints of MCUs, in order to obtain a model for tiny incremental on-device learning. Our goal is to design a methodology to integrate CNAS with the TyBox toolbox.
In particular, the approach presented in this paper consists of:

- the application of compression techniques to reduce the computational load and memory demand of a CNAS-designed neural network;
- the implementation of an extended version of the TyBox toolbox incorporating full-integer quantization, to design an incremental on-device learning version of the reduced network.

The performance of the resulting incremental model has been tested over a multi-image classification problem across diverse application scenarios. Comparative analyses were conducted against the original CNAS model and an alternate incremental on-device toolbox.

1.2. Thesis structure

The thesis is organized as follows. Chapter 2 introduces the notion of ML, giving a basic background of deep learning. In addition, some general information about TinyML and AutoML are given, since they are fundamental to fully understand the concepts discussed in this work. Chapter 3 provides a formal description of CNAS and TyBox, along with an analysis of some state-of-the-art algorithms. Chapter 4 describes the proposed solution, providing an in-depth description of our pipeline and highlighting the techniques applied at each stage. Chapter 5 details the results achieved by the proposed methodology and reports the experiments conducted on a multi-class image classification scenario. In Chapter 6, conclusions and future works are finally drawn.

2. Background

In this chapter, we introduce fundamental concepts that serve as the foundation for the topics discussed in the following chapters. Section 2.1 explores the subfield of AI known as Machine Learning, emphasizing its novelty with respect to traditional engineering methods for algorithm development. Additionally, we delve into Deep Learning (DL), a widely popular subset of ML, and Convolutional Neural Networks (CNN) for image processing. Section 2.2 discusses the domain of TinyML, which offers efficient processing solutions in the field of ML. This section highlights the rationale behind TinyML, its inherent advantages, and the key techniques associated with it, such as pruning and quantization. Section 2.3 provides an introduction to AutoML, an ML branch that aims at reducing the onerous development costs by automating the entire ML pipeline. This section particularly focuses on prominent solutions for automating the creation of efficient neural architectures, including Neural Architecture Search and Evolutionary Multi-Objective Surrogate-Assisted NAS (MSUNAS).

2.1. Machine Learning overview

Machine Learning [46] is a branch of AI that enhances computers with the ability to learn from data, adapt to changing circumstances, and improve their performance over time. Unlike conventional engineering approaches where explicit rules and algorithms are painstakingly designed, ML systems are trained to recognize patterns and relationships within data, enabling them to make predictions, classify information, and make informed decisions. In conventional engineering approaches, solutions are typically crafted through a meticulous process of rule-based design. Engineers prescribe explicit instructions and logic to accomplish tasks or solve problems. These systems often struggle when confronted with the complexity of real-world data, as it can be challenging to anticipate and encode all possible variations and interactions. Machine Learning, in contrast, harnesses the power of data to generalize from examples, allowing systems to adapt and perform effectively in dynamic and unstructured environments.
A prominent class of Machine Learning techniques is the one of Artificial Neural Networks (ANNs) [30]. These networks mimics the structure and function of the human brain. ANNs consist of interconnected nodes, or artificial neurons, organized in layers. Data is processed through these neurons, with each neuron applying a mathematical operation. ANNs can be shallow with only a few layers or deep with multiple layers, depending on the complexity of the task at hand.

2.1.1 Deep Learning and Convolutional Neural Networks

Deep Learning [30] represents a subset of Machine Learning characterized by the use of Deep Neural Networks (DNNs). DNNs are structured into distinct layers: an Input Layer, Hidden Layers, and an Output Layer. The Input Layer serves as the entry point for raw data, where features are ingested and propagated through the network. Hidden Layers, located between the Input and Output Layers, conduct complex transformations and hierarchical feature extraction and enable DNNs to model intricate patterns and abstractions within data. Finally, the Output Layer produces the network's predictions or classifications, providing insights or decisions based on the input data's processed information. Deep Learning has led to remarkable breakthroughs in various domains, including image and speech recognition, natural language processing, and autonomous systems. It thrives on vast datasets and powerful computational resources, facilitating the development of models with superior performance.

In Deep Learning, Convolutional Neural Networks (CNNs) specialize in tasks involving grid-like data, such as images and videos [9]. At the heart of CNNs is a unique architectural design that leverages the concept of convolution. Convolution is a mathematical operation that involves passing a small filter or kernel over the input data to detect local patterns and features. These kernels are learnable parameters within the CNN, and their convolution with the input data results in feature maps that highlight specific characteristics of the input. CNNs typically comprise several layers, each with a specific function:

- Convolutional Layers, which perform the convolution operation with learnable filters. They are responsible for detecting low-level features, such as edges and textures;
- Pooling Layers, which downsample feature maps, reducing their spatial dimensions while retaining important information. Common pooling operations include max-pooling and average-pooling;
- Fully Connected Layers, that connect every neuron to every neuron in the previous and subsequent layers, allowing the network to learn complex relationships in the data.

One of the key strengths of CNNs lies in their ability to automatically extract hierarchical features from data. As data flows through the network, each layer learns to recognize increasingly complex and abstract features. For example, in image classification, early layers might detect edges and corners, while deeper layers identify more complex shapes and object parts. This hierarchical feature extraction is critical for understanding the semantics of the input data. CNNs have revolutionized computer vision, enabling applications like image classification, object detection, and image generation. CNNs excel at categorizing images into predefined classes, making them the go-to choice for tasks like recognizing objects in photographs. They are used for locating and classifying objects within images or videos, enabling applications like autonomous driving and surveillance. Moreover, CNNs can also generate new images, whether it's enhancing image quality, applying artistic styles, or creating entirely new content.

2.2. TinyML

Tiny Machine Learning (TinyML) [47] represents a convergence of Machine Learning and embedded systems, enabling intelligent decision-making directly on edge devices. The rise of the Internet of Things (IoT) has ushered in a new era of interconnected devices, encompassing everything from environmental sensors and wearables to industrial machinery and autonomous vehicles. These devices generate vast amounts of data at the edge of the network, demanding efficient, localized processing capabilities. This is where TinyML excels, as it empowers these devices to perform essential processing activities locally, reducing latency and dependency on cloud resources. TinyML is, at its essence, a specialized subset of ML designed to thrive in resource-constrained environments. It acknowledges the limitations of edge devices, including limited memory, processing power, and energy resources, and adapts ML techniques to make them feasible in such settings. This is in stark contrast to traditional ML, which often relies on the computational resources of powerful servers or cloud infrastructure.

Employing the localized processing of activities, TinyML brings several advantages:

- Latency reduction: in IoT applications where real-time or near-real-time decision-making is paramount, TinyML's capacity to process data directly on edge devices becomes critical. By doing so, it significantly mitigates the latency associated with sending data to remote servers for analysis. In autonomous vehicles, for instance, the ability to make split-second decisions without relying on a distant data center can be a matter of life and death.
- Privacy and Security enhancement: TinyML reinforces privacy and security by keeping sensitive data within the confines of edge devices. Data remains localized, reducing exposure to potential breaches during data transmission to remote servers. This is particularly vital in applications like healthcare, where patient data must be safeguarded.
- Bandwidth efficiency: transmitting raw sensor data to the cloud can be bandwidth-intensive and costly,

especially in applications with numerous devices. TinyML's on-device processing conserves network bandwidth and lowers operational costs by reducing the volume of data that needs to be sent to remote servers. In remote monitoring systems or industrial automation, this can lead to significant cost savings.

- Reliability: relying on an internet connection is often not a viable option. By enabling ML models to operate directly on edge devices, TinyML mitigates the dependence on cloud-based services and remote servers, thus reducing vulnerability to network outages and latency.

2.2.1 Deep Compression techniques

While the potential advantages of TinyML are substantial, executing Machine Learning models on resource-constrained edge devices presents unique challenges. Deep Learning models, known for their complexity and resource demands, are often too large to fit on these devices. This is where deep compression techniques, such as pruning and quantization, come into play. Compression techniques are pivotal in enabling the deployment of efficient ML models in TinyML scenarios. The choice between the diverse forms of pruning and quantization necessitates a thorough understanding of the trade-offs between efficiency and accuracy, ensuring that models perform optimally on resource-constrained edge devices. The following sections describe the compression techniques implemented in our methodology.

Structured pruning

Pruning is a powerful technique employed in the domain of neural networks that focuses on the surgical removal of weight connections within a model [12]. The fundamental goal of pruning is to strip away non-essential components of the neural network, thereby achieving a streamlined, more efficient model, all while preserving its predictive prowess. This technique is particularly valuable in the context of resource-constrained environments, such as those encountered in TinyML, where computational resources are sparse, and model size plays a critical role. Neural networks, due to their dense and interconnected nature, often contain numerous weight connections that do not contribute significantly to the model's overall performance. Pruning identifies and prunes away these non-essential connections, resulting in a more concise and computationally efficient model. Pruning can take on two distinct forms: unstructured and structured. Structured pruning focuses on the removal of entire neurons, channels, or layers from the neural network. This method maintains the network's architectural integrity by retaining a predefined structural pattern. For instance, structured pruning might involve the elimination of complete convolutional filters in a Convolutional Neural Network (CNN) or entire neurons in a Feed-Forward network [43]. It effectively reduces the model's size while preserving its fundamental architecture. Unstructured pruning [25], in contrast, operates at a more granular level. It involves the removal of individual weights within the neural network by converting them to zero, irrespective of their location or relationship within the architecture. Unstructured pruning offers a higher degree of flexibility by considering each weight's importance independently. However, it can result in a model with an irregular and less predictable structure, as weights are pruned without regard for architectural patterns.

The structured pruning technique proposed in [32] introduces an innovative technique for filter pruning in CNNs. The key steps presented include:

- **Filter Ranking:** The first step in the filter pruning process is to rank the filters based on their importance. This ranking metric is devised to evaluate the impact of each filter on the overall network output. Filters that contribute the least to the network's performance are identified as candidates for pruning. The ranking is crucial in determining which filters can be pruned without significant loss of accuracy.
- **Iterative and Gradual Pruning:** Rather than removing filters abruptly, an iterative and gradual pruning approach is introduced. Filters are pruned incrementally over multiple iterations. In each iteration, a certain percentage of the least important filters are pruned. This gradual approach allows the network to adapt to the changing architecture and recover any lost accuracy through fine-tuning.
- **Fine-Tuning:** After each round of filter pruning, the CNN undergoes a fine-tuning process. Fine-tuning is essential to restore and potentially enhance the model's accuracy. During this phase, the pruned network is trained on the original dataset, with a focus on the remaining filters. Fine-tuning ensures that the model retains its ability to capture essential features and patterns in the data.
- **Pruning Threshold:** the pruning threshold determines the proportion of filters to be pruned in each iteration. This threshold can be set based on user-defined criteria, such as a desired model compression ratio or a target level of computational efficiency. Adjusting the pruning threshold allows for control over the trade-off between model size reduction and accuracy preservation.
- **Impact Analysis:** the importance of analyzing the impact of filter pruning on model accuracy and efficiency. It evaluates the trade-offs between computational cost reduction and model degradation, providing insights into the effectiveness of the proposed technique.

Full-integer quantization

Quantization revolves around the concept of representing numerical values in a more compact form, typically by reducing the bit-width or precision of model parameters [23]. In conventional Deep Learning, parameters are typically stored as 32-bit floating-point numbers. In quantization, these parameters are converted to lower bit-width representations, such as 8-bit integers. This reduces the memory and computational requirements, enabling models to run efficiently on resource-constrained hardware.

In the context of quantization, two primary approaches emerge: post-training quantization and quantization-aware training. The post-training quantization approach involves quantizing an already trained model. After training a full-precision model, quantization techniques are applied to convert its weights and activations to lower bit-width representations. While this method is straightforward and does not require changes to the training process, it may not fully exploit the potential for accuracy preservation during quantization. Quantization-aware training, in contrast, integrates quantization considerations during the model's training phase. The model is trained with quantization constraints, ensuring that the weights and activations are learned in a quantization-friendly manner. This approach often leads to more accurate quantized models, as the network adapts to the limitations imposed by lower bit-width representations from the outset.
Additionally, quantization techniques encompass a spectrum of strategies, ranging from full-integer quantization to more intricate approaches that involve fractional values and dynamic scaling [37]. Full-integer quantization is the most straightforward technique, where model parameters are quantized to integer values. For example, 32-bit floating-point numbers may be quantized to 8-bit integers. While this approach simplifies hardware implementation, it can result in a significant loss of precision and, subsequently, model accuracy. Fixed-point quantization allows for a mix of integer and fractional parts in the quantized values. This hybrid representation retains more precision than full-integer quantization but comes at the cost of increased complexity in hardware and software. Dynamic quantization adapts the quantization scale for each layer or parameter based on the data distribution. This dynamic scaling approach mitigates some of the accuracy loss seen in fixed-point and full-integer quantization but requires additional computational resources for scaling.

2.2.2 Incremental on-device learning

The conventional model development workflow in TinyML assumes a technological decoupling between training and inference phases. Training requires a large computational load and memory demand that often exceed the capabilities of tiny devices. Most of the solutions present in the literature assume that devices only support the inference of ML models, while the training is carried out in the Cloud where appropriate computing and memory resources are available. This separation has facilitated the adoption of machine learning in resource-constrained environments like IoT and edge computing.
While the decoupling of training and inference simplifies the resource constraints on edge devices, it presents several challenges. Decoupled models rely on a constant connection to the cloud for updates and refinements. This reliance on network connectivity introduces latency, making it unsuitable for applications requiring real-time or near-real-time decision-making. Furthermore, these models are rendered ineffective in scenarios with limited or intermittent connectivity. Cloud-based training requires data to be transmitted to remote servers, potentially raising privacy and data security concerns. Sensitive information may be exposed during data transfer, which is particularly problematic in applications such as healthcare or surveillance. Moreover, separating training and inference prevents TinyML solutions to be incrementally trained or adapted during their operational life by exploiting fresh information coming from the field.
For these reasons, incremental on-device learning represents a paradigm shift within TinyML, addressing the limitations of the decoupled training-inference model. Instead of viewing model development as a one-time, off-device endeavor, incremental on-device learning enables edge devices to adapt and fine-tune models over time, directly onboard the device itself. The incremental approach allows (i) to make TinyML solutions adaptive over time to manage additional tasks while they are operating (e.g., a new gesture-recognition command should be learned by the TinyML applications), (ii) to support the fine-tuning of TinyML models on specific users or settings (e.g., a set of vocal commands is pre-trained on generic users, while the final user fine-tunes the model with his/her own specific voice), and (iii) to deal with concept drift in the process generating the data (e.g., a person-recognition TinyML devices, trained to operate in indoor conditions, is moved outdoor).

Very few solutions for on-device training of TinyML models are available in the literature. These solutions can be grouped into two main families according to the type of TinyML models supporting the on-device training: ML and DL models.
Existing literature on on-device training solutions for TinyML machine learning models predominantly emphasizes incremental learning and concept drift. For example, [18] proposed a solution addressing the concept drift problem and adapting the algorithms on the basis of new knowledge, by integrating a deep convolutional feature

extractor and a K-NN classifier. Due to the nature of the K-NN algorithm, the adaptation consists in adding the new extracted features-label pairs to the training dataset. The same approach was used in [17] to address the incremental learning scenario.
The on-device training solutions for TinyML deep learning models present in the literature mainly focus on transfer learning. [39] and [40] suggest a solution where the feature extraction part of their networks is kept fixed and only the last layer is retrained. Extending this concept, TyBox [38] introduces an innovative framework that incorporates a buffer for facilitating incremental training of the classification layers within the model. An in-depth analysis of TyBox can be found in Section 3.1
As of now, none of the incremental TinyML solutions documented in the existing literature have been specifically conceived for integration into Neural Architecture Search [41]. This is primarily due to the fact that NAS typically generates models with memory and storage requirements that are orders of magnitude greater than those compatible with embedded systems or IoT devices.

2.3. AutoML

Automated Machine Learning (AutoML) stands as a transformative innovation in the field of machine learning, offering a streamlined approach to model development and optimization [26]. It aims to automate various aspects of the machine learning pipeline, from data preprocessing and feature engineering to model selection, hyperparameter tuning, and even model deployment. It democratizes machine learning by enabling non-experts to harness the power of ML without requiring an in-depth understanding of its intricacies. AutoML leverages sophisticated algorithms and optimization techniques to find the best models and hyperparameters, all while minimizing manual intervention.
AutoML introduces several noteworthy advantages over traditional manual ML practices. Firstly, AutoML accelerates the model development process dramatically. Automating repetitive and time-consuming tasks allows one to focus on higher-level tasks like problem formulation and domain expertise. Secondly, AutoML opens the doors to machine learning for a broader audience. Its user-friendly interfaces and automation capabilities reduce the steep learning curve traditionally associated with ML, enabling domain experts and non-experts alike to harness the power of data-driven decision-making. Lastly, AutoML systematically explores a wide range of models, hyperparameters, and preprocessing techniques. This exhaustive search often leads to models that outperform those manually created, as it can discover subtle interactions and optimizations that might be missed by human intuition alone.

2.3.1 Neural Architecture Search

A particularly promising family of algorithms belonging to AutoML is the Neural Architecture Search (NAS). NAS leverages the same principles of automation but applies them specifically to the creation of neural networks, which are the building blocks of many ML and DL applications. By automating the design process, NAS reduces the need for manual architectural engineering, making deep learning accessible to a broader audience and accelerating the development of high-performing models.

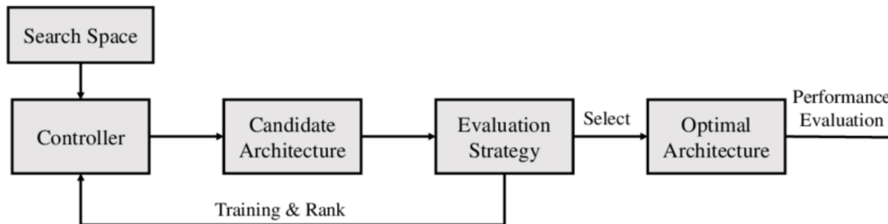


Figure 1: A general framework of NAS.

NAS algorithms are categorized according to three components:

- **Search Space**, which defines the set of possible architectures to be considered during the search. It defines the possible combinations and configurations of layers, connections, and operations within a neural network and spans from simple architectures, like FFNNs [43], to intricate and deeply nested structures, such as CNNs. The search space can also incorporate architectural innovations, such as skip connections, attention mechanisms, or specialized layers. The richness of this space allows NAS to explore a vast landscape of possibilities, seeking architectures that optimize specific objectives, be it accuracy, efficiency, or other performance metrics;
- **Search Strategy**, which establishes how the architectural search space is traversed to find optimal or near-optimal network architectures. NAS employs various search algorithms, ranging from Evolutionary

algorithms (EA) and Reinforcement Learning (RL) to Gradient-based optimization methods. Each strategy has its unique characteristics. EAs mimic the principles of natural selection by evolving a population of architectures over generations, while RL formulates the architecture search as a Markov Decision Process, with an agent learning to select architectural components iteratively. Additionally, gradient-based methods optimize architectures by continuously updating their parameters, requiring a gradient estimator to assess architectural changes' impact. The choice of search strategy profoundly influences NAS's efficiency and effectiveness in discovering high-performing architectures;
- **Performance Estimation Strategy**, which refers to the process of estimating the performance to be optimized. Performance estimation strategy is pivotal in guiding the search process, allowing NAS to let go unpromising architectures and focus computational resources on those with the potential for superior performance.

A general framework of NAS is described in [41] and shown in Fig. 1. The procedure explores a search space of predefined operations and their connections. Then, a controller selects the pool of possible candidate architectures from the search space. The candidate architectures are trained and ranked based on their performance on the validation set. The ranking of the candidate architectures is used to calibrate the search strategy and generate new candidates. The process iterates until reaching a stop criterion and provides the optimal architecture. Finally, the optimal architecture performance is evaluated on the validation set.

The literature contains a limited number of solutions that implements NAS for incremental learning. In [44], an incremental variant of DARTS is introduced. This solution incorporates prediction space regularization and knowledge distillation loss within the NAS framework. These measures are applied to encourage the learning of new class with minimal degradation to existing class knowledge. Furthermore, the approach incorporates class-balanced fine-tuning of the classification head for previously learned classes, in order to mitigate the natural tendency of models to exhibit bias towards more recent classes. It is worth noting that, while some NAS-based solutions have addressed the problem of incremental learning, currently, no incremental *on-device* learning NAS solutions are available in the literature.

2.3.2 One-shot NAS

A prominent family of NAS solutions is the so-called One-Shot NAS, that relies upon a Once-For-All (OFA) supernet, a network that encapsulates many possible architecture configurations. The key advantage of OFA is the decoupling of the search and training phases: instead of training each architecture individually, only the supernet is trained once on a big dataset like Imagenet [16] and then the candidates' architecture weights and performances are obtained simply by inference on a different dataset, without the requirement of additional training. The inference is performed by selecting only part of the OFA supernet. It flexibly supports different depths, widths, kernel sizes, and resolutions without retraining. The computational burden is thus shifted to the construction of the supernet, which is built only once before the search.

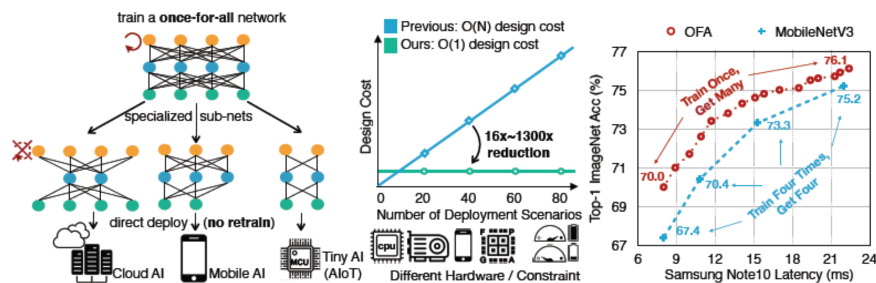


Figure 2: Left: a single once-for-all network is trained to support versatile architectural configurations including depth, width, kernel size, and resolution. Given a deployment scenario, a specialized subnetwork is directly selected from the once-for-all network without training. Middle: this approach reduces the cost of specialized deep learning deployment from O(N) to O(1). Right: once-for-all network followed by model selection can derive many accuracy-latency trade-offs by training only once, compared to conventional methods that require repeated training. [11]

Training the once-for-all network can be cast as a multi-objective problem, where each objective corresponds to one sub-network. With such a large number of sub-networks that share weights, thus interfering with each other, trying to simultaneously optimize each sub-network is a computationally prohibitive approach to adopt.

In response to this challenge, the work [11] proposes a progressive shrinking (PS) algorithm for training the once-for-all network. Rather than directly optimizing the once-for-all network from scratch, PS first trains the largest neural network, characterized by maximal depth, width, and kernel size, then progressively fine-tunes the once-for-all network to support smaller sub-networks that share weights with the larger ones. This approach yields several advantages, including the provision of a initialization by selecting the most important weights of larger sub-networks. Additionally, it affords the opportunity to distill smaller sub-networks, which greatly improves the training efficiency.

2.3.3 MSUNAS - Evolutionary Multi Objective Surrogate-Assisted NAS

Typically, NAS solutions based on OFAs aim at optimizing only the classification accuracy of the designed neural networks. Differently, Hardware-Aware NAS takes into account other figures of merit such as computational complexity or memory demand. An example is Evolutionary Multi Objective Surrogate-Assisted NAS (MSuNAS) [34]. NAS is typically treated as a bi-level optimization problem, where an inner optimization loops over the weights of the network for a given architecture, while the outer optimization loops over the possible network architecture configurations in the search space. Learning the optimal weights of the network in the lower level necessitates costly iterations of stochastic gradient descent. Similarly, exhaustively searching the optimal architecture is prohibitive due to the discrete nature of the architecture description, size of search space and our desire to optimize multiple, possibly competing, objectives. To obtain a significant speed up in the outer optimization, MSuNAS makes use of a surrogate accuracy predictor, that significantly accelerates the evaluation of each candidate architecture during the search process avoiding its training. MSuNAS is an OFA-based NAS [11], but differently from other OFA algorithms, which obtain the weights of the candidate architectures directly from the supernet, MSuNAS uses the weights inherited from the supernet only as an initialization to the lower level optimization. Such a fine-tuning process affords the computation benefit of the supernet, while at the same time improving the correlation in the performance of the weights initialized from the supernet and those trained from scratch. The training is performed not for all samples, but only for the samples which are close to the current trade-off in the search space. As a result the computational overhead is reduced.

NSGA-II

MSuNAS is based on NSGA-II (Non-dominated Sorting Genetic Algorithm-II), an evolutionary algorithm belonging to the multi-objective optimization family [15]. Multi-objective optimization algorithms involve more objective functions at the same time and try to find the most suitable trade-off between two or more conflicting objectives. The method is applied to the upper level surrogate predictor which is learnt in an online manner during the search, start the construction of the accuracy predictor from only a limited number of architectures sampled randomly from the search space. Given this surrogate and another objective the algorithm generates the new candidate architectures.

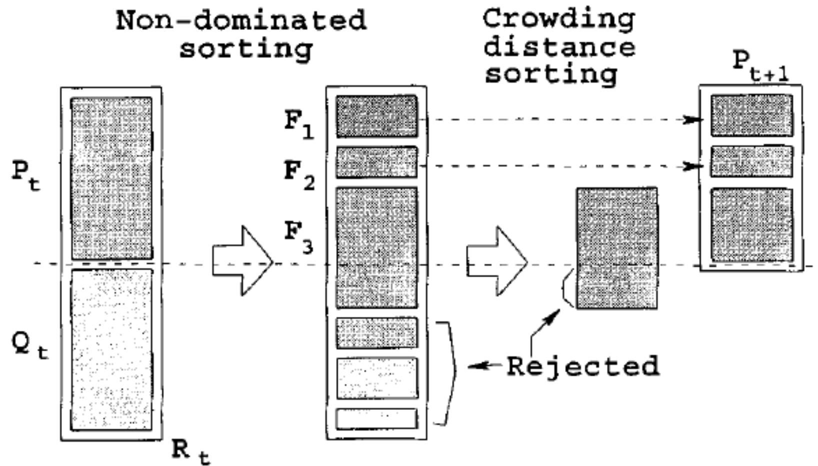


Figure 3: NSGA-II procedure [15]

NSGA-II incorporates the concept of non-dominated sorting. At the core of non-dominated sorting is the concept of Pareto dominance. In multi-objective optimization, a solution is considered Pareto-dominant over another if it is at least as good as the other solution in all objectives and strictly better in at least one objective.

In other words, a solution A dominates another solution B if it offers an improvement in at least one objective without worsening any other objective and can be expressed as follows:

$$A(x1|y1) \text{ dominates } B(x2|y2) \text{ when } (x1 <= x2 \text{ and } y1 <= y2) \text{ and } (x1 < x2 \text{ or } y1 < y2)$$

Non-dominated sorting organizes solutions into fronts or layers based on their Pareto dominance relationships [1]. The first front consists of solutions that are not dominated by any other solution in the population. The second front comprises solutions that are dominated only by solutions in the first front. This process continues, with each subsequent front containing solutions that are dominated only by solutions in previous fronts. Within each front, solutions are ranked based on their crowding distance. Crowding distance measures how densely solutions are packed in the objective space. Solutions with higher crowding distances are preferred because they contribute more to maintaining diversity in the population. Once non-dominated sorting is complete, NSGA-II uses the ranked fronts and crowding distances to select solutions for the next generation.

$R_t = P_t \cup Q_t$	combine parent and offspring population
$\mathcal{F} = \texttt{fast-non-dominated-sort}(R_t)$	$\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \ldots)$, all nondominated fronts of R_t
$P_{t+1} = \emptyset$ and $i = 1$	
until $\lvert P_{t+1}\rvert + \lvert\mathcal{F}_i\rvert \leq N$	until the parent population is filled
$\quad\texttt{crowding-distance-assignment}(\mathcal{F}_i)$	calculate crowding-distance in $\mathcal{F}_i$
$\quad P_{t+1} = P_{t+1} \cup \mathcal{F}_i$	include ith nondominated front in the parent pop
$\quad i = i + 1$	check the next front for inclusion
Sort$(\mathcal{F}_i, \prec_n)$	sort in descending order using $\prec_n$
$P_{t+1} = P_{t+1} \cup \mathcal{F}_i[1 : (N - \lvert P_{t+1}\rvert)]$	choose the first $(N - \lvert P_{t+1}\rvert)$ elements of $\mathcal{F}_i$
$Q_{t+1} = \texttt{make-new-pop}(P_{t+1})$	use selection, crossover and mutation to create a new population Q_{t+1}
$t = t + 1$	increment the generation counter

Figure 4: NSGA-II pseudocode [15].

The NSGA-II main loop consists of three steps:

- Create offspring and a combined population Rt = Pt + Qt where Pt is the best offspring and Qt is the new offspring generated;
- Rank and sort offspring due to performance on defined target indexes;
- Take best members to create new population including a good spread in solutions. Pt+1 needs to have the same size of population as Pt. Rt is divided in fronts which shares same indexes. Less fronts mean better performance.

The generation of new offsprings involves tournament selection, crossover, and mutation. During tournament selection a subset of solutions is chosen from the current population to form the parents for the next generation. The key idea is to favor individuals that are non-dominated within their local neighborhoods. This process ensures that a diverse set of non-dominated solutions is preserved from one generation to the next. Then, during crossover, pairs of parent solutions are selected from the chosen subset, and their genetic information is combined to generate new offspring solutions. The crossover operator promotes the creation of offspring solutions that inherit good traits from both parents while maintaining diversity within the population. Finally, the mutation step introduces further diversity by making small, random perturbations to some of the newly generated offspring solutions. Mutation helps the algorithm escape local optima and explore regions of the solution space that may otherwise be missed.

MSuNAS algorithm description

The MSuNAS algorithm is shown in Fig. 5 and a brief description is presented in the following lines.
Lines 1 - 7: MSUNAS firstly initializes an empty archive to store all trained CNNs (line 1). Then it samples randomly N architectures from the search space (line 2), and for each architecture evaluates their top1 and complexity (lines 4-5) and then stores the evaluated/trained architecture in the archive (line 6). These steps can be skipped if the algorithm resumes from a specific iteration collecting the architectures from the archive instead of filling it like in the previously described.
Lines 8 – 18: At each iteration, accuracy prediction surrogate models Sf are constructed from an archive of previously evaluated architectures (line 9) through a selection mechanism, dubbed Adaptive Switching (AS), which constructs four types of surrogate models at every iteration and adaptively selects the best model according to a correlation metrics called Kendall's Tau [31]. These four different surrogates for accuracy prediction come from the literature : namely, Multi Layer Perceptron (MLP) [33], Classification And Regression Trees (CART) [45], Radial Basis Function (RBF) [10] and Gaussian Process (GP) [14].

Algorithm 1: MSuNAS

Input : $\mathcal{SS}$ (search space), $\mathcal{S}_w$ (supernet), $\mathcal{C}$ (complexity obj), N (initial samples), K (max. iterations).

1 $\mathcal{A} \leftarrow \emptyset$;
2 **while** $i < N$ **do**
3 $\quad \alpha \leftarrow$ sample($\mathcal{SS}$)
4 $\quad w_o \leftarrow \mathcal{S}_w(\alpha)$
5 $\quad acc \leftarrow$ SGD(α, w_o)
6 $\quad \mathcal{A} \leftarrow \mathcal{A} \cup (\alpha, acc)$
7 **end**
8 **while** $j < K$ **do**
9 $\quad \mathcal{S}_f \leftarrow$ construct from $\mathcal{A}$ // (MLP / CART / RBF / GP)
10 $\quad \tilde{\boldsymbol{\alpha}} \leftarrow$ NSGA-II($\mathcal{S}_f, \mathcal{C}$)
11 $\quad \boldsymbol{\alpha} \leftarrow$ subset from $\tilde{\boldsymbol{\alpha}}$
12 $\quad$ **for** α *in* $\boldsymbol{\alpha}$ **do**
13 $\quad\quad w_o \leftarrow \mathcal{S}_w(\alpha)$
14 $\quad\quad acc \leftarrow$ SGD(α, w_o)
15 $\quad\quad \mathcal{A} \leftarrow \mathcal{A} \cup (\alpha, acc)$
16 $\quad$ **end**
17 **end**
18 **Return** NDsort($\mathcal{A}$).

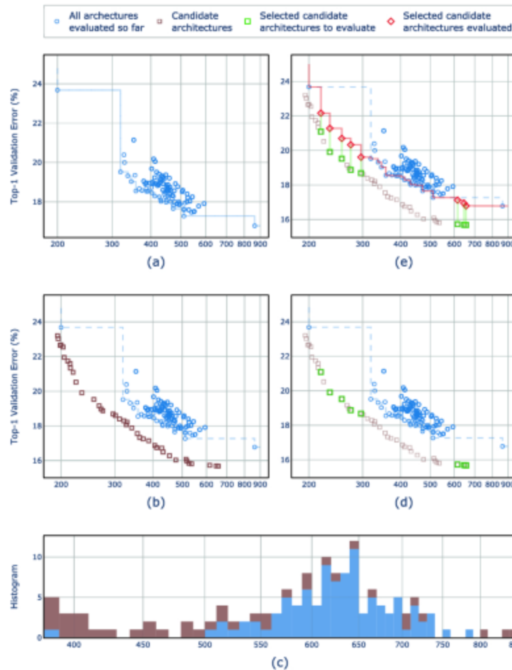


Figure 5: A sample run of MSuNAS on ImageNet: In each iteration, accuracy-prediction surrogate models Sf are constructed from an archive of previously evaluated architectures (a). New candidate architectures (brown boxes in (b)) are obtained by solving the auxiliary single-level multi- objective problem (line 10 in Algo 1). A subset of the candidate architectures is chosen to diversify the Pareto front (c) - (d). The selected candidate architectures are then evaluated and added to the archive (e). After search, we report the nondominated architectures from the archive. The x-axis in all sub-figures is the #MAdds.

New candidate architectures are obtained by solving the auxiliary single-level multi-objective problem NSGA-II(Sf,C) where NSGA-II is a standard multi-objective algorithm which takes in input the surrogate model Sf obtained by the Adaptive Switching and the other objective in interest to the user (line 10).
A subset of the candidate architectures is chosen to diversify the Pareto front (line 11).
The selected candidate architectures are then evaluated and added to the archive (lines 12-16).
At the end of the loop (so when the maximum number of iterations is reached) the algorithm outputs the non-dominated solutions from the pool of evaluated architectures. (line 18).

2.3.4 MobileNetV3

MobileNetV3 is the third iteration of the MobileNet series of neural network architectures designed specifically for mobile and embedded devices. It represents a significant advancement in the field of efficient deep learning models, offering a balance between model size, computational efficiency, and high performance.
MobileNetV3 was developed by Google's research team and introduced as a solution to the increasing demand for deep learning models that can run efficiently on resource-constrained devices, such as smartphones, IoT devices, and edge computing platforms. MobileNetV3 builds upon the success of its predecessors, MobileNetV1 [28] and MobileNetV2 [42], by introducing innovative architectural features and optimizations.
MobileNetV3 is characterized by a unique blend of innovative layers and optimizations that push the boundaries of performance while maintaining a minimal computational footprint.

Inverted Residual Block

At the heart of MobileNetV3's architecture are the Inverted Residual Blocks, which provide a powerful framework for building highly efficient deep neural networks. Fundamental components of Inverted Residual Blocks are Depthwise separable convolutions [13] and Squeeze-and-Excitation (SE) modules [29].
Depthwise separable convolutions break down the convolution operation into two steps: Depthwise convolutions (DC) and Pointwise convolutions (PC).

- Depthwise convolutions focus on applying a filter on each input channel independently. Rather than having a single convolutional kernel for each input channel, as in traditional convolutions, DCs employ

a separate 3x3 convolutional kernel for each input channel. This reduces computational redundancy and significantly lowers the number of parameters. The result is a set of feature maps that capture spatial information effectively.

- Pointwise convolutions follow depthwise convolutions. Since DC is only used to filter the input channel, it does not combine them to produce new feature vectors. They aim to create new features through linear combinations of the filtered channels produced by DCs. PCs employ a small kernel size (1x1) to perform channel-wise linear transformations. This step increases the network's capacity for modeling complex relationships between features while keeping computational costs low.

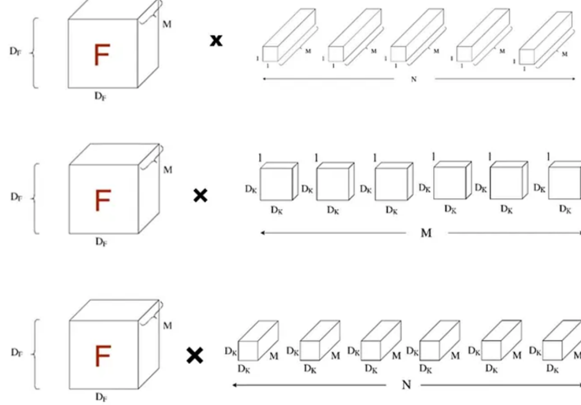


Figure 6: The different types of convolutions. From top to bottom: standard, depthwise, pointwise.

The adoption of depthwise separable convolutions leads to a reduction in computation. As shown in Fig. 7, the computation cost of a standard convolution is:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

where D_F is the special dimensions of the input feature map and D_K is the size of the convolution kernel. M and N are the number of input and output channels respectively. Conversely, the cost of a depthwise separable convolution can be expressed as:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + \cdot M \cdot N \cdot D_F \cdot D_F$$

The first factor refers to the cost of Depthwise convolution, while the second one refers to the cost of Pointwise convolution. Comparing the computational cost of the two convolution methods, we get a reduction in computation of:

$$\frac{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

The Squeeze-and-Excitation (SE) module can be optionally added after the depthwise separable convolution. The SE module is designed to recalibrate channel-wise feature responses, emphasizing informative channels and suppressing less relevant ones within feature maps. The SE module operates in two distinct phases: squeezing and exciting.

In the squeezing phase, the SE module applies global average pooling to the feature maps generated by a preceding convolutional layer. Global average pooling computes the average activation value for each channel over the entire spatial dimension of the feature maps. This process results in a channel-wise summary, effectively capturing the importance or relevance of each channel's features across the entire input.

Following the squeezing phase, the SE module employs two fully connected layers to model and enhance channel-wise dependencies. The excitation phase adapts the contribution of each channel based on its importance, enabling the network to focus on the most informative features.

The output of the SE module is applied directly to its input by a simple element-wise multiplication, which scales each channel/feature map in the input tensor with it's corresponding learned weight from the the SE module.

3. Related Literature

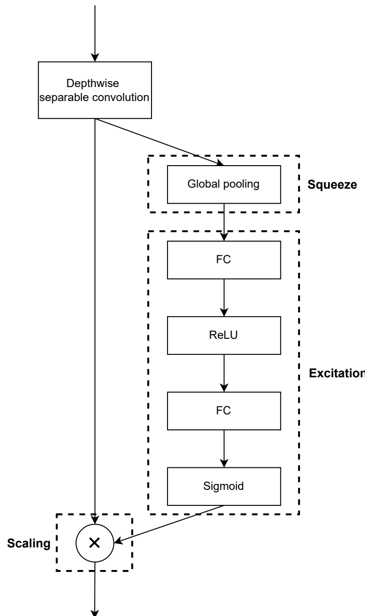


Figure 7: A schema of the SE module implemented in MobileNetV3.

3.1. The TyBox toolbox

TyBox is a state-of-the-art toolbox for the automatic design and code-generation of incremental on-device TinyML models [38]. It fills a critical void in the existing literature. Although incremental TinyML solutions have been emerging, none have featured a comprehensive toolbox that fully supports the automated processes of model design and code generation. TyBox not only bridges this technological gap but also extends its functionality to encompass multi-class classification problems, a domain not yet addressed by incremental TinyML solutions, which have primarily focused on binary classification problems. TyBox supports two different types of TinyML models, i.e., Feed-Forward Neural Networks (FFNNs) [43] and Convolutional Neural Networks (CNNs) [9], representing two well-known and widely used models in the field of machine learning and deep learning. Currently, TyBox is designed to receive in input the TinyML model in Tensorflow (TF) file format [8], since Tensorflow Lite for Micro (TFLM) represents one of the most widely used framework for TinyML.
TyBox is designed to receive in input a standard TinyML model $y = \Phi(I)$, where I is the input and y is the output, and the technological constraint M on the on-device RAM memory that must be satisfied by the designed incremental TinyML solution (on both inference and training phases). An overview of the TyBox toolbox is provided in Fig. 8. TyBox comprises the following two modules: the automatic incremental design module and the automatic code-generation module.

The automatic incremental design module operates by taking $\Phi(\bullet)$ and M as inputs and automatically designing $\Omega(\bullet)$, the incremental version of $\Phi(\bullet)$, composed by a fixed feature extraction block $\Phi_f(\bullet)$ (when dealing with CNNs), an incrementally learnable classification block $\Phi_c(\bullet)$ and a buffer B. Two main actions are carried out by this module: firstly, it divides $\Phi(\bullet)$ into $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$; secondly, it calculates the optimal size for the buffer B to maximize the amount of stored data while respecting the constraint on the RAM memory M.
$\Phi(\bullet)$ is partitioned in two components: the feature extractor $\Phi_f(\bullet)$ and the classifier $\Phi_c(\bullet)$. $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ are assumed to be separated by a Flatten layer, which is the point of the model where $\Phi_f(\bullet)$ is split. This division allows for a reduction in both memory usage and computational demands, as only the classifier $\Phi_c(\bullet)$ is retrained on-device. The drawback of such an approach is that $\Phi_f(\bullet)$ remains static and is not incrementally trained over time. This drawback is partially mitigated by the fact that the first convolutional layers of a CNN are typically characterized by coarse-grain extracted features and these features can be considered general-purpose feature extractors that can be applied to different CNN tasks.
The purpose of the buffer B is to store supervised samples acquired from the field. Its primary function is to mitigate the phenomenon known as "catastrophic forgetting" [36], a phenomenon that entails the deletion of previously acquired knowledge when new field data is acquired. To address this issue, the incremental model is retrained on all the samples in B every time a new supervised sample becomes available. The considered learning algorithm is the backpropagation algorithm. The size of the buffer significantly impacts learning capabilities, with larger buffers enhancing learning performance but simultaneously increasing memory demands. Since our incremental solution retrains only $\Phi_c(\bullet)$, B is designed to only store the activations ψ_Is produced

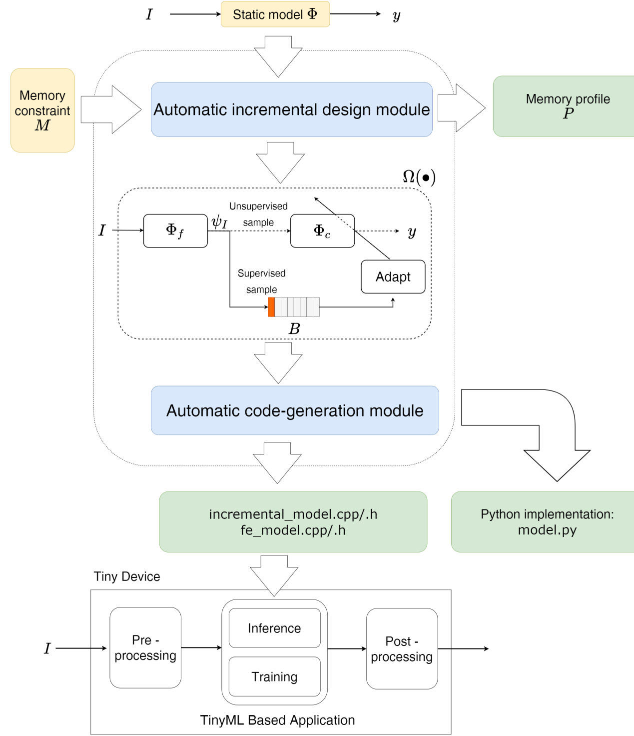


Figure 8: Overview of TyBox. In yellow are highlighted the required inputs, in blue the two modules composing TyBox and in green the produced outputs.

by $\Phi_f(\bullet)$ (together with the corresponding supervised information) and not the original input data I, hence reducing the memory demand of the buffer.

The automatic code-generation module receives in input $\Omega(\bullet)$ and generates the C++ codes and library implementing the inference of $\Phi_c \circ \Phi_f$ and the incremental on-device training of Φ_c. These automatically generated files are intended for direct integration into the firmware of the target tiny device and used by the TinyML application. A comprehensive description of the automatic code-generation process can be found in [38].

Consistency in notation has been maintained throughout this work to reference the same components.

3.2. Constrained NAS

As outlined in Section 2.3.3, MSuNAS [34] is a bi-objective problem which takes as first objective the accuracy while the second objective is the params or the macs (i.e. not a joint but a mutually exclusive optimization of the two indexes). In many scenarios, particularly in the context of TinyML, it would be desirable to optimize both parameters, MACs, and activations while also introducing constraints that can effectively guide the search towards the region of feasible solutions. This leads to the choice of adopting a Constrained Neural Architecture Search (CNAS).

The approach proposed in [22] is a NAS solution that extends MSuNAS by imposing constraints related to the technological and the functional requirements of the device considered for the deployment. The proposed CNAS integrates functional constraints $\mathcal{FC}$ associated with permissable operations within a designed network. To achieve this, layers and activation functions containing specific operations are deliberately excluded from the search space. These constraints are applied to the search space by substituting the layers and activation functions containing not allowed operations with ones including the allowed operations.
Additionally, the technological constraints $\mathcal{TC}$ are imposed in the secondary objective of the optimization that accounts jointly for the number of parameters, MACs and activations of a designed architecture. The inclusion of technological constraints becomes particularly relevant when the neural network must be deployed on a resource-constrained device with limited memory and computational capabilities. It's important to note that in the MSuNAS framework, both $F_P(\tilde{x})$ and $F_M(\tilde{x})$ are exclusively employed as secondary objectives for the optimization process and are not treated as constraints during the search. In addition, $F_A(\tilde{x})$ has been introduced in CNAS, which distinguishes it from the MSuNAS framework. Remarkably, FP (ex) and FA(ex)

account for the total memory demand of ex, while FM(ex) accounts for its computational demand.

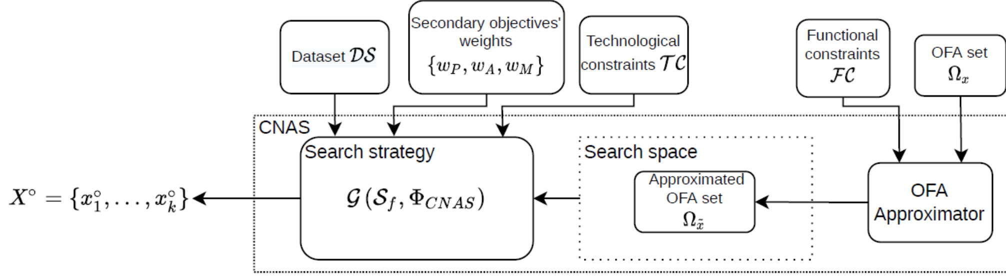


Figure 9: The CNAS framework proposed in [22], composed of an OFA Approximator and a Search Strategy module.

An overall description of the CNAS framework proposed in [22] is given in 9. CNAS receives as input a dataset $\mathcal{DS}$, an OFA set Ω_x, from which the candidate neural networks are obtained, and the two classes of constraints which can be considered in the search procedure, namely the functional constraints $\mathcal{FC}$ and the technological constraints $\mathcal{TC}$. Moreover, a vector of weights $W = [\omega_P,\ \omega_A,\ \omega_M]$, used to balance the different search objectives, can be specified. At the end of the search, CNAS returns the set of the k optimal network architectures $X^o = \{x_1^o, \ldots, x_k^o\}$.
The CNAS constrained optimization problem can be formulated as follows:

$$\begin{aligned} &\text{minimize } \mathcal{G}\left(\mathcal{S}_f(\tilde{x}),\ \Phi_{CNAS}(\tilde{x})\right) \\ &s.t.\ \tilde{x} \in \Omega_{\tilde{x}} \end{aligned} \tag{1}$$

where $\mathcal{G}$ is the bi-objective optimization function, $\tilde{x}$ and $\Omega_{\tilde{x}}$ represent a candidate neural network and the search space of the search, respectively, $\mathcal{S}_f$ is the accuracy of $\tilde{x}$ predicted by the surrogate model. $\Phi_{CNAS}(\tilde{x})$ is the new objective introduced by CNAS and is defined as follows:

$$\begin{aligned} \Phi_{CNAS}(\tilde{x}) = {} & \omega_P \cdot (F_P(\tilde{x}) + \alpha \cdot \max\,(0,\ (F_P(\tilde{x}) - \overline{F}_P))) + \\ & \omega_A \cdot (F_A(\tilde{x}) + \alpha \cdot \max\,(0,\ (F_A(\tilde{x}) - \overline{F}_A))) + \\ & \omega_M \cdot (F_M(\tilde{x}) + \alpha \cdot \max\,(0,\ (F_M(\tilde{x}) - \overline{F}_M))) \end{aligned}$$

where $F_P(\tilde{x}), F_A(\tilde{x}), F_M(\tilde{x})$ account for the total weights, activations and MACs present in $\tilde{x}$, while $\overline{F}_P, \overline{F}_A$ and $\overline{F}_M$ represent the corresponding technological constraints defined in the search procedure. α is a penalty factor, whose role is to assign a high value to $\Phi_{CNAS}(\tilde{x})$ when a technological constraint is violated in $\tilde{x}$.
The functional constraints are managed by means of the OFA Approximator module, which enforces them on the OFA set Ω_x. Let $\Omega_x = \{\Theta_1,\ \ldots,\ \Theta_N\}$ be the set of OFAs which constitute the search space of the NAS, and let τ be a layer or activation function that is forbidden in Ω_x, representing a functional constraint (i.e., τ can not be used in the network candidates). The OFA approximation modifies Ω_x into $\Omega_{\tilde{x}} = \{\tilde{\Theta}_1,\ \ldots,\ \tilde{\Theta}_N\}$, where:

$$\tilde{\Theta}_k = \begin{cases} \Theta_k & \text{if } \Theta_k \text{ does not use } \tau \\ \Theta_k^{\tau,\tilde{\tau}} & \text{otherwise} \end{cases} \tag{2}$$

for k in $\{1,\ \ldots,\ \text{N}\}$ where $\Theta_k^{\tau,\tilde{\tau}}$ refers to Θ_k where the layer τ is replaced with the layer $\tilde{\tau}$ (i.e., $\Theta_k^{\tau,\tilde{\tau}}$ does not use τ). The rules used to replace τ with $\tilde{\tau}$ are specified by the user. The module returns the Approximated OFA set $\Omega_{\tilde{x}}$, which is the search space of CNAS.
The core of CNAS is the Search Strategy module. It uses the genetic algorithm NSGA-II [15] to solve the bi-objective problem introduced in Eq. 1, by jointly optimizing the objectives $\mathcal{S}_f(\tilde{x})$ and $\Phi_{CNAS}(\tilde{x})$ on the dataset $\mathcal{DS}$, evaluating network architectures candidates from $\Omega_{\tilde{x}}$. The search process is iterative: at each iteration the best surrogate accuracy predictor $\mathcal{S}_f$ is obtained through a selection mechanism which constructs four types of surrogate models and adaptively selects the best one via cross-validation. Then, the candidates are ranked according to their predicted accuracy and a new set of candidates - to be evaluated in the next round - is obtained by NSGA-II. The networks produced have the most suitable trade-off between the accuracy and the number of parameters, MACs and activations dictated by the target deployment device.

The CNAS proposed in [22] demonstrates strong compatibility with TinyML, which requires the deployment of ML models on severely-constrained hardware. The objective function presented in Eq. 1 is well suited to be used in this case, since $F_P(\tilde{x}), F_A(\tilde{x})$ and $F_M(\tilde{x})$ are related to the technological aspects of the considered device.

At the same time, the imposition of functional constraints, represented in 2, allows to work in the scenario of Homomorphic Encryption (HE), a particular type of encryption schemes which support the computation of a restricted set of operations directly on encrypted data [24]. It is possible to design deep learning solutions able to operate on encrypted data, hence guaranteeing the privacy of users [20].

4. Proposed Solution

The goal of this work is to design a methodology to integrate NAS with tiny incremental on-device learning able to design incremental models that can be deployed on tiny devices that adhere to the strict constraints of MCUs.
NAS [41], with its capacity to automatically discover optimized neural network architectures, has significantly advanced the development of high-performing models. However, the transition from a discovered architecture to a deployable application often involves manual and error-prone coding tasks, which can be time-consuming and limit the reproducibility of research findings. By integrating NAS with Automatic Code Generation, it is possible to bridge the gap between research and deployment. This integration should focus on creating a unified framework where, after an architecture is discovered by NAS, the corresponding code for that architecture, including data preprocessing, model construction, and training procedures, is generated automatically. This not only accelerates the deployment of novel architectures but also ensures consistency and reliability in the implementation process, making cutting-edge AI models more accessible and practical for a wider range of applications.

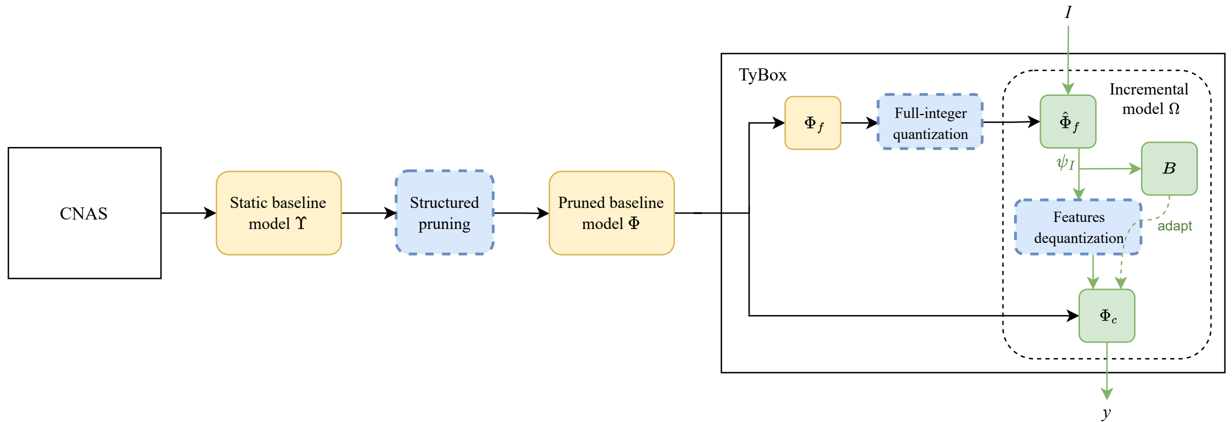


Figure 10: An overview of the proposed methodology. In yellow are highlighted the models and components produced in the intermediate steps, in blue the techniques applied and in green the components of the resulting incremental solution.

The solution proposed in this work integrates the CNAS presented in Section 3.2 with the TyBox toolbox presented in Section 3.1, in order to introduce a methodology for the automated design and code generation of incremental on-device learning models that adhere to the strict constraints of MCUs. Specifically, our work expands the CNAS TinyML image multi-class classification case study, where the CNAS algorithm has been applied on a supernet based on MobileNetV3 [27], a state-of-the-art CNN.
The networks designed by CNAS predominantly target mobile devices, which possess significantly greater computational resources compared to MCUs. To address this issue, various compression techniques have been explored. Among the considered methods, structured pruning has been selected as the most suitable approach. Pruning alone proves insufficient in achieving a network that meets the restrictions of tiny devices. Consequently, the TyBox toolbox has been extended to incorporate full-integer quantization.
The techniques implemented have been selected for their ability to considerably reduce the model size while mitigating the drop in accuracy. A detailed account of their effects is presented in Section 5.

In the following sections, a detailed description of the proposed methodology is presented, alongside the design choices that led to its final configuration. Each stage is analyzed individually, with particular attention directed toward the proposed algorithm. A visual representation of the proposed methodology is given in Fig. 10.

4.1. PyTorch to Tensorflow model conversion

The CNAS algorithm has been developed in PyTorch, while the TyBox toolbox has been designed to receive in input a model in TensorFlow file format. Given the different formats of the two solutions, we opted to convert the CNAS-designed model in TensorFlow. The selection of TensorFlow was motivated by the fact that TensorFlow Lite for Microcontrollers represents one of the most widely used framework for TinyML.

The architecture selected as the baseline $\Upsilon(\bullet)$ for our experiments corresponds to the smallest optimal model discovered in the CNAS TinyML case study [22], where the CNAS algorithm has been applied on a supernet based on MobileNetV3 [27], which has been described in Section 2.3.4. The architecture of the PyTorch baseline model is represented in Fig. 11.

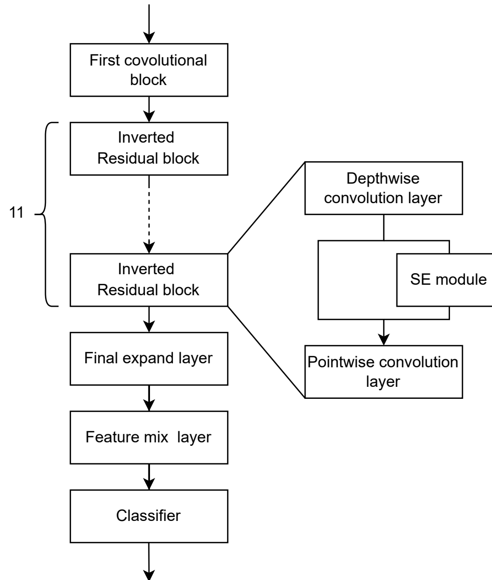


Figure 11: Architecture of the CNAS subnetwork used as baseline in this work.

The First convolution block and the Final expand layer are both composed by a convolutional layer, a batch normalization layer, and a Hswish activation. The Feature mix layer is composed only by a convolutional layer with Hswish as activation function. Finally, the Classifier is composed solely by a dropout layer and a fully connected layer. The Inverted Residual blocks are composed by a depthwise separable convolution, which has been described in Section 2.3.4, comprising the depthwise and pointwise convolution layers and their respective batch normalization and activation layers. Additionally, some Inverted Residual blocks have the SE module (described in Section 2.3.4) [29], located between the depthwise and pointwise convolution layers.
The conversion of the PyTorch model has been done with a manual approach. Most of the PyTorch layers the compose the baseline model have a TF counterpart. Only PyTorch *nn.Hardswish()* [2] has not a proper TF counterpart. To address this issue, the regular *tf.keras.activations.swish* [3] has been used.

4.2. Structured pruning

Once the PyTorch baseline model has been converted, producing the TensorFlow version of $\Upsilon(\bullet)$, it undergoes the process of structured pruning.
In contrast to conventional pruning techniques [12], which simply convert the weights with smaller magnitudes to zeros, structured pruning involves the effective compression of model occupation by removing its non-essential components. In our scenario, structured pruning has been applied to the Convolutional layers to eliminate their least relevant filters, as proposed in [32]. The filter's relevance is estimated by computing the L1-norm of the filter's weights. To achieve the optimal performance on the reduced model, multiple filter configurations have been tested for each convolutional layer.
A reasoned order was applied, targeting the convolutional layers with the highest number of filters down to those with fewer filters. In our specific context, it was observed that layers containing a greater number of parameters - and consequently having a higher computational load - often exhibited a significant discrepancy in the L1-norm values of their filters. The proposed approach prioritizes the pruning of filters in the latter convolutional layers of $\Upsilon(\bullet)$, where a higher proportion of less influential filters is found compared to $\Upsilon(\bullet)$'s earlier convolutional

layers. This strategy effectively removes a larger quantity of less critical filters, contributing to streamlined model architectures without compromising performance.
A more in-depth analysis of the application of the described approach is provided in Section 5.3.1

4.3. Full-integer quantization

Once the static baseline model $\Upsilon(\bullet)$ has been pruned, resulting in $\Phi(\bullet)$, it undergoes a decoupling process within TyBox. To further reduce the memory and computational load of the model, TyBox has been extended to incorporate full-integer quantization.
Full-integer quantization has been identified as the optimal method in this scenario due to its capability to convert model input, model output, weights, and activation outputs into 8-bit integer data, compared to other quantization techniques which may leave some amount of data in floating-point [23]. This allows to achieve up to a 4x reduction in memory usage and up to a 3x improvement in latency.

The pruned model $\Phi(\bullet)$ is partitioned in a feature extractor $\Phi_f(\bullet)$ and a classifier $\Phi_c(\bullet)$. $\Phi_f(\bullet)$ is composed by all the convolutional blocks of $\Phi(\bullet)$, while $\Phi_c(\bullet)$ consists solely of the final Dense layer of the model. The primary advantage of TyBox lies in its incrementally learnable classifier, which constitutes a negligible portion of the overall model size. In order to reduce the model dimension while maintaining its prediction abilities, quantization is exclusively applied to $\Phi_f(\bullet)$, producing $\hat{\Phi}_f(\bullet)$. This approach allows the quantized feature extractor to have an 8-bit resolution, while the incremental classifier retains a 32-bit resolution, ensuring a more precise classification process is maintained. Since $\hat{\Phi}_f(\bullet)$ produces 8-bit outputs, while $\Phi_c(\bullet)$ is designed to receive 32-bit inputs, $\psi_I(\bullet)$ undergoes a dequantization process before being passed to $\Phi_c(\bullet)$. The same process is also employed during the incremental training phase, where the quantized data samples stored in the buffer undergo dequantization before before being utilized for training the classifier.
We provide here fragments of python code, which represent the modifications made to implement full-integer quantization within TyBox.
The function `convert_to_tflite()` converts $\Phi_f(\bullet)$ into a .tflite model by using the `convert()` function of the TFLiteConverter. In order to implement the 8-bit conversion of the model, some additional parameters have been defined:

```
def convert_to_tflite(model, yield_representative_dataset=None):
    if yield_representative_dataset:
        [...]
        converter_Mf_lite.inference_input_type = tf.int8
        converter_Mf_lite.inference_output_type = tf.int8
        [...]
```

In presence of a representative dataset, `convert_to_tflite()` performs a full-integer quantization before converting the model. The resolution selected for the 8-bit quantization is tf.int8, since it is the only one supported by TensorFlow Lite for Microcontrollers.
The function `evaluate()` is used to evaluate the accuracy of the classifier and to incrementally train it each time new data is provided.

```
# inputs: inputs of the incremental classifier , these are the features
#          extracted by the feature extractor
# targets: target labels of the inputs, used to evaluate the model
#          classification accuracy
def evaluate(self, inputs, targets, dequantization_params=None):
    [...]
    if dequantization_params:
        input_scale, input_zero_point = dequantization_params
    for input_index in range(len(inputs)):
        #  During quantization:
        #  input_data = input_data / input_scale + input_zero_point
        #  input_data = input_data.astype(input_details["dtype"])
        fe_output_data = inputs[input_index]
        if fe_output_data.dtype == np.uint8:
            fe_output_data = fe_output_data.astype('float32')
            fe_output_data = (fe_output_data - input_zero_point) * input_scale

        self.execute_forward_pass(fe_output_data)
        target_label = list(targets[input_index]).index(max(targets[input_index]))
```

```
        calculated_label = list(self.layers[-1]).index(max(self.layers[-1]))
        if target_label == calculated_label:
            score += 1
    return score/len(inputs)
```

The function receives in input the features extracted by $\hat{\Phi}_f(\bullet)$, that have a resolution of 8-bit. Before the incremental learning and the evaluation phases, the data undergoes a dequantization process in order to match the 32-bit input resolution of $\Phi_c(\bullet)$. Dequantization is achieved by applying the inverse operations of those carried out during the quantization process. The specific dequantization parameters are retrieved from the output details of the TFLiteInterpreter.
To correctly calculate the memory requirements of $\hat{\Phi}_f(\bullet)$ and determine the appropriate size for the buffer B, the TyBox Profiler has been configured to receive explicit resolution parameters for the weights and activations of both the feature extractor and classifier.

```
class Profiler:
    def __init__(self, network_name, model, mem_optimization=1, precisions):
        [...]
```

By applying full-integer quantization only on the feature extractor, the memory occupation of the complete model does not achieve an exact 4x reduction, since the classifier, although only constituting a negligible portion of the model size, retains a resolution of 32-bit.
As expressed in [38], the corresponding memory demands (in Bytes) to store the parameters m_k^θ and the activations m_k^a of a layer can be easily computed as follows:

$$m_k^\theta = |\theta_k| \cdot M_w$$
$$m_k^a = |a_k| \cdot M_w$$

where M_w is the dimension in Bytes required to store a single value (either weight or activation). Without quantization, M_w would assume a value of 4 since weights are stored as 32-bit floating-point values. In our scenario, since weights and activations of the $\hat{\Phi}_f(\bullet)$ are stored as 8-bit integer values, the memory demand is:

$$\hat{m}_k^\theta = |\theta_k| \cdot \hat{M}_w$$
$$\hat{m}_k^a = |a_k| \cdot \hat{M}_w \tag{3}$$

where $\hat{M}_w$ assumes a value of 1.
The total amount of memory M_{Φ_f} required for the weights and activations of $\Phi_f(\bullet)$ and the one M_{Φ_c} of $\Phi_c(\bullet)$ can be computed as follow:

$$M_{\Phi_f} = \sum_{k=0}^{k<=l} m_k^\theta + \max_{k=0}^{k<l}(m_k^a + m_k^{a+1})$$
$$M_{\Phi_c} = \sum_{k=l+1}^{k<=K} (m_k^\theta + m_k^a)$$

being K the total number of processing layers in the model and l the index of the last layer of $\Phi_f(\bullet)$. Conversely, the total amount of memory $\hat{M}_{\Phi_f}$ required to store the weights and activations of $\hat{\Phi}_f(\bullet)$ is:

$$\hat{M}_{\Phi_f} = \sum_{k=0}^{k<=l} \hat{m}_k^\theta + \max_{k=0}^{k<l}(\hat{m}_k^a + \hat{m}_k^{a+1}) \tag{4}$$

while the memory required by the classifier remains the same. Hence, through the application of full-integer quantization solely on the feature extractor, the reduction factor from M_{Φ_f} to $\hat{M}\Phi f$ is exactly 4x, but this reduction does not apply to the entire models, as the memory requirement of M_{Φ_c} must also be taken into account.

In addition to reducing the memory occupation of the model, the applied compression techniques contribute to a decrease in the memory demand of the TyBox buffer B. Each sample stored in B correspond to a vector of extracted features Ψ_I produced by $\hat{\Phi}_f(\bullet)$. Pruning the filters of the last convolutional layer of $\Phi_f(\bullet)$ reduces the total number of features extracted, thus reducing the length of Ψ_I. Moreover, by applying full-integer quantization, the vectors of features extracted by $\hat{\Phi}_f(\bullet)$ have a resolution of 8-bit. By reducing the dimensionality of each latent representation, the memory required to store a sample is consequently diminished.

The dimension of the buffer M_B (in Bytes) and the number of samples N_B that can be stored on the device are automatically computed on the basis of the total available memory M and the memory demand for weights and activations computed in Equations 3 and 4. Without quantization, M_B and N_B are calculated as follows:

$$M_B = \left\lfloor \frac{M}{M_w} \right\rfloor \cdot M_w - M_{\Phi_f} - M_{\Phi_c}$$
$$N_B = \frac{M_B}{M_w \cdot |\Psi_I|}$$

On the contrary, in our scenario, the amount of memory $\hat{M}_B$ allocated to the buffer B and the number of samples $\hat{N}_B$ that can be stored on the device are:

$$\begin{aligned} \hat{M}_B &= \left\lfloor \frac{M}{\hat{M}_w} \right\rfloor \cdot \hat{M}_w - \hat{M}_{\Phi_f} - \hat{M}_{\Phi_c} \\ \hat{N}_B &= \frac{\hat{M}_B}{\hat{M}_w \cdot |\Psi_I|} \end{aligned} \tag{5}$$

Comparing the previous equations, it is possible to note that, with the same available memory M, the memory $\hat{M}_B$ accessible by the buffer is greater than M_B, since the quantized feature extractor occupies less memory. Moreover, since each vector of extracted features Ψ_I produced by $\hat{\Phi}_f(\bullet)$ has a reduced occupation, the number of samples that can be stored in B is higher. On the other hand, given the reduced dimensionality of Ψ_I, less memory is needed to achieve a buffer with the same length in the standard TyBox scenario with no quantization. Upon comparing the preceding equations, it becomes evident that, with the same available memory M, the memory $\hat{M}_B$ accessible to the buffer surpasses that of M_B, due to the reduced memory footprint $\hat{M}_{\Phi_f}$ of the quantized feature extractor. Furthermore, given the reduced size of Ψ_I, less memory is required to maintain a buffer of equivalent length compared to the standard TyBox scenario without quantization.

5. Experimental results

5.1. Dataset

Our work expands the CNAS TinyML image multi-class classification case study, where the CNAS algorithm has been applied on a supernet based on MobileNetV3 [27], using the CIFAR-10 dataset for the evaluation of the model performance. For our experiments, other than CIFAR-10 [5], also the Imagenette dataset [21] has been considered, since it is a dataset for image classification with the same number of classes of CIFAR-10.

5.1.1 CIFAR-10

The CIFAR-10 dataset (Canadian Institute For Advanced Research) stands as a cornerstone in computer vision research, serving as a gold standard for evaluating and benchmarking a wide array of Machine Learning and Deep Learning models [5]. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images, with 6000 images per class, labelled with one of 10 mutually exclusive classes: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck). There are 50000 training images and 10000 test images.
CIFAR-10 offers several key characteristics that make it a valuable resource for computer vision experiments:

- Real-world relevance: The dataset's images are real-world photographs, which means they contain variations in lighting, background, pose, and object orientation, reflecting the challenges encountered in real-world computer vision applications.
- Balanced classes: CIFAR-10 exhibits a balanced class distribution, with each class having an equal number of examples. This balance ensures that models are not biased towards any specific category.
- Small image size: The images in CIFAR-10 are relatively small, measuring only 32x32 pixels. This constraint compels models to learn abstract representations from limited spatial information, making it an excellent testbed for evaluating model capacity and generalization.
- Diverse Object Categories: The dataset spans a diverse range of object categories, encompassing both animate and inanimate objects, which facilitates the development of models capable of recognizing various types of objects.

CIFAR-10 has been widely employed as a benchmark dataset for numerous computer vision tasks, including image classification, object recognition, feature learning, and transfer learning [4]. Researchers often use it to

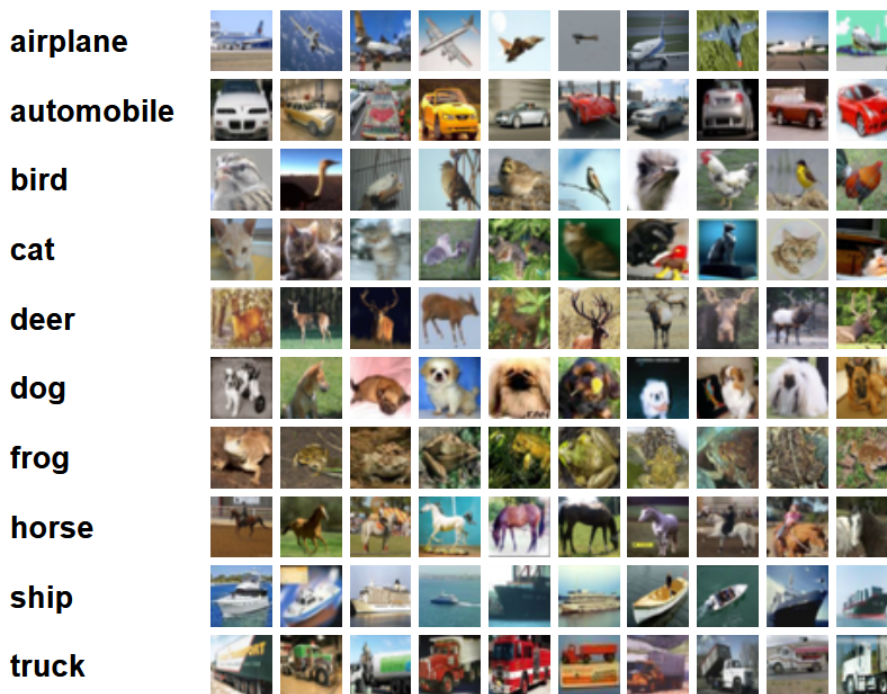


Figure 12: A sample of the 32x32 color images present in the CIFAR-10 dataset. [5]

assess the performance of different ML and DL architectures, including CNNs. Moreover, CIFAR-10 serves as a foundation for exploring techniques such as data augmentation, regularization, and model compression.
The CIFAR-10 dataset has emerged as an indispensable resource for computer vision research, fostering innovation and advancement in the field. Its real-world images, balanced class distribution, and diverse object categories make it an ideal choice for evaluating the efficacy and robustness of various computer vision algorithms and models.

5.1.2 Imagenette

Imagenette is a subset of 10 easily classified classes from the Imagenet dataset [21]. Imagenette is derived from the Imagenet dataset, which is renowned for its extensive collection of images across thousands of object categories, but offers a more accessible and compact alternative for researchers and practitioners to experiment with DL models. Imagenette simplifies this complexity by selecting ten representative object classes, significantly reducing the dataset's size and complexity. These classes encompass a diverse range of objects, including animals, everyday items, and various objects found in natural and urban environments: tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute.
One of the defining characteristics of Imagenette is its image resolution. Other than full-size, the dataset is available in two alternative variants, which maintain the same classes and structure as the full Imagenette dataset, but have images resized to a smaller dimension (320 px or 160 px). This reduction in resolution poses a unique challenge to DL models, as it requires them to recognize objects and patterns from limited spatial information. Consequently, Imagenette is well-suited for assessing model generalization and the effectiveness of transfer learning techniques.

Imagenette has found extensive utility in computer vision research, particularly in the context of transfer learning and fine-grained image classification. Researchers frequently employ Imagenette as a testbed for transfer learning experiments. Pretrained models trained on larger datasets like Imagenet can be fine-tuned on Imagenette, allowing for the evaluation of how well these models generalize to new Moreover, Imagenette's diverse but simplified set of object categories lends itself to fine-grained classification tasks. Researchers use the dataset to develop and assess models capable of distinguishing between visually similar object classes, a challenging task often encountered in real-world applications.
Imagenette plays a crucial role in advancing the field of computer vision by providing a versatile and accessible dataset for researchers and practitioners. Its simplified yet diverse object classes, reduced image resolution, and focus on transfer learning and fine-grained classification make it a valuable resource for exploring the capabilities and limitations of deep learning models in a controlled and manageable setting.

5.2. Baseline model conversion

The CNAS architecture used as the baseline $\Upsilon(\bullet)$ for the following experiments has a parameters occupation of 8.72MB, and is characterized by the following configuration:

- *params*: 2.14M, *macs*: 6.85M, *activations*: 0.23M;
- *top1 accuracy*: 89.9%

As expressed in Section 4.1, given the differences in implementation between CNAS [22] and TyBox [38], we opted to convert the CNAS-designed PyTorch model to TensorFlow.
The model architecture has been delineated in Section 4.1. The presence of SE module, which has been described in Section 2.3.4, introduces non-linearities in the baseline model $\Upsilon(\bullet)$. The non-linearity of $\Upsilon(\bullet)$ renders it incompatible with existing automatic PyTorch to TensorFlow convertion tools, such as ONNX converters [6], which are designed to work with linear models.
In response to the non-linearity of the baseline model, our initial approach has been to manually perform the conversion of the CNAS architecture and its corresponding weights to TensorFlow. However, we have not been capable to create a TF model able to emulate its PyTorch counterpart, as the converted TF model failed to produce correct inferences when provided with data.

Given the challenges introduced by the non-linear CNAS model, we opted to manually convert the CNAS architecture and train it from scratch in TensorFlow. To address the conversion process, a different input image size has been adopted. The CNAS supernet was initially trained on Imagenet [16], which contains 256x256 color images, while the subnetworks were fine-tuned on CIFAR-10 [5], comprising 32x32 color images that have been resized to match the resolution explored by the NAS. The PyTorch network chosen as baseline for our experiment has an input size of 40x40. Due to resource limitations, conducting the supernet training on the Imagenet dataset in TF was not feasible. Consequently, only the training on CIFAR-10 has been performed, using an input image size of 32x32.
The network has been trained for 150 epochs using a custom configuration, with Adam optimizer, a learning rate of 0.0015, and early stopping (patience of 20 epochs), resulting in an accuracy of 79.29%. The poorer performance of the TF training has been attributed to the differences in training, the diverse data preprocessing between the two ML libraries and the different input image sizes required by the two networks.

5.3. Proposed methodology

This section details the results achieved by the application of the proposed methodology described in Section 4. The target technological constraint M we imposed on the on-device RAM memory, and that must be satisfied by the final incremental TinyML solution, is 1 MB.

5.3.1 Structured pruning

One of the compression techniques used for model compression is pruning. Initially, our approach involved the utilization of the standard weight pruning technique provided by TensorFlow. This technique makes use of the function *tfmot.sparsity.keras.prune_low_magnitude()* [7] to identify and prune model weights characterized by low magnitudes (i.e., weights that are close to zero) while preserving the more significant ones. Through the application of weight pruning, we were able to produce a compressed model that exhibited a 50% reduction in size while maintaining an accuracy level comparable to that of the original baseline model. However, weight pruning does not inherently reduce the model size. Instead, it identifies the least significant weights within the model and forces them to zero. Consequently, when saving the pruned model, the weights continue to occupy the same space, regardless of whether their values are zero or not. As a result, the memory footprint of the pruned model file does not align with the size indicated during the pruning procedure.

In order to actually reduce the model size, rather than using standard pruning techniques, we opted to employ structured filter pruning, which has been presented in Section 4.2. This approach involves the removal of the least significant filters from convolutional layers (the filter's relevance is estimated by computing the L1-norm of the filter's weights [32]). Structured filter pruning not only allows for an actual reduction in model occupation but also grants greater flexibility during the pruning process. Unlike the pruning method proposed by TensorFlow, which uniformly enforces a reduction across the model or specific layers, structured filter pruning enables the manual selection of filters to be removed from each layer. This level of customization permits the creation of compressed models that maintain performance levels comparable to the baseline model.
As anticipated in Section 4.2, our prioritization strategy for pruning has focused on convolutional layers characterized by the highest filter count. In Table 1, all the convolutional layers of the baseline model $\Upsilon(\bullet)$ are

presented.

Layer	#filters
conv2d	16
conv2d_1	16
conv2d_2	16
conv2d_3	48
conv2d_4	48
conv2d_5	24
conv2d_6	72
conv2d_7	72
conv2d_8	24
conv2d_9	72
conv2d_10	72
conv2d_11	24
conv2d_12	72
conv2d_13	40
conv2d_14	120
conv2d_15	120
conv2d_16	32
conv2d_17	120
conv2d_18	40
conv2d_19	120
conv2d_20	120
conv2d_21	80
conv2d_22	240
conv2d_23	240
conv2d_24	80
conv2d_25	240
conv2d_26	240
conv2d_27	64
conv2d_28	240
conv2d_29	112
conv2d_30	336
conv2d_31	336
conv2d_32	88
conv2d_33	336
conv2d_34	112
conv2d_35	336
conv2d_36	336
conv2d_37	88
conv2d_38	336
conv2d_39	160
conv2d_40	480
conv2d_41	480
conv2d_42	120
conv2d_43	480
conv2d_44	160
conv2d_45	960
conv2d_46	1280

Table 1: Convolution layers of the baseline model $\Upsilon(\bullet)$ and their respective number of filters.

It is worth noting that, in our particular context, a substantial concentration of filters is observed within the final layers of the model. For this reason, the pruning process has been applied starting from these latter layers, which contribute more to the computational load of the model, having a higher number of filters. Moreover, the convolutional filters within the final layers of the model exhibit a more pronounced disparity in their respective L1-norm values. This suggests that filters characterized by lower L1-norm values in these layers carry considerably reduced significance when compared with filters displaying higher L1-norm values.
Fig. 13 illustrates the graphical representations of filter values for six distinct convolutional layers within the model $\Upsilon(\bullet)$. The upper set of graphs depicts three middle layers, while the lower set represents some of the final layers in the model. These visualizations offer insights into the distribution of L1-norms among the filters within various segments of the model.
In the majority of final layers, only a small portion of filters is characterized by high L1-norm values, while the others have very low L1-norms. This behaviour is manifested by the presence of an inflection point in the graphs. Conversely, in the middle (and initial) layers, it is difficult to find filters that significantly outweigh others in terms of L1-norm value. In this initial layers, the differences in L1-norm value among filters are notably more modest in comparison to the pronounced variations observed in the final layers.

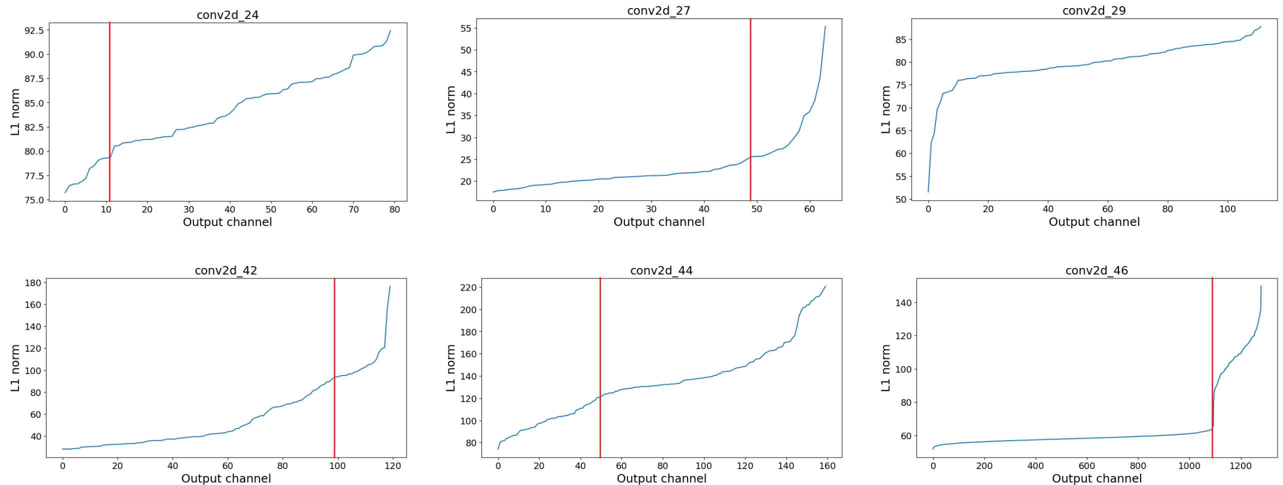


Figure 13: Distribution of L1-norm values in convolutional filters across some layers of $\Upsilon(\bullet)$. The upper graphs illustrate convolutional layers positioned in the middle section of the model, while the lower graphs represent a selection of the final layers. It is of particular significance to observe that in the upper graphs, the filters L1-norms have closer values compared to the ones in the bottom graphs. The red line represent the division between pruned and non-pruned filters (the line is only present if pruning has been applied).

Model	Size (%)	Accuracy
Υ	8.72MB (100%)	79.29%
$\Phi_{4.31MB}$	4.31MB (49.4%)	79.48%
$\Phi_{2.89MB}$	2.89MB (33.1%)	77.15%
$\Phi_{2.56MB}$	2.56MB (29.4%)	65.18%
$\Phi_{2.13MB}$	2.13MB (24.4%)	51.66%
$\Phi_{1.74MB}$	1.74MB (19.9%)	13.26%

Table 2: Structured filter pruning results.

The primary layers considered for pruning are the layers characterized by the presence an inflection point in the filters graph. The filters subject to pruning correspond to those positioned prior to the inflection point. A visual representation of this concept is provided in Fig. 13, where the red line delineates the demarcation between pruned (left) and non-pruned (right) filters.

Various filter pruning configuration have been tested to evaluate the performance of the compressed models. The results are presented in Table 2.
The most notable observation concerns the accuracy of the compressed networks. By removing the least significant convolutional filters of $\Upsilon(\bullet)$, a compression of up to 49.4% of the original size can be achieved without exhibiting any significant accuracy drop. Upon surpassing this threshold, ulterior pruning causes a gradual decline in accuracy, ultimately leading to poor results when too many filters are removed.

5.3.2 TyBox full-integer quantization

To accompany structured pruning in the task of model compression, full-integer quantization has been employed by extending the TyBox toolbox, as described in 4.3. The designated approach for our problem is post-training quantization. We emphasize that, to reduce the model dimension while maintaining its prediction abilities, quantization has been exclusively applied to $\Phi_f(\bullet)$

Before FE quantization	After FE quantization
$\Phi_{4.31MB}$ 4.31MB (49.4%) 79.48%	$\Omega_{1.19MB}$ 1.19MB (13.7%) 78.5%
$\Phi_{2.89MB}$ 2.89MB (33.1%) 77.15%	Ω_{801kB} 801kB (9.2%) 75.0%

Table 3: Effects of TyBox Feature Extractor full-integer quantization.

In accordance with the established memory constraint *M*, the primary model selected for deployment is $\Phi_{2.89MB}(\bullet)$. An evaluation of the performance of $\Phi_{4.31MB}(\bullet)$ has also been conducted to assess the impact of a more relaxed pruning. The effects of the TyBox Feature Extractor full-integer quantization are shown in Table 3.

The implementation of full-integer quantization in TyBox induces a substantial reduction in model size, accompanied by a concurrent decrease in accuracy, that is influenced by the strictness of the previously performed pruning. Considering $\Phi_{2.89MB}(\bullet)$, although experiencing a 2.15% drop in accuracy, the resulting TyBox incremental model $\Omega_{801kB}(\bullet)$ achieves an occupation of 801kB, making it suitable for the deployment on a device with memory constraint *M*.
The size reduction does not correspond to exactly 75% because, as anticipated in Section 4.3, only the feature extractor, which constitutes the 98.8% of $\Omega_{801kB}(\bullet)$ and the 99.4% of $\Omega_{1.19MB}(\bullet)$, has been quantized.

5.4. Application scenarios

This section details all the experiments carried out to validate the effectiveness and efficiency of the methodology proposed in Section 4.

The experimental setting concerns the image classification on a multi-class problem. For this purpose, the CIFAR-10 [5] and Imagenette [21], described in Section 5.1 datasets have been considered. To validate the performance of the incremental models, we considered the same application scenarios that have been used for the validation of the TyBox multi-class case study [38]:

- Concept drift, which refers to the variability in the data distribution over time that can occur after deploying a ML model on-device [19]. By subjecting the model to concept drift, we emulate the dynamic nature of data encountered in real-world applications like sensor networks, financial markets, or environmental monitoring. This experiment measures the ability of the designed incremental model to recognize and accommodate changes in the process generating the data. A model proficient in handling concept drift can maintain its predictive accuracy and relevance, even as the underlying data undergoes transformations. It ensures that the model remains a reliable tool for making decisions or predictions in applications where data dynamics are prevalent.
- Incremental learning, where the tasks to be solved is extended after the model has been deployed on-device [35]. This experiment is pivotal for assessing the model's ability to learn and adapt to additional tasks without forgetting the previously learned ones. Such versatility is crucial in various applications, including adaptive robotics, evolving sensor networks, and dynamic content filtering. A model adept at incremental learning can continually acquire new knowledge while retaining previously learned information. This capability is vital in scenarios where the model needs to evolve with evolving tasks or datasets.
- Transfer learning, which entails training a model for a primary task and then reusing or adapting it for a secondary task. In our experiment, we initially train a static CNAS model for a specific task. Then, after being compressed and deployed, it is incrementally re-trained on-device to address a different classification task. The utility of this experiment lies in evaluating the model's capacity to transfer and leverage previously acquired knowledge effectively. Transfer learning enhances model efficiency by avoiding the need for training from scratch for every new task, which can be resource-intensive and time-consuming. It is particularly valuable when addressing a series of related tasks or domains where common features or knowledge can be reused.

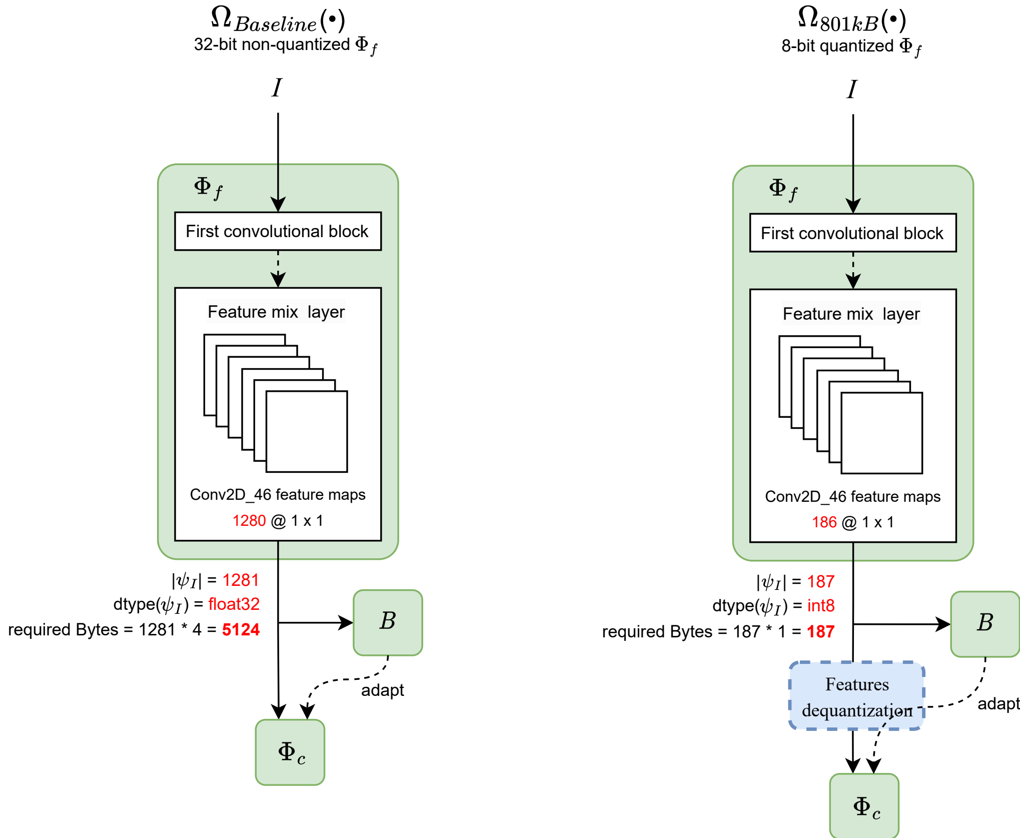


Figure 14: Differences in occupation between vectors of features extracted by $\Omega_{Baseline}$, the non-quantized incremental version of the baseline model Υ, and Ω_{801kB}, the compressed incremental model produced by our methodology. Pruning the filters of the last convolutional layer of Ω_f (right) allows to reduce the number of features extracted. Moreover, by implementing full-integer quantization in TyBox, these features can be represented by 1 Byte, rather than 4.

Two distinct solutions are considered for the comparison:

- The non-quantized TyBox incremental version of the original baseline model, $\Omega_{Baseline}(\bullet)$: this model is the result of a straightforward integration of CNAS and TyBox, where the standard unaltered version of TyBox is applied to the baseline model $\Upsilon(\bullet)$, without any pruning or quantization. By comparing the performance of models generated by the proposed methodology and the conventional CNAS and TyBox implementation, we can evaluate the enhancements and optimizations that our approach introduces, other that its effectiveness.
- TinyOL [40], an alternative incremental on-device toolbox, which perform the incremental training only on the latest supervised sample received rather than all the samples stored in B, equivalent to a quantized TyBox incremental model with buffer size equal to 1. The comparison with TinyOL is crucial to assess the effectiveness of the incremental solution designed by TyBox, in order to evaluate the advantages brought by the inclusion of the buffer B in the NAS for incremental on-device learning scenario.

The performance of $\Omega_{1.19MB}(\bullet)$ are also provided, to analyze the impact of a less strict pruning. To ensure comparable results, many different learning rates have been tested for each model. Experimental results are listed in Figures 15, 16, and 17.
Each experiment has been represented with five graphs. The initial graph within each set represent the mean over 5 repetitions for all the tested models and, for the sake of clarity, the confidence intervals have been omitted. Each of the following four graphs of a set report the results of a single model, showing both mean and confidence intervals.

The incremental model used in this experimental setting is $\Omega_{801kB}(\bullet)$, since it is the one that respects the target constraints imposed by M (1 MB). Considering that during the deployment some of the available 1MB memory is dedicated to the storage of libraries and code, the memory constraint imposed in the TyBox experiments is 830kB. This allows to have a buffer B of around 29kB. By applying structured pruning on the last convolutional layer of $\Omega_f(\bullet)$ and by implementing full-integer quantization in TyBox, each individual buffer sample requires only 187 bytes for storage (against the 5124 bytes required for the storage of the feature vectors extracted by $\Omega_{Baseline}(\bullet)$, see Fig. 14), so $B_{801kB}(\bullet)$ achieves a capacity of 158 samples. To provide a more meaningful comparison with the other evaluated models, the same buffer capacity has been maintained for each experiment. In each experiment, the training set is provided in an incremental manner during the experiments, while 200 elements of the test set are used to evaluate the classification accuracy after each supervised sample is provided.

5.4.1 Concept drift

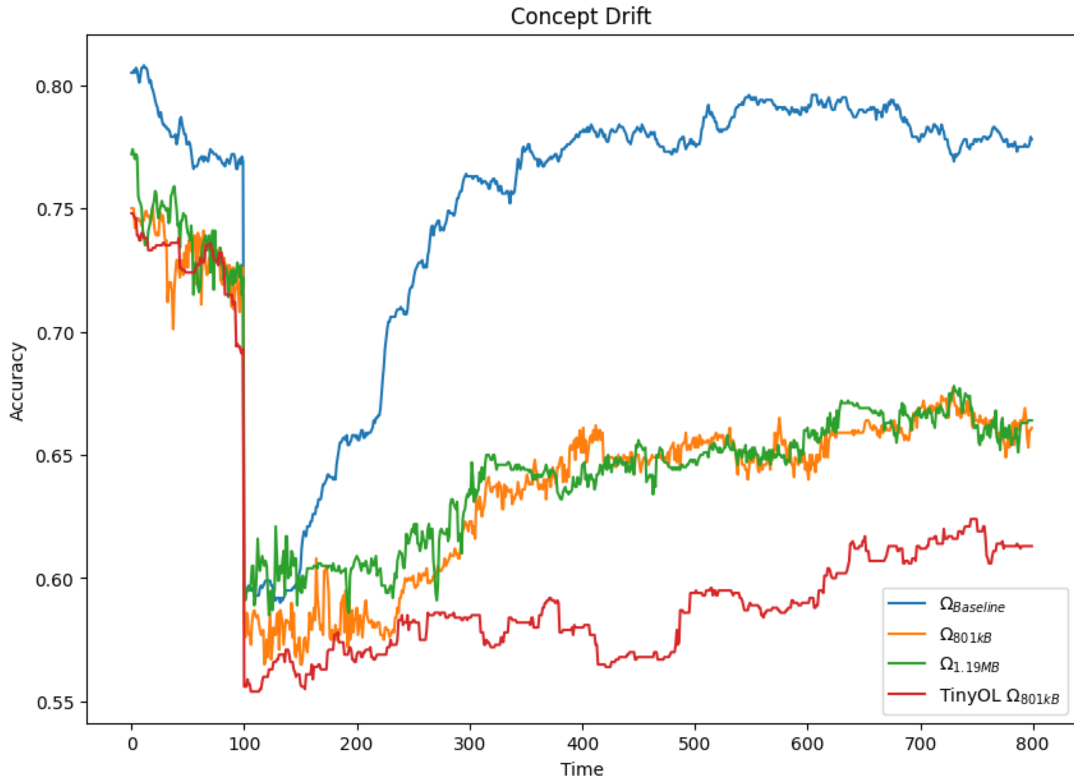
Following the formalization of concept drift previously introduced, we defined an application scenario where the process generating the data evolve over time. In particular, we considered an abrupt concept drift affecting the CIFAR-10 [5] multi-class classification problem where classes 4 and 6 are swapped at sample 100. The total number of training samples is 800. The performance for the different configurations of the TyBox and TinyOL incremental solutions and for the incremental baseline model are illustrated in Fig. 15.

For all the models, an initial adjustment in accuracy can be observed. This behaviour is most certainly to be attributed to incremental learning. After the drop in accuracy caused by the concept drift, all the models are characterized by an initially slow recovery phase. This behaviour is a consequence of the fact that, after the concept drift, the buffer needs to adapt to the new data distribution by removing obsolete samples. Although both $\Omega_{801kB}(\bullet)$ (0.658 ± 0.019) and $\Omega_{1.19MB}(\bullet)$ (0.663 ± 0.021) are able to gradually recover from the concept drift, they do not manage to reach their initial level of accuracy. This is reasonable considering the substantial compression the models have been subjected to. Notably, the TyBox $\Omega_{801kB}(\bullet)$ incremental solutions display a superior recovery compared to its TinyOL counterpart (0.613 ± 0.014). This is certainly attributed to the increased buffer size. As expected, $\Omega_{Baseline}(\bullet)$ is able to completely recover from the effects of concept drift (0.778 ± 0.007).

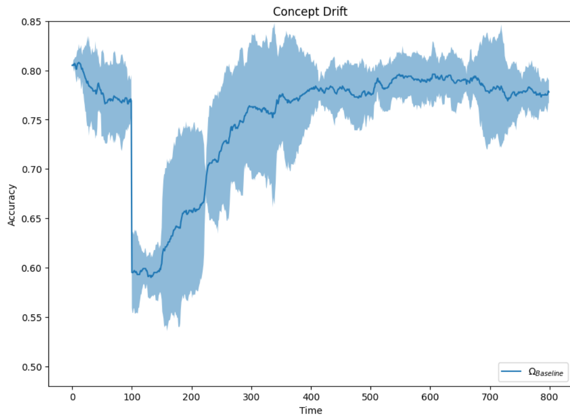
5.4.2 Incremental learning

This application scenario models the setting where the classification task is incrementally extended to model a broader classification problem. More specifically, the classification problem on the CIFAR-10 dataset is initially configured comprising only classes from 0 to 7, while at sample 100 even classes 8 and 9 are included in the classification problem. 700 samples are used as incremental training set, while the test set comprises 200 samples containing all the 10 classes. The classification accuracy for all the evaluated models are depicted in Fig. 16.

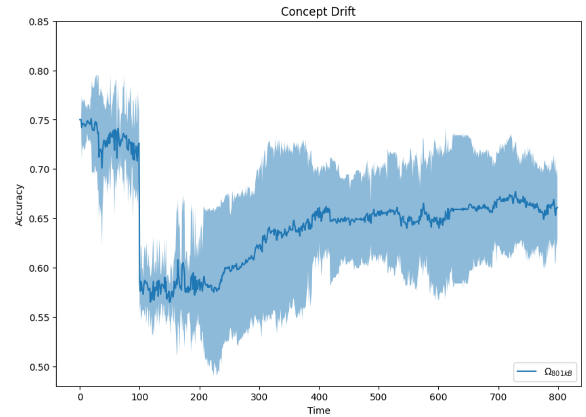
Given the design of this experimental problem, the models used in this experiment are different from those of the other two scenarios. The baseline CNAS architecture has been only trained over the classes 0-7, resulting in the model $\Upsilon^{IL}(\bullet)$, which achieves an accuracy of 61.42%. The inferior performance, in comparison to



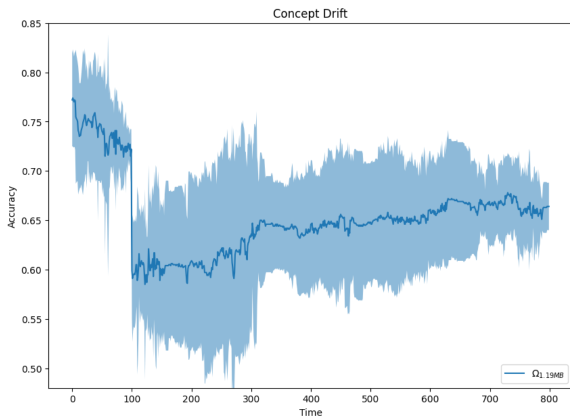
(a) Concept Drift scenario: means comparison.



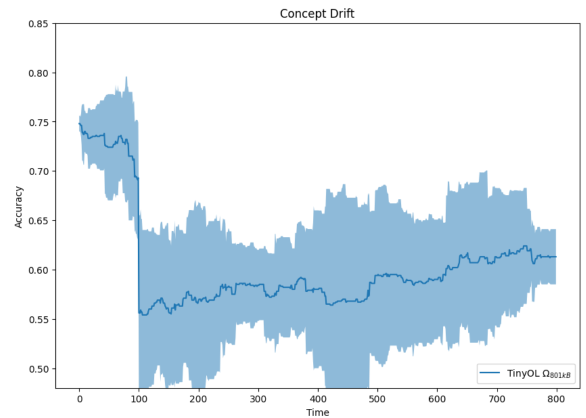
(b) $\Omega_{Baseline}(\bullet)$.



(c) $\Omega_{801kB}(\bullet)$.

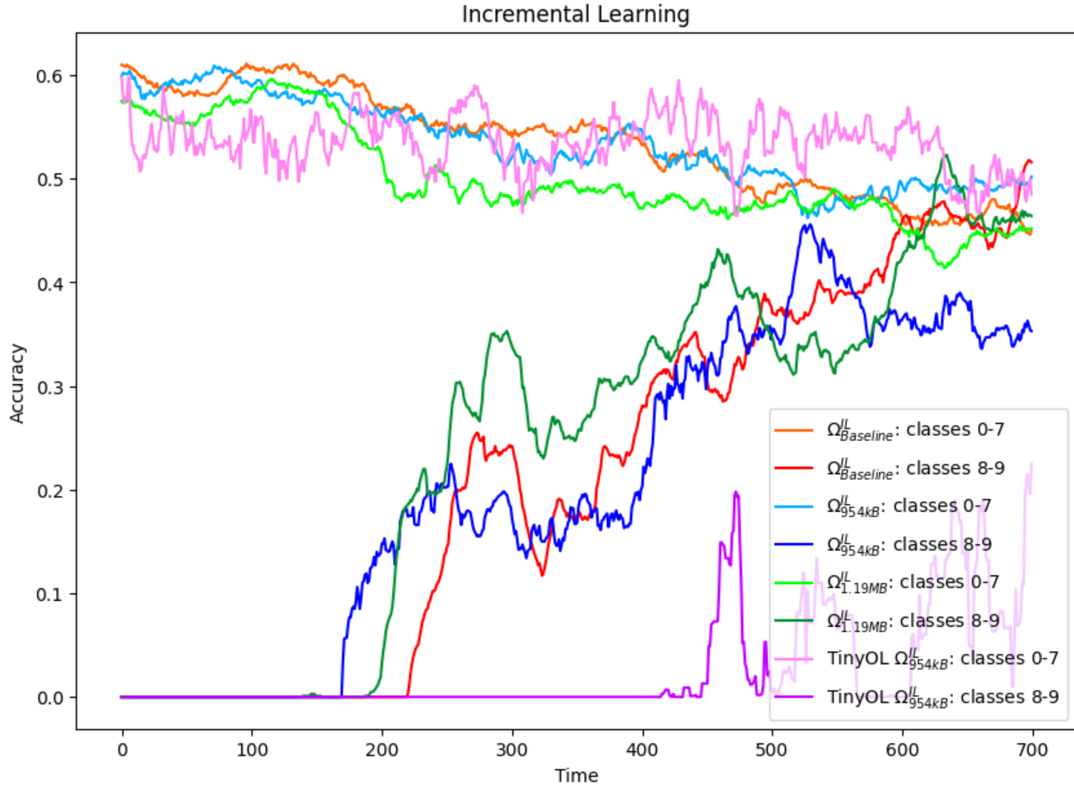


(d) $\Omega_{1.19MB}(\bullet)$.

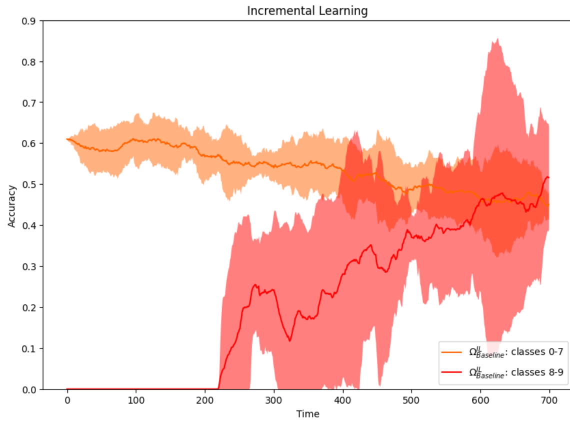


(e) TinyOL $\Omega_{801kB}(\bullet)$.

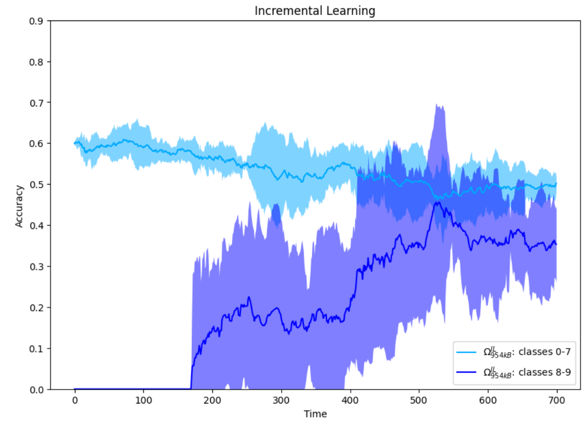
Figure 15: Concept Drift scenario results.



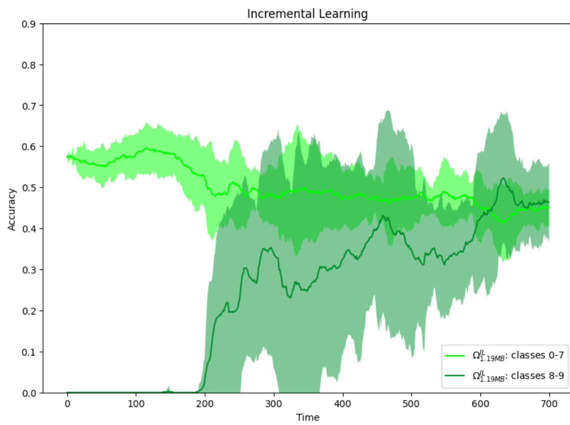
(a) Incremental Learning scenario: means comparison.



(b) $\Omega^{IL}_{Baseline}(\bullet)$.



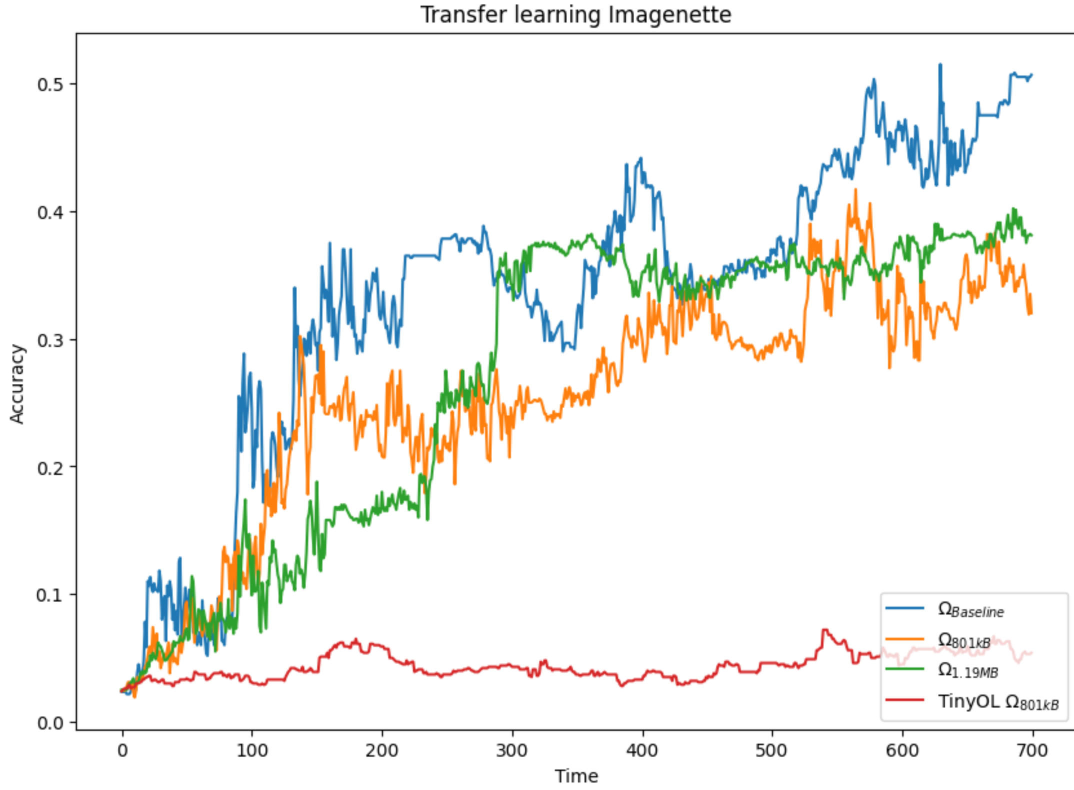
(c) $\Omega^{IL}_{954kB}(\bullet)$.



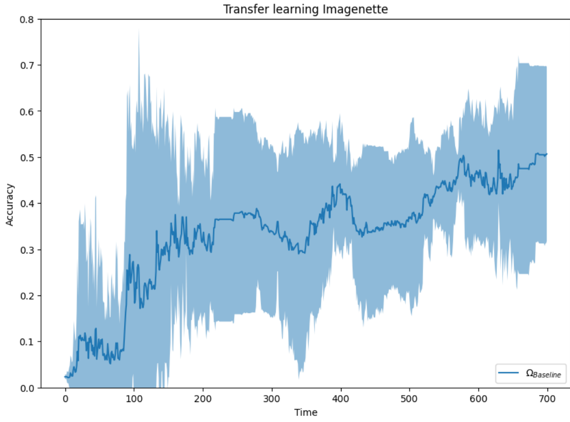
(d) $\Omega^{IL}_{1.19MB}(\bullet)$.

(e) TinyOL $\Omega^{IL}_{954kB}(\bullet)$.

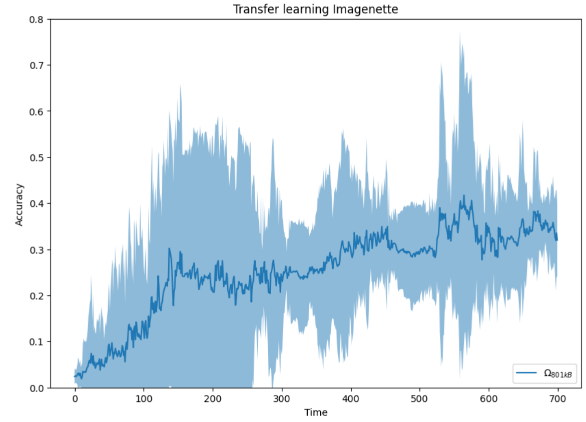
Figure 16: Incremental Learning scenario results.



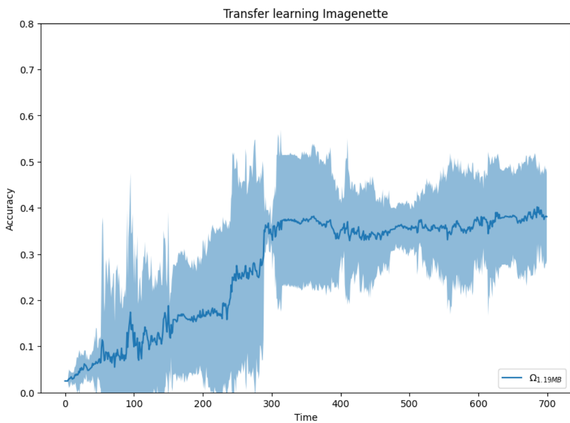
(a) Transfer Learning scenario: means comparison.



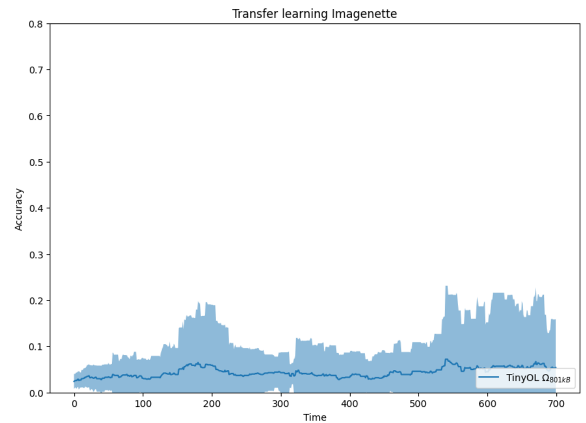
(b) $\Omega_{Baseline}(\bullet)$.



(c) $\Omega_{801kB}(\bullet)$.



(d) $\Omega_{1.19MB}(\bullet)$.



(e) TinyOL $\Omega_{801kB}(\bullet)$.

Figure 17: Transfer Learning scenario results.

training the model on all 10 classes (79.29%), is distinctly noticeable. It is worth to mention that networks that underwent a more substantial training were able to achieve higher accuracies on classes 0-7, but exhibite challenges when attempting to learn classes 8-9. The incremental models used for the experiment have been derived consequently derived from $\Upsilon^{IL}(\bullet)$. Considering the distinctions in the training dataset, the largest compression achievable without introducing any significant drop in accuracy provides an incremental model $\Omega^{IL}_{954kB}(\bullet)$ of size 954kB. The other models evaluated in this experiment are $\Omega^{IL}_{954kB}(\bullet)$, the non-quantized TyBox incremental version of $\Upsilon^{IL}(\bullet)$, the TinyOL version of $\Omega^{IL}_{954kB}(\bullet)$, and $\Omega^{IL}_{1.19MB}(\bullet)$, an incremental model produced with our methodology that has been subjected to a less strict pruning than $\Omega^{IL}_{954kB}(\bullet)$.
All the models maintain acceptable performance in the previously learned classes. $\Omega^{IL}_{954kB}(\bullet)$ is able to partially learn the new classes (0.353 $\pm$ 0.047) but struggles to reach a level of accuracy close to the old classes (0.502 $\pm$ 0.008). On the other hand, both $\Omega^{IL}_{1.19MB}(\bullet)$, benefitting from a more relaxed pruning, and $\Omega^{IL}_{Baseline}(\bullet)$, manage to achieve classification accuracies on the new classes (0.464 $\pm$ 0.047, 0.518 $\pm$ 0.089) comparable to those of the old classes (0.452 $\pm$ 0.022, 0.422 $\pm$ 0.021). In contrast, the TinyOL solution is only able to partially learn the new classes (0.243 $\pm$ 0.240) due to its limited buffer size.

5.4.3 Transfer learning

To test the incremental learning ability of the proposed solution, we initially $\Upsilon(\bullet)$ on CIFAR-10 and we applied the incremental learning procedure on Imagenette. This dataset has been divided into training (700 samples) and testing (200 samples). Results for this experiment are illustrated in Fig. 17

$\Omega_{Baseline}(\bullet)$ (0.505 $\pm$ 0.092), as well as $\Omega_{801kB}(\bullet)$ (0.320 $\pm$ 0.041) and $\Omega_{1.19MB}$ (0.381 $\pm$ 0.047), is able to learn the Imagenette dataset over time, although the two TyBox incremental models do not manage to attain results comparable to those achieved by $\Omega_{Baseline}(\bullet)$. This degradation in performance is undeniably related to the severity of the compression executed. Lastly, given their constrained buffer size, the TinyOL solution fail to effectively learn the new dataset (0.021 $\pm$ 0.020).

6. Conclusions

The aim of this work was to design a methodology to integrate Neural Architecture Search with tiny incremental on-device learning. At the time of delivery, this is the first time NAS has been considered for incremental on-device learning.
Starting from a Constrained NAS and by making use of the TyBox toolbox, it has been possible to desgin for an incremental version of a CNAS subnetwork that conforms with the restrictions of tiny devices. This was achieved by applying structured pruning and by extending TyBox with full-integer quantization. Moreover, the code for deploying the designed incremental network has been automatically generated. Automatic code-generation integrates very well with NAS, since it bridges the gap between research of an architecture and its deployment.

The effectiveness and efficiency of the proposed methodology were evaluated across diverse application scenarios. Comparative analyses were conducted against the original CNAS model and an alternate incremental on-device toolbox. Measurements showed that solutions designed with our methodology achieve performance comparable to those of the original CNAS model, and consistently outperforms the incremental models produced by the alternative toolbox.

Future directions encompasses several aspects. Firstly, the accuracy drop deriving from the baseline model conversion needs to be addressed. Achieving a propoer conversion of the complex non-linear CNAS-designed architecture from PyTorch to TensorFlow is crucial to better evaluate the effectiveness of the proposed methodology. Secondly, the impacts of alternative compression techniques should be explored. Full-integer quantization is a natural evolution for the TyBox toolbox, as explained in [38], and structured pruning has been selected given its compatibility with quantization. However, investigating the potential benefits of additional compression techniques could help to further optimize the model in terms of occupation. Lastly, an integration of the proposed methodology in the CNAS search strategy should be considered. This integration has the potential to fully automate the design and deployment of networks, streamlining the process and enhancing efficiency in model development and adaptation.

References

[1] Pareto front. `https://en.wikipedia.org/wiki/Pareto_front`, Accessed: 2023-09-04.

[2] Pytorch hard swish. https://pytorch.org/docs/stable/generated/torch.nn.Hardswish.html, Accessed: 2023-09-04.

[3] Tensorflow swish. https://www.tensorflow.org/api_docs/python/tf/keras/activations/swish, Accessed: 2023-09-04.

[4] Cifar-10 benchmark. https://paperswithcode.com/dataset/cifar-10, Accessed: 2023-09-10.

[5] Cifar-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html, Accessed: 2023-09-10.

[6] Onnx. https://github.com/onnx/onnx, Accessed: 2023-09-10.

[7] Tensorflow prunelowmagnitude function. https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/sparsity/keras/prune_low_magnitude, Accessed: 2023-09-10.

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

[9] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[10] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Practical neural network performance prediction for early stopping. *CoRR*, abs/1705.10823, 2017.

[11] Han Cai, Chuang Gan, and Song Han. Once for all: Train one network and specialize it for efficient deployment. *CoRR*, abs/1908.09791, 2019.

[12] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations, 2023.

[13] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[14] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. *CoRR*, abs/1812.08934, 2018.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[17] Simone Disabato and Manuel Roveri. Incremental on-device tiny machine learning. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, AIChallengeIoT '20, page 7–13, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Simone Disabato and Manuel Roveri. Tiny machine learning for concept drift. *CoRR*, abs/2107.14759, 2021.

[19] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, 2015.

[20] Alessandro Falcetta and Manuel Roveri. Privacy-preserving deep learning with homomorphic encryption: An introduction. *Comp. Intell. Mag.*, 17(3):14–25, aug 2022.

[21] Fast.ai. Imagenette. https://github.com/fastai/imagenette, Accessed: 2023-09-10.

[22] Matteo Gambella and Manuel Roveri. Searching neural architectures with constraints, 2021.

[23] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[24] Shruthi Gorantala, Rob Springer, and Bryant Gipson. Unlocking the potential of fully homomorphic encryption. *Commun. ACM*, 66(5):72–81, apr 2023.

[25] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[26] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[27] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019.

[28] 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.

[29] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017.

[30] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning, 04 2021.

[31] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[32] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[33] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part I*, volume 11205 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2018.

[34] Zhichao Lu, Kalyan Deb, Erik Goodman, Wolfgang Banzhaf, and Vishnu Boddeti. Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search. 07 2020.

[35] Marc Masana, Xialei Liu, Bartlomiej Twardowski, Mikel Menta, Andrew D. Bagdanov, and Joost van de Weijer. Class-incremental learning: survey and performance evaluation. *CoRR*, abs/2010.15277, 2020.

[36] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[37] Ioan Lucan Orăşan, Ciprian Seiculescu, and Cătălin Daniel Caleanu. Benchmarking tensorflow lite quantization algorithms for deep neural networks. In *2022 IEEE 16th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 000221–000226, 2022.

[38] Massimo Pavan, Eugeniu Ostrovan, Armando Caltabiano, and Manuel Roveri. Tybox: An automatic design and code-generation toolbox for tinyml incremental on-device learning. *ACM Trans. Embed. Comput. Syst.*, jun 2023.

[39] Vikram Ramanathan. Online on-device mcu transfer learning. 2020.

[40] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[41] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions, 2021.

[42] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[43] Murat Sazli. A brief review of feed-forward neural networks. *Communications Faculty Of Science University of Ankara*, 50:11–17, 01 2006.

[44] James Seale Smith, Zachary Seymour, and Han-Pang Chiu. Incremental learning with differentiable architecture and forgetting search, 2022.

[45] Yanan Sun, Handing Wang, Bing Xue, Yaochu Jin, Gary G. Yen, and Mengjie Zhang. Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. *IEEE Transactions on Evolutionary Computation*, 24(2):350–364, 2020.

[46] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[47] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

Abstract in lingua italiana

L'apprendimento incrementale on-device è un promettente campo di ricerca in TinyML. Consiste nella capacità dei modelli di adattarsi a nuovi dati senza dipendere costantemente dalla connettività cloud. TyBox è un toolbox per la progettazione automatica e la generazione di codice di modelli TinyML incrementali on-device che affronta efficacemente questa sfida. Parallelamente, Neural Architecture Search è un potente approccio AutoML che mira a automatizzare la progettazione di un'architettura neurale, col fine di ottimizzarla in accuratezza, rispettando allo stesso tempo i requisiti computazionali per un determinato compito e dataset. Constrained NAS amplia il concetto di Neural Architecture Search, incorporando vincoli ispirati a TinyML per trovare un equilibrio tra prestazioni ed efficienza delle risorse. Tuttavia, si rivolge principalmente a dispositivi mobili con maggiori risorse computazionali, rendendolo inadatto per MCU. L'obiettivo di questo lavoro è progettare una metodologia per integrare Neural Architecture Search con l'apprendimento incrementale on-device. Al momento della consegna, questa è la prima volta che NAS viene considerata per l'apprendimento incrementale on-device. L'approccio prevede la compressione delle reti progettate da CNAS mediante il pruning dei filtri convoluzionali e l'estensione di TyBox con la quantizzazione a numeri interi, allo scopo di creare modelli incrementali che rispettino le restrizioni dei dispositivi di piccole dimensioni. Le prestazioni su classificazione multi-immagine dimostrano l'efficacia della metodologia proposta, mettendo in mostra la competitività dei modelli incrementali compressi rispetto rispetto al modello CNAS originale e a toolbox incrementali alternativi.

Parole chiave: ML, TinyML, apprendimento incrementale, NAS con vincoli, IA integrata