Scenario Based Run-time Switching for Adaptive CNN-based Applications at the Edge

SVETLANA MINAKOVA, Leiden University, The Netherlands
DOLLY SAPRA, University of Amsterdam, The Netherlands
TODOR STEFANOV, Leiden University, The Netherlands
ANDY D. PIMENTEL, University of Amsterdam, The Netherlands

Convolutional Neural Networks (CNNs) are biologically inspired computational models that are at the heart of many modern computer vision and natural language processing applications. Some of the CNN-based applications are executed on mobile and embedded devices. Execution of CNNs on such devices places numerous demands on the CNNs, such as high accuracy, high throughput, low memory cost, and low energy consumption. These requirements are very difficult to satisfy at the same time, so CNN execution at the edge typically involves trade-offs (e.g. high CNN throughput is achieved at the cost of decreased CNN accuracy). In existing methodologies, such trade-offs are either chosen once and remain unchanged during a CNN-based application execution, or are adapted to the properties of the CNN input data. However, the application needs can also be significantly affected by the changes in the application environment, such as a change of the battery level in the edge device. Thus, CNN-based applications need a mechanism that allows to dynamically adapt their characteristics to the changes in the application environment at run-time. Therefore, in this paper, we propose a scenario-based run-time switching (SBRS) methodology, that implements such a mechanism.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Embedded software**.

Additional Key Words and Phrases: convolutional neural networks, run-time adaptation, execution at the edge

ACM Reference Format:
Svetlana Minakova, Dolly Sapra, Todor Stefanov, and Andy D. Pimentel. 2021. Scenario Based Run-time Switching for Adaptive CNN-based Applications at the Edge. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 32 pages. https://doi.org/10.1145/3488718

1 INTRODUCTION

Convolutional neural networks (CNNs) [30] are biologically inspired graph computational models, highly optimized to process large amounts of dimensional data. They have the ability to automatically, effectively and adaptively extract and process high- and low-level abstractions from their input data. These abilities have allowed CNNs to become dominant in various computer vision tasks and natural language processing tasks, such as image classification, object detection, segmentation, and others [22]. Many modern applications, that use CNNs for solving their respective tasks, require the execution of these CNNs at edge devices, such as mobile phones and embedded devices [24, 43]. Examples of

Authors' addresses: Svetlana Minakova, s.minakova@liacs.leidenuniv.nl, Leiden University, Niels Bohrweg 1, Leiden, South Holland, The Netherlands, 2333 CA; Dolly Sapra, d.sapra@uva.nl, University of Amsterdam, Science Park 904, Amsterdam, North Holland, The Netherlands, 1098 XH; Todor Stefanov, Leiden University, Niels Bohrweg 1, Leiden, South Holland, The Netherlands, 2333 CA, t.p.stefanov@liacs.leidenuniv.nl; Andy D. Pimentel, University of Amsterdam, Science Park 904, Amsterdam, North Holland, The Netherlands, 1098 XH.

such applications are: object tracking in drones [9], navigation for self-driving cars [26], street surveillance in wireless cameras [1], and other [24]. Providing execution of CNNs in such applications is challenging due to the high demands placed on the CNNs by both the application and edge device. The most common of these demands are:

(1) **high accuracy**. The CNN should be able to properly perform a task, for which it is designed;
(2) **high throughput**. Typically, the applications, moved to the edge, require CNNs to provide real-time response;
(3) **low memory cost**. Most of the edge devices have a limited amount of memory available;
(4) **low energy cost**. The energy of battery-powered edge devices, like e.g. drones, is also strictly limited.

To ensure that a CNN conforms to the requirements (1) to (4) mentioned above, special techniques such as platform-aware CNN design [3, 5, 8, 21, 28, 32, 35], or CNN compression [2, 13, 15, 27, 29] are utilized. Unfortunately, these techniques typically involve trade-offs between the mentioned requirements [24]. For example, CNN weights compression techniques [2, 15] ensure a low CNN memory cost, but decrease the CNN accuracy. Thus, for a CNN-based application executed at the edge, only a priority subset of these requirements can be highly optimized. The selection of the priority requirements for a CNN-based application is typically performed once, during the CNN design, and remains static during the CNN inference run-time. In practice, these priorities are often affected by the application environment, and can change during the application run-time. For example, a CNN-based road traffic monitoring application, executed on a drone [9], can have different priorities, dependent on the situation on the roads and the level of the device's battery. If the traffic is heavy, the application should provide high throughput and high accuracy to process its input data, which typically means high energy cost. However, during a traffic jam, when the high throughput is not required, or in case the battery of the drone is running low, the application would function optimally by prioritizing energy efficiency over the high throughput. This example shows that CNN-based applications need a mechanism that can adapt their characteristics to the changes in the application environment (such as a change of the situation on the roads or a change of the device's battery level) at the application run-time. Moreover, such a mechanism should provide a high level of responsiveness, e.g., if a drone battery is running low, the CNN-based application, executed on the drone, should switch to an energy-efficient mode as soon as possible. However, to the best of our knowledge, neither existing Deep Learning (DL) methodologies [2, 3, 5, 8, 13, 15, 21, 27, 28, 32, 35] for resource-efficient CNN execution at the edge, nor existing embedded systems design methodologies [23, 36, 44] for execution of run-time adaptive applications at the edge, provide such a mechanism.

Therefore, in this paper, we propose a novel scenario-based run-time switching (SBRS) methodology for CNN-based applications, executed at the edge. In our methodology, we associate a CNN-based application with several scenarios. Every scenario is a CNN, specifically designed to conform to certain application's needs for accuracy, throughput, memory cost, and energy cost (see Section 6). During the application execution, the application environment can trigger the application to switch between the scenarios, thereby adapting the characteristics of a CNN-based application to changes in the application environment. To capture multiple application scenarios and allow for run-time switching between these scenarios, we represent a CNN-based application with SBRS using the novel SBRS Model of Computation (MoC), proposed in Section 7. We note that, being associated with multiple scenarios where every scenario is a CNN, the CNN-based application with SBRS can have high memory cost. As explained above, high memory cost is undesired for applications executed at the edge. To reduce the application memory cost, we introduce, as part of the SBRS MoC, the efficient reuse of components (layers and edges) among the different scenarios, and within every scenario. To ensure high application responsiveness to a scenarios switch request, we propose the SBRS transition protocol (see Section 9).

The SBRS transition protocol specifies switching from the old application scenario to a new application scenario so that both old and new scenarios remain consistent, and the new scenario starts to execute as soon as possible.

Paper contributions

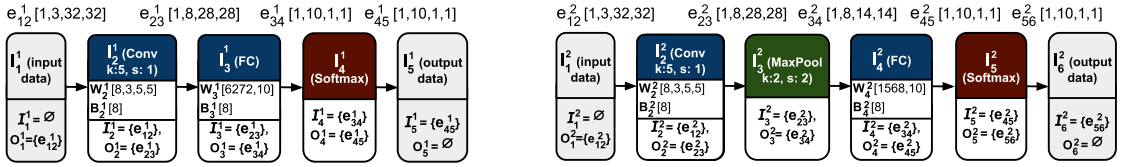
In this paper, we propose a novel scenario-based run-time switching (SBRS) methodology. Our methodology provides run-time adaptation of a CNN-based application, executed at the edge, to changes in the application environment. The SBRS methodology, proposed in Section 5, is our main novel contribution. Other important novel contributions within the methodology, are: 1) An approach for automated derivation of scenarios, associated with a CNN-based application (see Section 6); 2) A SBRS application model, which captures a CNN-based application with several scenarios (see Section 7); 3) An algorithm for automated derivation of a SBRS application model from a set of application scenarios (see Section 8); 4) A transition protocol for efficient switching between the CNN-based application scenarios (see Section 9).

2 RELATED WORK

The platform-aware neural architecture search (NAS) methodologies, proposed in [3, 8, 21, 28, 32, 35] and reviewed in survey [5], allow for automated generation of CNNs that solve the same problem, and are characterized with different accuracy, throughput, energy cost and memory cost. However, these methodologies do not propose a mechanism for run-time switching between these CNNs, while such mechanism is necessary to ensure that application needs are best served at every moment in time. In contrast to the NAS methodologies from [3, 5, 8, 21, 28, 32, 35], our methodology proposes such a mechanism, and ensures that application needs are best served at every moment in time.

The methodologies presented in [12, 14, 16, 25, 31, 34] propose resource-efficient runtime-adaptive CNN execution at the edge. These methodologies represent a CNN as a dynamic computational graph, where for every CNN input sample only a subset of the graph nodes is utilized to compute the corresponding CNN output. The subset of graph nodes is selected during the application run-time by special control mechanisms (e.g., control nodes, augmenting the CNN graph topology). The utilization of only a subset of graph nodes at every CNN computational step can increase the CNN throughput and accuracy, and typically reduces the CNN energy cost. However, the methodologies in [12, 14, 16, 25, 31, 34] cannot adapt a CNN to changes in the application environment, like changes of the device's battery level, which affect the CNN needs during the run-time. The adaptation in these methodologies is driven either by the complexity of the CNN input data [12, 14, 25, 31, 34] or by the number of floating-point operations (FLOPs), required to perform the CNN functionality [12, 16], while the changes in the application environment often cannot be captured in the CNN input data or estimated using FLOPs. In contrast to these methodologies, our SBRS methodology adapts a CNN-based application to the changes in the application environment, and therefore, allows to best serve the application needs, affected by such changes.

A number of embedded systems design methodologies, proposed in [23, 36, 44], allow for efficient execution of runtime-adaptive scenario-based applications at the edge. These methodologies represent an application, executed at the edge, in a specific model of computation (MoC), able to capture the functionality of a runtime-adaptive application associated with several scenarios, and ensure efficient run-time switching between the application scenarios. However, the methodologies in [23, 36, 44] cannot be (directly) applied to CNN-based applications due to a significant semantic difference between the MoCs, utilized in these methodologies and the CNN model [19], typically utilized by CNN-based applications. First of all, the MoCs utilized in [23, 36, 44] lack means for explicit definition of various CNN-specific features, such as CNN parameters and hyperparameters, while, as we show in Section 7, explicit definition of these features is required for the application analysis. Secondly, the MoCs utilized in methodologies [23, 36, 44] are not accepted



(a) CNN^1 (b) CNN^2

Fig. 1. CNN computational model

as input by existing Deep Learning (DL) frameworks, such as Keras [4] or TensorRT [38], widely used for efficient design, deployment and execution of CNN-based applications at the edge. In our methodology, we propose a novel application model, inspired by the methodologies [23, 36, 44], to represent a run-time adaptive CNN-based application and ensure efficient switching between the CNN-based application scenarios. However, unlike the methodologies [23, 36, 44], our methodology 1) explicitly defines and utilizes CNN-specific features for efficient execution of CNN-based applications at the edge, and 2) allows for utilization of existing DL frameworks for design, deployment, and execution of the CNN-based application at the edge.

3 BACKGROUND

In this section, we provide a brief description of the CNN computational model (Section 3.1) and CNN execution at the edge (Section 3.2). This section is essential for understanding the proposed methodology.

3.1 Convolutional Neural Network (CNN)

A convolutional neural network (CNN) is a computational model [22], commonly represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E. An example of a CNN model with $|L| = 5$ layers and $|E| = 4$ edges is given in Figure 1(a). Every layer $l_i \in L$ represents part of the CNN functionality. It performs operator op_i (such as Convolution, Pooling, etc.), parametrized with hyper-parameters hyp_i (such as kernel size, stride, etc.) and learnable parameters par_i (such as weights and biases). Operator op_i of layer l_i accepts as an input the data, provided by the layer's input edges I_i, and produces the result of the data transformation onto its output edges O_i. We define a layer as a tuple $l_i = (op_i, hyp_i, par_i, I_i, O_i)$, where op_i is the operator of l_i; hyp_i are the hyper-parameters of l_i; par_i are the learnable parameters of l_i; I_i and O_i are the input and output edges of l_i, respectively. An example of a CNN layer $l_2^1 = (Conv, \{k : 5, s : 1\}, \{W_2^1, B_2^1\}, \{e_{12}^1\}, \{e_{23}^1\})$ is shown in Figure 1(a). Layer l_2^1 performs Convolutional operator $op_2^1 = Conv$, parametrized with two hyper-parameters (kernel size $k = 5$ and stride $s = 1$) and parameters $par_2^1 = \{W_2^1, B_2^1\}$, where W_2^1 are the layer weights and B_2^1 are the layer biases. Operator op_2^1 accepts as an input the data, provided by input edges $I_2^1 = \{e_{12}^1\}$, and produces output data onto output edges $O_2^1 = \{e_{23}^1\}$.

Every edge $e_{ij} \in E$ specifies a data dependency between layers l_i and l_j, so that data produced by layer l_i is accepted as an input by layer l_j. An example of edge e_{12}^1, which represents a data dependency between layers l_1^1 and l_2^1, is shown in Figure 1(a), where layer l_2^1 accepts as an input the data, produced by layer l_1^1. The data produced and accepted by the CNN layers is stored in multidimensional arrays, called *tensors* [22]. In this paper, every data tensor has the shape $[N, C, H, W]$, where N, C, H, W are the tensor batch size [22], the number of channels, the height and the width, respectively. For example, the data exchanged between layers l_1^1 and l_2^1, shown in Figure 1(a), is stored in tensor [1,3,32,32] with batch size = 1, number of channels = 3, height and width = 32.

3.2 CNN execution at the edge

When executed on an edge device, a CNN utilizes the device memory and computational resources to execute all of its layers L in order, determined by its edges E. Typically, CNN layers are executed in sequential order, i.e., a CNN execution can be represented as $|L|$ computational steps, where at every i-th computational step, CNN layer $l_i \in L$ is executed.

The CNN execution at the edge is typically characterized by **A**ccuracy, **T**hroughput, **M**emory cost, and **E**nergy cost [5, 24, 43], hereinafter referred as **ATME characteristics**. The accuracy, typically measured in percents, characterizes the fraction of correct predictions generated by a CNN from the total number of predictions generated by the CNN. The throughput, typically measured in frames per second (fps), characterizes the speed with which the CNN is able to process input data and produce output data. The memory cost, typically measured in Megabytes (MB), specifies the total amount of memory required to execute a CNN. The energy cost, measured in Joules, specifies the amount of energy consumed by a CNN to process one input frame.

4 MOTIVATIONAL EXAMPLE

In this section, we show the necessity of devising a new methodology for execution of adaptive CNN-based applications at the edge. To do so, we present a simple example of a CNN-based application where the requirements change at run-time due to the changes in its environment. The application is discussed in the context of the existing methodologies reviewed in Section 2, and the scenario-based run-time switching (SBRS), our proposed methodology.

The example application performs CNN-based image recognition on a battery powered unmanned aerial vehicle (UAV). The UAV battery capacity defines a power budget, which is available for both the flight and CNN-based application execution. The distribution of the power budget between the flight and application is irregular, and depends on the weather conditions, which can change during the run-time (the UAV flight). In a calm weather, the UAV requires less power to fly and can thus spend more power on the CNN-based application. Conversely, when the weather is windy, the UAV requires a large amount of power to fly, and therefore has less power available for the CNN-based application. The weather prediction at the application design time is an impossible task. Nevertheless, the CNN-based application should be designed such that it: 1) meets the power constraint, imposed on the application by the UAV battery and affected by weather conditions; 2) demonstrates high image recognition accuracy (the higher the better).

Figure 2 illustrates an example of how the execution of such CNN-based application will transpire, when designed using the existing methodologies and our SBRS. Subplots (a), (b), (c) juxtapose the power available for the application execution (dashed line), against the power used by the application (solid line) during the UAV flight, which lasts 2 hours. The power available for the application execution is dependant on the UAV battery capacity and weather conditions. In this example, we assume that the CNN-based application is allowed to use up to 12 Watts of power in turbulent weather (0 to 0.1 hours and 1.0 to 1.5 hours) and up to 32 Watts of power in calm weather (0.1 to 1.0 hours and 1.5 to 2.0 hours). However, the actual power used by the application is ultimately determined by the application design methodology. Further, the subplots (d), (e), (f) show the image recognition accuracy demonstrated by the application. Subplots (g), (h), (i) show the current charge state (solid line) and minimum charge level (dashed line) of the UAV battery. If the current battery charge reaches the minimum allowed battery level, it may lead to an emergency landing of the UAV.

As a first case, we discuss the multi-objective NAS methodologies [3, 8, 21, 28, 32, 35] for the execution of the example application, that are typically designed and utilized without considering a run-time changing environment. In these methodologies, a CNN is obtained via an automated multi-objective search and characterized with constant accuracy

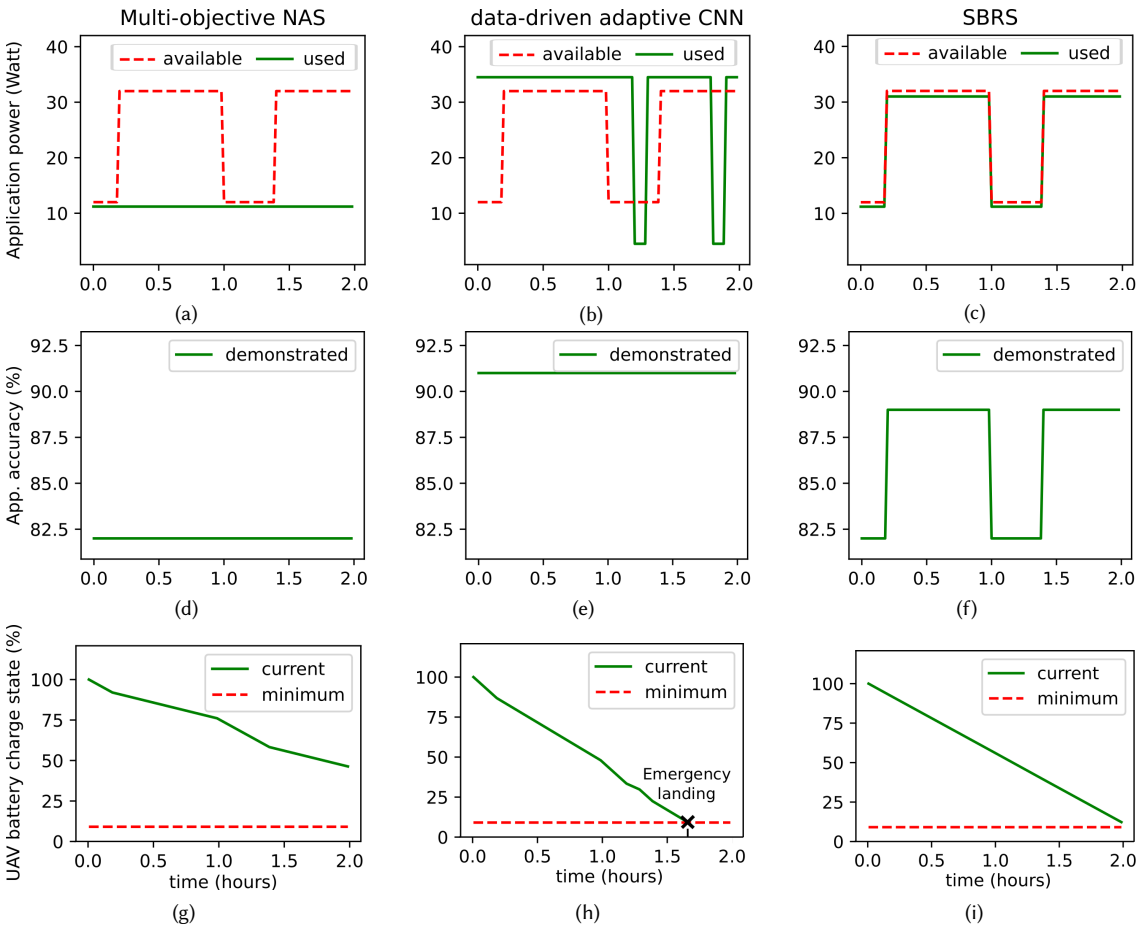


Fig. 2. Execution of a CNN-based application, affected by the application environment and designed using different methodologies

and power consumption. To guarantee that the application meets a power constraint, such a CNN has to account for the worst-case scenario, i.e., when the weather is always windy and therefore only 12 Watts are available for the application execution at any moment. In our illustrative example, such a CNN is characterized with 11.2 Watts of power and 82% accuracy (see Figure 2(a) and Figure 2(d), respectively). As shown in Figure 2(g), when the UAV reaches its destination after 2 hours of flight, it still has ≈50% battery charge left. On the one hand, it means that the application always meets the power constraint. On the other hand, the application could have spent ≈40% remaining UAV battery charge by utilizing a more accurate CNN, though demanding additional power . In other words, *the methodologies in [3, 8, 21, 28, 32, 35] can guarantee that the application meets the given platform-aware constraint, but cannot guarantee efficient use of available platform resources.*

As a second case, when the application is designed using data-driven adaptive methodologies, such as [12, 14, 25, 31, 34], the CNN execution is sensitive to the input data complexity. To process "easy" images, they may use a lower resolution or fewer layers, whereas processing "hard" images requires more computation. In this manner, an adaptive CNN-based application is able to adapt its power consumption depending on the input data complexity, while

demonstrating similar accuracy for all the inputs. However, such a CNN cannot adapt to the changing environmental conditions, which can not be explicitly captured in the input images. The application power consumption can change during the application run-time, based on the input images, although these changes may conflict with the application's requirements, driven by the weather conditions. For example, in Figure 2(b), between 1.0 and 1.25 hours, the CNN consumes significant amount of power despite the necessity to switch to the low power mode. This may lead to increased UAV power consumption over the flight duration and, eventually, to the violation of the application power constraint, causing an emergency landing as illustrated in Figure 2(h). Thus, *the methodologies in [12, 14, 25, 31, 34] are not suitable for CNN-based applications executed at the edge in changing environment, because these can neither properly adapt the application to the environment variations, nor guarantee that the application constantly meets platform-aware constraints.*

Another case of adaptive CNN-based application methodologies, is where the application can adaptively change the number of floating-point operations (FLOPs) spent on the image recognition, such as those in [12, 16]. However, as shown in numerous works [7, 32, 33] FLOPs is an inaccurate indicator for real-world platform-aware characteristics such as power consumption or throughput. These characteristics depend on many other factors, for instance, the ability of the platform to perform parallel computations, time and energy overheads caused by the data transfers, internal hardware limitations, etc. Consequently, the number of FLOPs spent during the application run-time, neither guarantee that the application meets power constraint nor estimate the application efficiency in terms of real-world platform-aware characteristics. In other words, *even though, the methodologies in [12, 16] enable run-time CNN adaptivity, these cannot be directly deployed for applications with real-world platform-aware requirements and constraints.*

To summarize, the existing works lack a methodology to design an adaptive CNN-based application, for real-world platform-aware requirements and constraints, specifically affected by the environment variations at run-time. The motivation behind our current proposal, SBRS, is to enable such run-time adaptivity. To design an application using our SBRS, we perform multi-objective NAS, similar to those in [3, 8, 21, 28, 32, 35]. However, unlike these methodologies, we derive multiple CNNs for each scenario. For example, the first scenario for our example application for windy weather, can have an associated CNN with 11.2 Watts power consumption and 82% accuracy. The second scenario, for calm weather, is represented by a CNN with 31.0 Watts power consumption and 89% accuracy. At run-time, the application switches between these scenarios, based on the weather conditions. Additionally, our methodology explicitly defines the switching mechanism based on triggers generated due to an environment change at run-time. The execution of the CNN-based application with SBRS is shown in Figure 2 (c), (f), (i). Particularly, Figure 2(i) highlights that the application meets the given power constraint, i.e. the UAV battery charge does not go below the minimum level before 2 hours, and SBRS uses all available power to achieve higher application accuracy in comparison with Figure 2(d). Thus, *by switching among the scenarios, SBRS guarantees that a CNN-based application, affected by the environment, meets platform-aware constraints while efficiently exploiting the available platform resources to improve its accuracy.*

5 SBRS METHODOLOGY

In this section, we present our novel scenario-based run-time switching (SBRS) methodology, which allows for run-time adaptation of a CNN-based application, executed at the edge, to changes in the application environment. The general structure of our methodology is given in Figure 3. Our methodology accepts as an input a baseline CNN and one or more requirements sets, associated with the CNN-based application. A baseline CNN is an existing CNN (e.g., AlexNet [22], ResNet [22], or another), proven to achieve good results at solving a CNN-based application task (e.g., classification). The requirements sets describe a scope of needs, associated with the devised application. Every application requirements set $r = (r_a, r_t, r_m, r_e)$ specifies the application priority for high accuracy (r_a), high throughput (r_t), low memory cost (r_m),

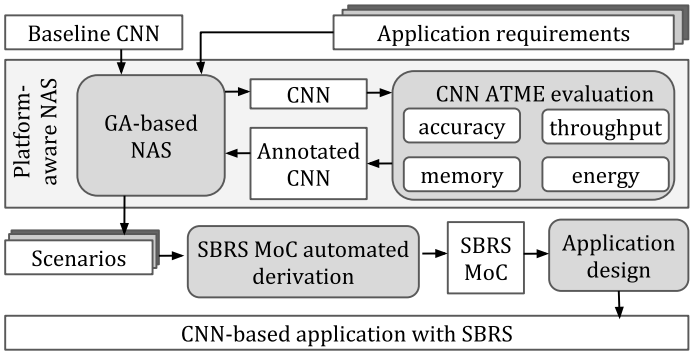


Fig. 3. SBRS methodology

and low energy cost (r_e), respectively. One application can have one or several sets of requirements, characterising the application needs at different times of the application execution. The requirements sets are defined by the application designer at the application design time. As an output, our methodology provides a CNN-based application with SBRS capabilities, able to adapt its characteristics to the changes in the application environment during the application run-time.

Our methodology consists of three main steps, performed offline. At Step 1, for every set of application requirements r, accepted as an input by our methodology, we derive an application scenario, i.e., a CNN which conforms to the given set r of application requirements. To perform this step, we use the automated platform-aware Neural Architecture Search (NAS), explained in detail in Section 6. At Step 2, we use the scenarios generated by Step 1, and the algorithm proposed in Section 8, to automatically derive a SBRS MoC of a CNN-based application with scenarios. The SBRS MoC, proposed in Section 7, captures the scenarios associated with the CNN-based application, and allows for run-time switching among these scenarios. Moreover, the SBRS MoC features efficient reuse of the components (layers and edges) among and within application scenarios, thereby ensuring efficient utilization of the platform memory by the CNN-based application with SBRS. Finally, at Step 3, we use the SBRS MoC derived at Step 2 to design a final implementation of the CNN-based application with SBRS. The final implementation of the CNN-based application performs the application functionality with run-time adaptive switching among the application scenarios, illustrated in Section 4, and following the switching protocol presented in Section 9.

6 SCENARIOS DERIVATION

In this section, we discuss the automated derivation of application scenarios, which essentially generates a collection of CNNs. Each CNN services a different set of requirements, that are determined by its associated scenario. The derivation process builds upon an existing evolutionary Neural Architecture Search (NAS) methodology [42], which searches for the best CNN in terms of a high accuracy only. We extend this NAS algorithm to focus on multiple objectives, namely the ATME characteristics, to arrive at the *pareto front*, which is a set of CNNs with pareto optimality w.r.t. all the given objectives. In a pareto optimal set, none of the objectives can be further improved without worsening some of the other objectives.

Our multi-objective search algorithm is based on an evolutionary approach, which consists of a population of individual CNNs, and the population evolves over multiple iterations. In each iteration, the CNN models are trained on the given dataset and are evaluated against each objective. After all evaluations, the best models found so far are

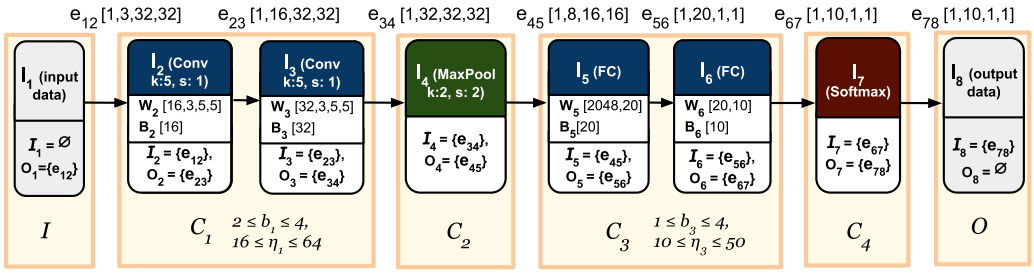


Fig. 4. An example of cluster design from a given baseline CNN. Layers of the same type are grouped into a cluster. The cluster is further made flexible to allow more layers and neurons per layer which are then constrained by definite bounds.

chosen to be parents for the next iteration, which are then altered through genetic operators, to create models for the next iteration. In other words, the models that are not as good as the rest of the population are removed, and replaced by new models created from the better performing ones. In this manner, the design space of possible CNNs is explored in a natural evolution based process. The purpose of doing this iteratively is to slowly improve the population as a whole, where newly selected individuals (the new generation) perform better than the older generation on at least one of the evaluation objectives.

Genotype Creation

Genotype refers to the blueprint of the search space to perform an evolutionary optimization algorithm. All the possible CNN designs are encoded into a genotype to define a general structure of a CNN model architecture, along with bounds and constraints on various parameters. In our current work, this genotype is created using the baseline CNN, which is provided as an input to the SBRS methodology.

The baseline CNN is analyzed first and then split into multiple clusters, each containing consecutive layers of the same type and same feature map size. In a typical CNN, until a feature map size reduction layer, such as maxpool, is encountered, the feature map size can be kept unchanged through optimal padding. Figure 4 illustrates an example of cluster formation for a simple CNN. All the convolutional layers operating in succession, without any maxpool layer, are grouped as one cluster.

The channel depth may vary in a cluster and all its layers, which means that the number of neurons per layer are changeable in any cluster. These clusters are then made flexible and adaptable, by allowing them to have slightly different numbers of layers than the baseline CNN. Moreover, cluster constraints are defined at this step, such as minimum and maximum number of layers in the cluster, along with bounds on the number of neurons per layer. In the example shown in Figure 4, the cluster $C1$ of convolutional layers is now bounded with minimum 2 and maximum 4 layers, where each layer can have between 16 and 64 neurons.

In evolutionary terms, the sequence of clusters along with their bounds define the genotype for the evolutionary NAS. Formally, a genotype, with I and O as input and output layers, can be defined as:

$$Genotype = \{I, C_1, C_2 ... C_l, O\},$$

$$\text{where, Cluster } C_k = \{C_k^{type}, \beta_k^{min}, \beta_k^{max}, \eta_k^{low}, \eta_k^{up}, \pi_k\}$$

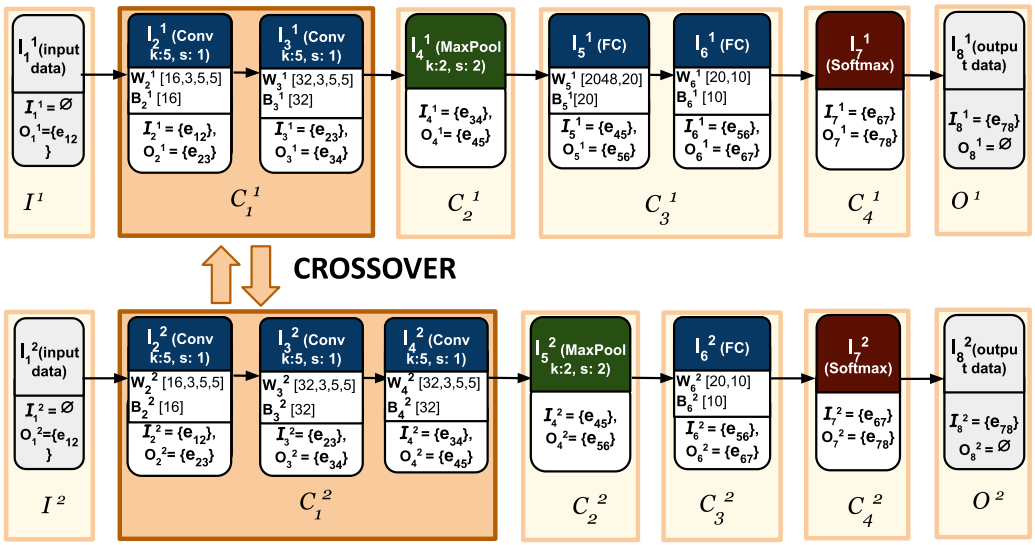


Fig. 5. An example of a crossover operation. The Cluster at position 1 is selected for a crossover between two CNN models. Two Layers in the first CNN are swapped with three layers in the second CNN.

Every cluster C_k in the genotype has layers of the same type defined by C_k^{type}, such as convolution, fully connected or pooling. The bounds on the number of layers in the cluster are specified by β^{min} and β^{max} as minimum and maximum values. This means that if a cluster has b layers, then $\beta_k^{min} \leq b \leq \beta_k^{max}$. The cluster also puts constraints on the number of neurons per layer through η_k^{low} and η_k^{up}, and other possible layer specific parameters, π_k, such as kernel size and stride in a convolutional layer. For a layer l_{ki} in cluster C_k, represented by the tuple $(op_{ki}, hyp_{ki}, par_{ki}, I_{ki}, O_{ki})$, the operator op_{ki} is always the same as C_k^{type}, and its hyper-parameters hyp_{ki} are selected from the parameters specified by π_k. The learnable parameters (weights and biases), par_{ki}, are dependent on the number of neurons in the layer η_{ki} and other hyper-parameters, so $par_{ki} = f(\eta_{ki}, hyp_{ki})$, where $\eta_k^{low} \leq \eta_{ki} \leq \eta_k^{up}$.

To initialize the population, a random selection of CNNs is derived from the genotype definition. Every CNN architecture in the population has exactly the same number of clusters as defined by the genotype, however the number of layers and number of neurons per layer can be randomly polled from the cluster bounds, thus creating a variety of architectures.

The edges defined in the CNN computational model are not explicitly stated in the genotype definition. It is implied that edges between layers of a cluster are an intrinsic part of the corresponding cluster. On the other hand, the edges that connect clusters to each other are external to the cluster definition and are maintained in an unchanged manner during all genetic operations.

Genetic Operators

Various genetic operators are crucial building blocks of any evolutionary algorithm. They not only define how the population moves forward from one iteration to next, but are also crucial in making sure that a maximum design space is explored during the search. We define two genetic operators, namely mutation and crossover, to perform alterations on the CNN models at every iteration.

The mutation operator randomly selects a layer from a randomly selected cluster and one of the parameters is changed by a small value. For example, the mutation can alter the number of neurons in the genotype of the selected convolutional layer. To which extent the mutation can alter the layer in one iteration is defined by algorithm configurations and is simultaneously constrained by the corresponding cluster bounds.

In contrast, a crossover operator selects two individuals from the population and swaps a whole cluster between these two models. The swap occurs for a specific but randomly chosen cluster position. Depending upon the cluster bounds, the number of layers present in the chosen models at the same cluster position, can be vastly different. For instance, as illustrated in Figure 5, a cluster consisting of two convolutional layers in a model, can perform the swap with another cluster containing three convolutional layers in the second model. By replacing a section of the model with a dissimilar number of layers, the algorithm allows for exploration of rather different model structures. However, the crossover operator is disruptive, and more training is needed to recover the loss incurred due to this operation. Crossover in abundance can prevent the algorithm from converging, hence the rate of crossover is reduced as the iterations continue.

CNNs ATME evaluation

In this section, we describe the evaluation of CNN ATME characteristics, explained in Section 3.2, utilized by the platform-aware multi-objective evolutionary NAS.

6.0.1 Accuracy. To evaluate the efficiency of a CNN, we use a state-of-the-art cross-validation technique [39]. In this technique, a CNN efficiency metric is measured by application of a CNN to a special set of data, called validation dataset [39]. The most popularly used metric, CNN accuracy, is computed as the number of correctly processed input frames to the total number of the CNN input frames.

It is important to note that even though we refer to evaluation of a CNN as accuracy, it is possible to use any other evaluation metric suitable to the application. For instance, F-1 score, precision, recall, PR-AUC (Area under curve for precision recall) are some of the metrics used for CNNs for imbalanced datasets.

6.0.2 Memory. The CNN memory cost M is computed as:

$$M = \sum_{l_i \in L} (|par_i| * size_{p \in par} + \sum_{e_{ij} \in O_i} |Y_i| * size_{y \in Y_i}) \tag{1}$$

Where $|par_i|$ is the total number of the learnable parameters of layer l_i; $size_{p \in par}$ is the amount of memory in MB, occupied by one learnable parameter; Y_i is the data tensor, produced by layer l_i onto its every output edge $e_{ij} \in O_i$; $size_{y \in Y_i}$ is the amount of memory in MB, occupied by one element of data in Y_i.

6.0.3 Throughput and Energy. The CNN throughput T is computed as:

$$T = N / \sum_{l_i \in L} t_i \tag{2}$$

where N is the CNN batch size, i.e., the number of frames, processed by every CNN layer l_i [22]; $\sum_{l_i \in L} t_i$ is the time in seconds, required to perform execution of the CNN *CNN(L, E)*, represented as a sequence of $|L|$ computational steps, where at every step a CNN layer $l_i \in L$ is executed (see Section 3.1); t_i is the time required to execute layer $l_i \in L$. Analogously, the CNN energy cost ξ is computed as:

$$\xi = \sum_{l_i \in L} \xi_i / N \tag{3}$$

where ξ_i is the energy cost (in Joules) associated with the execution of CNN layer l_i. We note that execution time t_i and energy cost ξ_i, associated with CNN layer l_i and utilized in Equation 2 and Equation 3, are notoriously hard to evaluate analytically [5]. Therefore, in our methodology, we obtain t_i and ξ_i by performing measurements on the target edge device.

Algorithm

Here, we describe the multi-objective evolutionary NAS Algorithm utilized to obtain the pareto set w.r.t. the ATME characteristics. The partial training of all the models in the population and evolutionary architecture exploration through genetic operators are performed in every iteration. Partial training refers to training for a short interval or using a subset of the total dataset. The partial training techniques allows a CNN architecture to be searched during the training process itself [42]. Algorithm 1 outlines the complete approach.

The algorithm starts with $CreateGenotype()$, creating the genotype from a given baseline CNN. $InitializePopulation()$ then generates a population of neural networks of size N_p using the genotype created and initializes them by training them for an epoch. Afterwards, this iterative algorithm runs for N_g generations.

$Train()$ trains all individuals with randomly selected data from the training dataset for one epoch using τ_{params} training parameters, such as learning rate and batch size. The pareto set $Pareto_{fr}$ is initially an empty set. $EvaluatePopulation()$ evaluates the population using the ATME evaluation parameters as previously described. $NSGAIISelection()$ selects the $(1 - \Omega)\%$ best individuals using non-dominatd sorting of all individuals based on multiple objectives, as defined by the NSGA-II selection algorithm [17]. The pareto set is updated using the best individuals found so far. To keep the population size constant, $\Omega\%$ randomly selected individuals are added back to the pool. $MutatePopulation()$ and $CrossoverPopulation()$ are the evolutionary operators, which select individuals from the population with a selection probability of P_m and P_r, respectively. The population is updated with genetically modified individuals while models

Algorithm 1: Multi-objective Evolutionary NAS

Evolutionary Inputs : N_g, N_p, P_r, P_m, Ω, $CNN_{baseline}$
Training Inputs : τ_{params}

$G_{type} \leftarrow CreateGenotype(CNN_{baseline})$
$\wp_o \leftarrow InitializePopulation(N_p, G_{type})$
$Pareto_{fr} \leftarrow InitializeEmpty()$
for $i \leftarrow 0 \ldots N_g$ **do**
  $\wp_i \leftarrow Train(\wp_{i-1}, \tau_{params})$
  $ATME_i \leftarrow EvaluatePopulation(\wp_i)$
  $\wp_{best} \leftarrow NSGAIISelection(\Omega, \wp_i, ATME_i)$
  $Pareto_{fr} \leftarrow updatePareto(Pareto_{fr}, \wp_{best})$
  $\wp_r \leftarrow randomFrom(\Omega, \wp_i)$
  update $\wp_i \leftarrow \wp_{best} + \wp_r$
  $\wp_{mu} \leftarrow MutatePopulation(\wp_i, P_m)$
  $\wp_{rc} \leftarrow CrossoverPopulation(\wp_i, P_r)$
  $\wp_{remaining} \leftarrow UnchangedPopulation()$
  update $\wp_i \leftarrow \wp_{mu} + \wp_{rc} + \wp_{remaining}$
end
return $Pareto_{fr}$

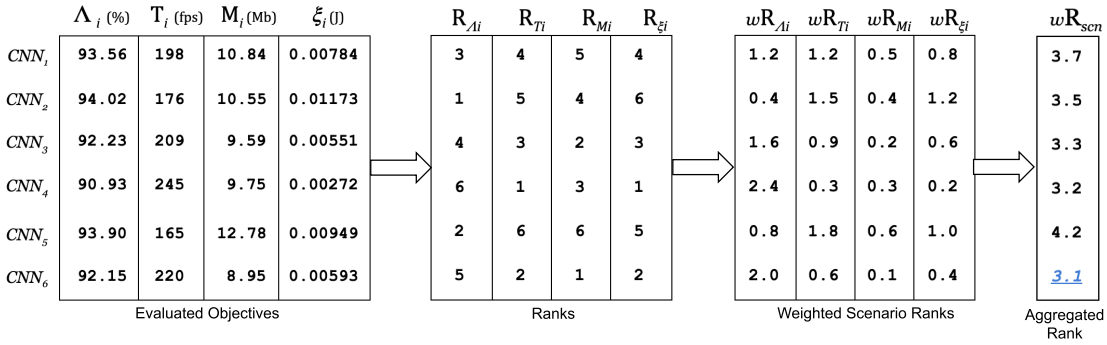


Fig. 6. An example of scenario selection. First, a simple ranking is applied to evaluated objectives. Next, the scenario requirements set ($r_a = 0.4, r_t = 0.3, r_m = 0.1, r_e = 0.2$) is used to compute the weighted ranks for the given scenario. Finally, the aggregated rank is calculated and the model with the lowest rank value (CNN_6) is selected as the model associated with this scenario.

that did not get selected to have an alteration stay in the population unchanged. Finally, when the predefined number of iterations have been performed, the algorithm returns the pareto set (i.e., final pareto front) constructed through all the iterations.

Scenario Selection

The scenario selection task, which follows the pareto set creation, refers to the selection of the appropriate model designated for each scenario. Every intended scenario is depicted by a requirements set $r = (r_a, r_t, r_m, r_e)$, where r_a, r_t, r_m, r_e refers to the importance of accuracy, throughput, memory and energy, respectively. Together, these variables constitute the influence factor of each objective in the scenario by assigning a weight value to the requirements such that $r_a + r_t + r_m + r_e = 1.0$. For example, in a scenario where only high accuracy is pivotal, i.e. $r_a = 1.0$, the requirements set is $r = (1.0, 0, 0, 0)$. However, in a scenario where all the objectives are equally important, the requirements set becomes $r = (0.25, 0.25, 0.25, 0.25)$. For a complex scenario where the throughput and energy are critical factors and accuracy is still moderately significant, the requirements set may be represented as $r = (0.2, 0.4, 0, 0.4)$.

The next task is to post-process all the CNN models in the pareto set, for instance adding *BatchNorm* layers after every *Conv* layer. These CNNs are not fully trained yet by the Algorithm 1, hence they are further trained, to achieve the best possible accuracy. Subsequently, hardware metrics can once more be evaluated at this point, especially if the structure of the CNN was modified, such as by adding or removing some layers. For every CNN model in the pareto set, each objective is separately ranked from 1 to N, where 1 is the best value of an objective (in the set), and N, on the other hand, is the worst. The ranking dominance concept, introduced in [41], has been extended here with weighted aggregation of ranks based on requirements set to derive a suitable CNN model to represent a scenario.

For a model CNN_i, having a rank R_{Oi} for a given objective O, and associated requirement value r_o, its weighted rank wR_{Oi} for the objective in consideration is computed as $r_o * R_{Oi}$. Subsequently, for each scenario, the weighted ranks are aggregated using the following equation

$$wR_{scn} = \sum_{\forall O \in \Theta} (r_o * R_{Oi}) \tag{4}$$

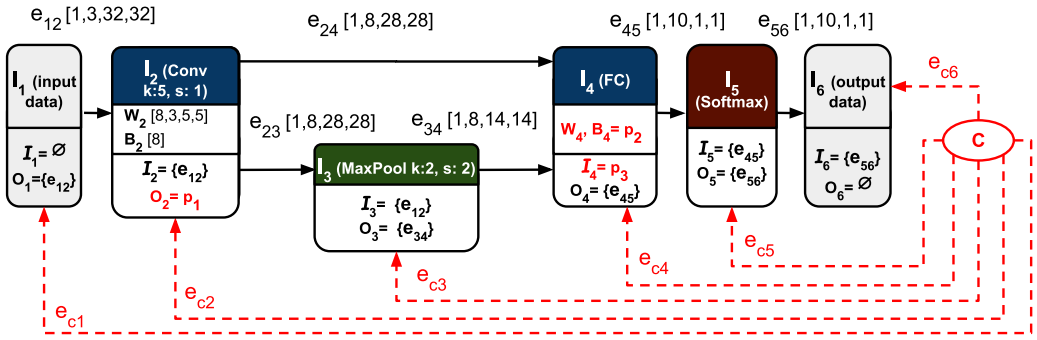


Fig. 7. An example of the SBRS MoC

where Θ is the set of all objectives. For the specific objectives in this work, i.e. Accuracy (Λ), Throughput (T), Memory(M) and Energy (ξ) for a model CNN_i, the equation translates to

$$wR_{scn} = (r_a * R_{\Lambda_i}) + (r_t * R_{T_i}) + (r_m * R_{M_i}) + (r_e * R_{\xi_i}) \tag{5}$$

After the computation of weighted rank, wR_{scn}, for each scenario, the lowest rank value is considered to be the best model representing that scenario. The weighted ranks and their respective aggregation is computed for each scenario in the application. In a situation where two or more models have the lowest rank value, a random model amongst them may be chosen. Alternatively, the ranks can be computed again with a slightly altered requirements set, such as assigning slightly higher importance to the accuracy requirement. Figure 6 exemplifies the process of a scenario selection where the scenario requirements set is ($r_a = 0.4, r_t = 0.3, r_m = 0.1, r_e = 0.2$), i.e., in this scenario all requirements have varying degrees of importance: high accuracy being the most crucial and memory being the least important one.

7 SBRS APPLICATION MODEL

In this section, we propose a SBRS MoC, which models a CNN-based application with scenarios. The SBRS MoC captures multiple scenarios associated with a CNN-based application, and allows for run-time switching among these scenarios. Every scenario in the SBRS MoC is a CNN, as explained in Section 3.1. Figure 7 shows an example of the SBRS MoC, which models a CNN-based application associated with two scenarios: scenario CNN^1 shown in Figure 1(a) and explained in Section 3.1, and scenario CNN^2 shown Figure 1(b). In this section, we use the example from Figure 7 to explain the SBRS MoC in detail. The SBRS MoC is formally defined as a scenarios supergraph, augmented with a control node c and a set of control edges E_c.

The **scenarios supergraph** $G(L, E)$ captures all components (layers and edges) in every scenario $CNN^s(L^s, E^s)$ of a CNN-based application with scenarios. It has a set of layers L, such that every layer l_i^s of every scenario CNN^s is captured by the functionally equivalent layer $l_n \in L$, and a set of edges E, such that every edge e_{ij}^s of every scenario CNN^s is captured by the functionally equivalent edge $e_{nk} \in E$. Table 1 shows the mapping of the components of scenarios CNN^1 and CNN^2, given in Rows 3 and 5 in Table 1, respectively, onto functionally equivalent components of the scenarios supergraph $G(L, E)$ of the SBRS MoC, given in Row 2 in Table 1. For example, Column 5 in Table 1 shows that layer l_3 in the scenarios supergraph captures layer l_3^2 of scenario CNN^2. Analogously, Column 10 in Table 1 shows that edge e_{23} of the scenarios supergraph captures edge e_{23}^2 of scenario CNN^2.

Table 1. Capturing of scenarios' components (layers and edges) in the scenarios supergraph

		layers						edges					
G	component	l_1	l_2	l_3	l_4	l_5	l_6	e_{12}	e_{23}	e_{24}	e_{34}	e_{45}	e_{56}
CNN^1	component	l_1^1	l_2^1	-	l_3^1	l_4^1	l_5^1	e_{12}^1	-	e_{23}^1	-	e_{34}^1	e_{45}^1
	control par.	-	O_2=p_1 ={e_{24}}	-	par_4=p_2= {W_3^1, B_3^1}; I_4=p_3={e_{24}}	-	-	-	-	-	-	-	-
CNN^2	component	l_1^2	l_2^2	l_3^2	l_4^2	l_5^2	l_6^2	e_{12}^2	e_{23}^2	-	e_{34}^2	e_{45}^2	e_{56}^2
	control par.	-	O_2=p_1 ={e_{23}}	-	par_4=p_2= {W_4^2, B_4^2}; I_4=p_3={e_{34}}	-	-	-	-	-	-	-	-
	reuse	op_1, hyp_1, par_1, I_1, O_1	op_2, hyp_2, par_2, I_2	-	op_4, hyp_4, O_4	op_5, hyp_5, par_5, I_5, O_5	op_6, hyp_6, par_6, I_6, O_6	e_{12}	-	-	-	e_{45}	e_{56}

To allow for efficient utilization of platform memory by a CNN-based application with scenarios, the SBRS MoC allows for full or partial reuse of components among the application scenarios. For example, as shown in Column 3 in Table 1, layer l_1 of the scenarios supergraph captures layer l_1^1 of scenario CNN^1 and layer l_1^2 of scenario CNN^2, i.e., layer l_1 of the scenarios supergraph is reused between scenarios CNN^1 and CNN^2. Moreover, as shown in Row 7, Column 3 in Table 1, every attribute of layer l_1 (operator op_i, hyperparameters hyp_1, etc.) is reused between scenarios CNN^1 and CNN^2, i.e., layer l_1 is fully reused between the scenarios. An example of partial reuse is given in Column 6 in Table 1, where layer l_4 of the scenarios supergraph captures layer l_3^1 of scenario CNN^1 and layer l_4^2 of scenario CNN^2. As shown in Row 7, Column 6 in Table 1, only attributes op_4, hyp_4, and O_4 of layer l_4 are reused among the scenarios CNN^1 and CNN^2. The attributes of layer l_4 that are not reused between the scenarios (i.e., par_4 and I_4) are specified via run-time adaptive control parameters, introduced into the scenarios supergraph by the SBRS MoC. For example, as shown in Row 4 and Row 6, Column 6 in Table 1, attributes par_4 and I_4 of supergraph layer l_4 are specified by control parameters p_2 and p_3, respectively. During the application run-time, control parameter p_2 takes values from the set $\{\{W_3^1, B_3^1\}, \{W_4^2, B_4^2\}\}$ and control parameter p_3 takes values from the set $\{\{e_{24}\}, \{e_{34}\}\}$. When $p_2 = \{W_3^1, B_3^1\}$ and $p_3 = \{e_{24}\}$, supergraph layer l_4 is functionally equivalent to layer l_3^1 of scenario CNN^1. When $p_2 = \{W_4^2, B_4^2\}$ and $p_3 = \{e_{34}\}$, supergraph layer l_4 is functionally equivalent to layer l_4^2 of scenario CNN^2.

The **control node** c of the SBRS MoC is a special node that communicates with the application environment, and determines the execution of scenarios in the application supergraph as well as the switching between these scenarios. It defines the execution of every scenario $CNN^s(L^s, E^s)$ associated with the CNN-based application as an execution sequence ϕ^s, functionally equivalent to the execution order of the layers of scenario $CNN^s(L^s, E^s)$ as explained in Section 3.2. Every computational step $\phi_i^s \in \phi^s, i \in [1, |L^s|]$ involves the execution of scenarios supergraph layer l_n, capturing layer l_i^s. If layer l_n is associated with control parameters, step ϕ_i^s specifies values for these parameters such that layer l_n becomes functionally equivalent to layer l_i^s. For example, the execution sequence of scenario CNN^1 is specified as $\phi^1 = \{(l_1, \emptyset), (l_2, \{(p_1, \{e_{24}\})\}), (l_4, \{(p_2, \{W_3^1, B_3^1\}), (p_3, \{e_{24}\})\}), (l_5, \emptyset), (l_6, \emptyset)\}$, where at step $\phi_1^1 = (l_1, \emptyset)$ layer l_1 of the scenarios supergraph, capturing layer l_1^1 of scenario CNN^1, is executed. The $\emptyset$ in step ϕ_1^1 specifies that there are no control parameter values set during the execution of ϕ_1^1; at step $\phi_2^1 = (l_2, \{(p_1, \{e_{24}\})\}$ layer l_2 of the scenarios supergraph is executed with control parameter p_1={e_{24}}, etc.

During the application run-time, control node c can receive a scenario switch request (SSR) from the application environment. The received request can trigger the control node to switch from the current (also called "old") scenario

CNN^o, executed by the node, to a new scenario CNN^n, more suitable for the application needs. The switching from scenario CNN^o to scenario CNN^n is performed under the SBRS transition protocol, which will be explained in Section 9.

The **set of control edges** E_c specifies control dependencies between the control node c and the supergraph layers L. Every control edge $e_{cn} \in E_c$ transfers control data, such as the aforementioned control parameters needed for the layer execution, from control node c to supergraph layer l_n.

8 SBRS MOC AUTOMATED DERIVATION

In this section, we propose an algorithm (see Algorithm 2) that automatically derives the SBRS MoC, as explained in Section 7, from a set of S application scenarios $\{CNN^s\}, s \in [1, S]$, provided by the platform-aware NAS (see Section 6). Algorithm 2 accepts as inputs the set of scenarios $\{CNN^s\}, s \in [1, S]$, and a set of adaptive layer attributes A.

The set A controls the amount of components reuse exploited by the SBRS MoC by explicitly specifying which attributes of the SBRS MoC layers are run-time adaptive. The more layers' attributes are specified in the set A, the more components reuse is exploited by the SBRS MoC. For example, $A = \emptyset$ specifies that the layers of the SBRS MoC have no runtime-adaptive attributes, i.e., only fully equivalent layers (and their input/output edges) are reused among the scenarios. If $A = \{par\}$, in addition to reuse of fully equivalent layers, the SBRS MoC reuses layers that have different parameters (weights and biases) but matching operator, hyperparameters, and sets of input/output edges.

As an output, Algorithm 2 provides an SBRS MoC, which captures application scenarios $\{CNN^s\}, s \in [1, S]$, and exploits components reuse specified by the set A. Figure 7 provides an example of a SBRS MoC, derived using Algorithm 2 for scenarios $\{CNN^1, CNN^2\}$, as shown in Figure 1(a) and Figure 1(b) respectively, and set $A = \{par, I, O\}$ of adaptive layer attributes.

In Lines 1 to 24, Algorithm 2 generates the scenarios supergraph of the SBRS MoC. In Line 1, it defines an empty set of scenarios supergraph layers L, an empty set of scenarios supergraph edges E, an empty set of control parameters Π, and an empty set of reused layers L^{reuse}. In Lines 3 to 9, Algorithm 2 adds layers to the supergraph layers set L. For every layer l_i^s of every scenario CNN^s, Algorithm 2 first checks if set L contains a layer l_n that can be reused to capture layer l_i^s. To perform the check, Algorithm 2 uses Equation 6, which compares those attributes of layers l_i^s and l^n that are not run-time adaptive (i.e., they are not specified in the set of adaptive attributes A). If every of those attributes match, layer l_n is used to capture the functionality of layer l_i^s (Lines 5 to 6 in Algorithm 2). Otherwise, a new layer l, capturing the functionality of layer l_i^s, is added to the scenarios supergraph (Lines 8 to 9 in Algorithm 2).

$$eq(l_i^s, l_n, A) = \begin{cases} true & \text{if } attr_n = attr_i^s, \forall attr \notin A \\ false & \text{otherwise} \end{cases} \tag{6}$$

Analogously, in Lines 10 to 17, Algorithm 2 adds edges to the supergraph edges set E such that 1) every edge e_{ij}^s of every scenario CNN^s is captured in a supergraph edge e_{kn}, and 2) functionally equivalent edges are reused among the scenarios. To check the functional equivalence of a supergraph edge e_{kn} and edge e_{ij}^s of scenario CNN^s, Algorithm 2 uses Equation 7.

$$eq(e_{ij}^s, e_{nk}, A) = \begin{cases} true & \text{if } eq(l_i^s, l_n, A) \wedge eq(l_j^s, l_k, A) \\ false & \text{otherwise} \end{cases} \tag{7}$$

In Lines 18 to 24, Algorithm 2 introduces control parameters into the reused layers of the scenarios supergraph to capture those attributes that cannot be reused among the scenarios. For example, to capture attribute I_4 of scenarios

Algorithm 2: Application model derivation

Input: $\{CNN^s\}, s \in [1, S]; A$
Result: $G(L, E, c, E_c)$

$L \leftarrow \emptyset$; $E \leftarrow \emptyset$; $\Pi \leftarrow \emptyset$; $L^{reuse} \leftarrow \emptyset$;
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  for $l_i^s \in L^s$ **do**
    if $\exists l_n \in L : eq(l_i^s, l_n, A)$ *//Equation 6* **then**
      if $l_n \notin L^{reuse}$ **then**
        $L^{reuse} \leftarrow L^{reuse} + l_n$;
    else
      $l \leftarrow$ new layer $(op_i^s, hyp_i^s, par_i^s, \emptyset, \emptyset)$;
      $L \leftarrow L + l$;
  for $e_{ij}^s \in E^s$ **do**
    if $\nexists e_{kn} \in E : eq(e_{kn}, e_{ij}^s, A)$ *//Equation 7* **then**
      $l_k = l_k \in L : eq(l_i^s, l_k, A)$;
      $l_n = l_n \in L : eq(l_j^s, l_n, A)$;
      $e_{kn} \leftarrow$ new edge (l_k, l_n);
      $E \leftarrow E + e_{kn}$;
      $l_k.O_k \leftarrow l_k.O_k + e_{kn}$;
      $l_n.I_n \leftarrow l_n.I_n + e_{kn}$;
for $l_n \in L^{reuse}$ **do**
  for $attr \in l_n$ **do**
    for $l_i^s \in L^s : eq(l_j^s, l_n, A), s \in [1, S]$ **do**
      $sattr = attr_i^s \in l_i^s : attr_i^s.name = attr.name$;
      if $sattr.value \neq attr.value \wedge attr.value \notin \Pi$ **then**
        $attr =$ new control parameter p;
        $\Pi \leftarrow \Pi + p$;
$\phi \leftarrow \emptyset$; $c \leftarrow$ new control node (ϕ);
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  $\phi^s = \emptyset$;
  for $i \in [1, |L^s|]$ **do**
    $l = l_n \in L : eq(l_i^s, l_n, A)$;
    $P \leftarrow \emptyset$;
    for $attr \in l : attr.value = p_q \in \Pi$ **do**
      $sattr = attr_i^s \in l_i^s : attr_i^s.name = attr.name$;
      if $attr.name = I \vee attr.name = O$ **then**
        $value \leftarrow \emptyset$;
        for $e_{ij}^s \in sattr.value$ **do**
          $e = e_{nk} \in E : eq(e_{ij}^s, e_{nk}, A)$;
          $value \leftarrow value + e$;
      else
        $value = sattr.value$;
      $P \leftarrow P + (p_q, value)$;
    $\phi^s \leftarrow \phi^s + (l, P)$;
  $\phi \leftarrow \phi + \phi^s$;
$E_c \leftarrow \emptyset$;
for $l_n \in L$ **do**
  $e_{cn} \leftarrow$ new control edge (c, l_n);
  $E_c \leftarrow E_c + e_{cn}$;
return $G(L, E, c, E_c)$

supergraph layer l_4, shown in Figure 7, Algorithm 2 introduces control parameter p_3 into layer l_4 (as explained in Section 7).

In Lines 25 to 46, Algorithm 2 augments the scenarios supergraph, derived in Lines 2 to 24, with a control node c and a set of control edges E_c. In Line 25, it defines a control node c with an empty set of execution sequences ϕ. In Lines 26 to 42 it generates execution sequence ϕ^s for every scenario CNN^s, captured by the scenarios supergraph, and adds the sequence ϕ^s to the set ϕ of the control node c. Every computational step $\phi^s_i, i \in [1, |L^s|]$ of the sequence ϕ^s is derived in Lines 28 to 41 of Algorithm 2. In Line 29, Algorithm 2 determines layer l of scenarios supergraph, capturing functionality of layer l^s_i of scenario CNN^s. In Lines 30 to 40, Algorithm 2 derives set P of parameter-value pairs that specifies the values for every control parameter p_q associated with layer l. In Lines 31 to 40, Algorithm 2 visits every attribute $attr$ of layer l, specified as control parameter p_q, and determines the value taken by the parameter p_q (and, therefore, by attribute $attr$) at the execution step ϕ^s_i. In Line 32, Algorithm 2 finds attribute $sattr$ of layer l^s_i, corresponding to the attribute $attr$ of layer l. For example, if attribute $attr \in l$ is a set of parameters par of layer l, Algorithm 2 finds attribute $sattr \in l^s_i$, which is a set parameters par^s_i of layer l^s_i. If attribute $attr$, specified by the control parameter p_q, is a list of input or output edges of layer l (the condition in Line 33 is met), the value for parameter p_q is specified in Lines 34 to 37 of Algorithm 2, as a subset of supergraph edges, functionally equivalent to the corresponding subset of edges in scenario CNN^s. Otherwise, the value of parameter p_q is specified in Line 39 of Algorithm 2 as the value of attribute $sattr$ of layer l^s_i. In Lines 43 to 46, Algorithm 2 creates a set of control edges E_c, such that for every scenarios supergraph layer l_n, set E_c contains a control edge e_{cn}, representing control dependency between layer l_n and the control node c. Finally, in Line 47, Algorithm 2 returns the SBRS MoC, capturing the functionality of every scenario $CNN^s, s \in [1, S]$, associated with the CNN-based application.

9 TRANSITION PROTOCOL

In this section, we present our novel transition protocol, called SBRS-TP, that ensures efficient switching between scenarios of a CNN-based application, represented using the SBRS MoC. As explained in Section 7, the control node c of the SBRS MoC can perform switching from an old application scenario CNN^o to a new application scenario CNN^n, upon receiving a scenario switch request (SSR) from the application environment. In the SBRS MoC, where the execution of scenarios CNN^o and CNN^n is represented using execution sequences ϕ^o and ϕ^n, respectively, switching between scenarios CNN^o and CNN^n means switching between the sequences ϕ^o and ϕ^n. We evaluate the efficiency of such switching by the response delay Δ, defined as the time between a SSR arrival during the execution of the current scenario CNN^o, and the production of the first output data by the new scenario CNN^n. The larger the delay Δ is, the less responsive the application is during a scenarios transition, thus the less efficient the switching is.

The most intuitive way of switching between scenarios CNN^o and CNN^n, hereinafter referred to as naive switching, is to start the execution of the new scenario CNN^n after all computational steps of the old scenario CNN^o are executed. An example of the naive switching is shown in Figure 8(a), where the CNN-based application represented by the SBRS MoC from Figure 7 switches from scenario CNN^1 to scenario CNN^2 upon receiving a SSR at the first execution step of scenario CNN^1. The upper axis in Figure 8(a) shows steps $\phi_i, i \in [1, 11]$, performed by the control node c during the scenarios switching. For example, Figure 8(a) shows that at step ϕ_1 (upon SSR arrival), control node c schedules step ϕ^1_1 of scenario CNN^1 for execution. The lower axis in Figure 8(a) indicates the start and end time of every step ϕ_i performed by the control node c. Every rectangle, annotated with layer l_n in Figure 8(a), shows the time needed to execute layer l_n. The response delay Δ of the naive switching, shown in Figure 8(a), is computed as $18 - 0.5 = 17.5$, where 0.5 is the time of SSR arrival and 18 is the time when scenario CNN^2 produces its first output, i.e., finishes its last step ϕ^2_6.

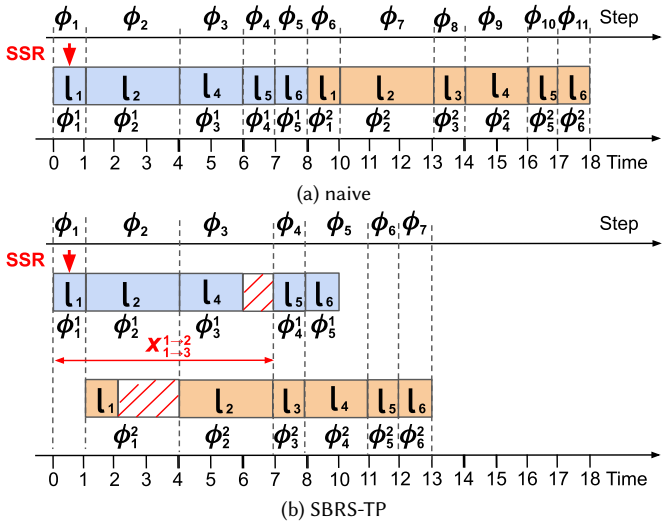


Fig. 8. Switching from scenario CNN^1 to scenario CNN^2

We note that this response delay can be reduced. Figure 8(b) shows an example of an alternative switching mechanism, referred to as the SBRS-TP transition protocol. Unlike in the naive switching, in SBRS-TP, every step $\phi_i^2, i \in [1, 6]$ of the new scenario CNN^2 is executed as soon as possible. For example, step ϕ_1^2 of the new scenario CNN^2 is executed at step ϕ_2, where ϕ_2 is the earliest step after the SSR arrival, at which step ϕ_1^2 can be executed. Step ϕ_1^2 cannot be executed earlier, i.e., at step ϕ_1, due to the components reuse. As explained in Section 7, layer l_1 and the platform resources allocated for execution of this layer are reused between scenarios CNN^1 and CNN^2, and thus cannot be used by scenarios CNN^1 and CNN^2 simultaneously. At step ϕ_1, layer l_1 is used by scenario CNN^1, executing step ϕ_1^1, and therefore, cannot be used for execution of step ϕ_1^2 of scenario CNN^2. However, step ϕ_1^2 of the new scenario CNN^2 can be executed at step ϕ_2, in parallel with step ϕ_2^1 of the old scenario CNN^1, because no components reuse occurs between these steps: step ϕ_2^1 uses layer l_2 for its execution, while step ϕ_1^2 uses layer l_1 (where $l_1 \neq l_2$) for its execution. Analogously, step ϕ_2^2 of the new scenario CNN^2 is executed at step ϕ_3, where ϕ_3 is the earliest step after the SSR arrival, at which step ϕ_2^2 can be executed. As explained in Section 7, according to the execution order adopted by scenario CNN^2, step ϕ_2^2 should be executed after step ϕ_1^2. Thus, in the example shown in Figure 8(b), step ϕ_2^2 should start after step ϕ_2, at which step ϕ_1^2 is executed. Moreover, step ϕ_2^2 of the new scenario CNN^2 cannot be executed at step ϕ_2, because at step ϕ_2 reused layer l_2, required for execution of step ϕ_2^2, is occupied by step ϕ_2^1 of scenario CNN^1. However, step ϕ_2^2 can be executed at step ϕ_3, when layer l_2 that is required for execution of step ϕ_2^2 is not occupied by scenario CNN^1, and step ϕ_1^2 is already executed. The response delay Δ of the switching mechanism shown in Figure 8(b) is $13 - 0.5 = 12.5$, and is much smaller than the response delay $\Delta = 17.5$ of the naive switching shown in Figure 8(a). Thus, the switching mechanism shown in Figure 8(b) is more efficient compared to the naive switching.

Our methodology performs efficient switching between scenarios of a CNN-based application using the SBRS-TP transition protocol, as illustrated in Figure 8(b). The SBRS-TP is carried out in two phases: the analysis phase, and the scheduling phase. The analysis phase is performed during the application design time, for every pair (CNN^o, CNN^n), with $o \neq n$, of the CNN-based application scenarios. During this phase, for every step ϕ_i^n of the new scenario CNN^n, SBRS-TP derives a minimum delay in steps $x_{1 \rightarrow i}^{o \rightarrow n}$ between step ϕ_i^n and the first step ϕ_1^o of the old scenario CNN^o. The

Algorithm 3: SBRS-TP analysis phase

```
   Input: φ^o, φ^n
   Result: X^{o→n}
1  X^{o→n} ← ∅; x = 0;
2  for i ∈ [1, |L^n|] do
3      (l_k, P^n) ← φ_i^n;
4      for φ_j^o ∈ φ^o do
5          (l_z, P^o) ← φ_j^o;
6          if k = z then
7              if j ≥ x then
8                  x = j;
9      X^{o→n} ← X^{o→n} + x;
10     x = x + 1;
11 return X^{o→n}
```

Algorithm 4: SBRS-TP scheduling phase

```
   Input: φ^o, φ^n, X^{o→n}
1  q = 1; i = 1; j = step^o_SSR;
2  wait until step φ_j^o is finished; j = j + 1; q = q + 1;
3  while j ≤ |L^o| do
4      start φ_j^o; j = j + 1;
5      if q ≥ x_{1→i}^{o→n} − step^o_SSR + 2 then
6          start φ_i^n; i = ((i + 1) mod |L^n|);
7      wait until started scenarios' steps are finished; q = q + 1;
8  while i ≤ |L^n| do
9      start φ_i^n;
10     wait until φ_i^n finishes; i = i + 1; q = q + 1;
```

delay $x_{1\rightarrow i}^{o\rightarrow n}$ is computed with respect to the data dependencies within scenarios CNN^o and CNN^n, and the components reuse between these scenarios, as discussed above. An example of delay $x_{1\rightarrow i}^{o\rightarrow n}$ is delay $x_{1\rightarrow 3}^{1\rightarrow 2} = 3$ of step ϕ_3^2, shown in Figure 8(b). Delay $x_{1\rightarrow 3}^{1\rightarrow 2} = 3$ specifies that step ϕ_3^2 of the new scenario CNN^2 cannot start earlier than 3 steps after the first step ϕ_1^1 of the old scenario CNN^1 has started, i.e., earlier than step ϕ_4.

The analysis phase of the SBRS-TP is presented in Algorithm 3. Algorithm 3 accepts as inputs execution sequences ϕ^o and ϕ^n, representing the old scenario CNN^o and the new scenario CNN^n, respectively. As an output, Algorithm 3 provides a set $X^{o\rightarrow n}$, where every element $x_{1\rightarrow i}^{o\rightarrow n} \in X^{o\rightarrow n}$, with $i \in [1, |L^n|]$, is the minimum delay in steps between step ϕ_i^n of the new scenario CNN^n and the first step ϕ_1^o of the old scenario CNN^o. An example of set $X^{o\rightarrow n}$ generated by Algorithm 3 for the scenario switching, shown in Figure 8(b), is the set $X^{1\rightarrow 2} = \{1, 2, 3, 4, 5, 6\}$. In Line 1, Algorithm 3 defines an empty set $X^{o\rightarrow n}$ and a variable x, equal to 0. Variable x is a temporary variable used to store delay $x_{1\rightarrow i}^{o\rightarrow n}$ of every execution step ϕ_i^n in Lines 2 to 10 of Algorithm 3. In Lines 2 to 10, Algorithm 3 visits every step ϕ_i^n of the new scenario CNN^n and computes delay $x_{1\rightarrow i}^{o\rightarrow n}$ associated with this step. In Lines 4 to 8, Algorithm 3 increases delay $x_{1\rightarrow i}^{o\rightarrow n}$, stored in variable x, with respect to the components reuse, as discussed above. It visits every step ϕ_j^o of the old scenario CNN^o, and if step ϕ_j^o and step ϕ_i^n share a reused layer (the condition in Line 6 is met), it delays the execution of step ϕ_i^n until step ϕ_j^o is finished. In Line 9, Algorithm 3 adds the delay of step ϕ_i^n, stored in variable x, to the set $X^{o\rightarrow n}$. In Line 10, Algorithm 3 increases the delay by one step, thereby defining an initial delay for the next step ϕ_{i+1}^n of the new scenario CNN^n. Finally, in Line 11, Algorithm 3 returns the set $X^{o\rightarrow n}$. The set $X^{o\rightarrow n}$ derived using Algorithm 3 for every pair of scenarios (CNN^o, CNN^n) is stored in the control node c of the scenarios supergraph, and used by the scheduling phase of the SBRS-TP at the application run-time.

The scheduling phase of the SBRS-TP is performed by the control node c during the application run-time, upon arrival of an SSR. During this phase, control node c performs switching from the old scenario CNN^o to the new scenario CNN^n, such that the steps of the new scenario CNN^n are executed as soon as possible with respect to the data dependencies within scenario CNN^n and the components reuse between scenarios CNN^o and CNN^n (as discussed above). The scheduling phase of the SBRS-TP is given in Algorithm 4. It accepts as inputs execution sequences ϕ^o and ϕ^n of the old scenario CNN^o and the new scenario CNN^n, respectively, and the set $X^{o\rightarrow n}$ derived by Algorithm 3 for scenarios CNN^o and CNN^n at the SBRS-TP analysis phase. In Line 1, Algorithm 4 defines variables i, j, and q, representing indexes of the current step ϕ_i^n of the new scenario CNN^n, current step ϕ_j^o in the old scenario CNN^o, and current step ϕ_q performed by the control node c, respectively. Upon SSR arrival, $i = 1$, $q = 1$, and $j = step^o_{SSR}$ where $step^o_{SSR} \geq 1$ is the step in the old scenario CNN^o at which the SSR arrived. For the example shown in Figure 8(b), $step^o_{SSR} = 1$ because SSR arrives at step ϕ_1^1 of the old scenario CNN^1. In Line 2, Algorithm 4 performs the first step ϕ_1 of the scenarios switching. During this step, Algorithm 4 waits until step ϕ_j^o, during which the SSR arrived, finishes. In Lines 3 to 7, Algorithm 4 schedules the remaining steps of the old scenario CNN^o, until scenario CNN^o is finished (the condition in Line 3 is false) and, if possible, schedules steps of the new scenario CNN^n in parallel with the steps of the old scenario CNN^o. Step ϕ_i^n of the new scenario CNN^n can start in parallel with step ϕ_j^o of the old scenario CNN^o if the minimum distance $x^{o\rightarrow n}_{1\rightarrow i}$ between steps ϕ_1^o and ϕ_i^n is observed (the condition in Line 5 is met). In Line 7, Algorithm 4 waits until the steps of scenarios CNN^o and CNN^n, started in Lines 4 to 6, finish. In Lines 8 to 10, Algorithm 4 schedules the remaining steps of scenario CNN^n, until scenario CNN^n produces an output data (the condition in Line 8 is false). After Algorithm 4 finishes, scenario CNN^n becomes the current scenario and will be executed for every input given to the CNN-based application until the next SSR.

10 EXPERIMENTAL STUDY

To evaluate our novel SBRS methodology, we perform an experiment, where we apply our methodology to three real-world CNN-based applications with scenarios. We conduct our experiment in four steps. The first three steps perform in-depth per-step analysis of our methodology and demonstrate the merits of our methodology through two real-world CNN-based applications from different domains. The fourth step compares our methodology to the most relevant existing work.

In Step 1 (Section 10.2), we use the platform-aware NAS, explained in Section 6, to automatically derive a set of application scenarios for three CNN-based applications, explained in details in Section 10.1. We show the time required to derive the scenarios, and the ATME characteristics of every derived scenario. By performing this experiment, we evaluate the effectiveness of our platform-aware NAS, and show the diversity of the application scenarios, derived by this approach for the real-world CNN-based applications.

In Step 2 (Section 10.3), we use Algorithm 2, proposed in Section 8, to automatically generate SBRS MoCs for the CNN-based applications, derived at Step 1. For every application, we generate two SBRS MoCs with different sets of adaptive layer attributes A: $A = \{I, O, par\}$ and $A = \{I, O\}$, respectively. We measure and compare the memory cost of every CNN-based application, when the application is represented as 1) the SBRS MoCs with $A = \{I, O, par\}$; 2) an SBRS MoC with $A = \{I, O\}$; 3) a set of scenarios, where every scenario is represented as a CNN model, explained in Section 3.1. By performing this experiment, we evaluate the efficiency of the memory reuse, exploited by the SBRS MoC, proposed in Section 7.

In Step 3 (Section 10.4), we measure and compare the responsiveness of the CNN-based applications, represented as SBRS MoCs, derived in Step 2, during the scenarios switching, when switching is performed: 1) under the SBRS-TP

Table 2. CNN-based applications

App.	task	baseline CNN	dataset	app. requirements sets
Pascal VOC	Image recongition	ResNet [18]	Pascal VOC[20]	r_1=(1.0, 0.0, 0.0, 0.0) r_2=(0.7, 0.0, 0.3, 0.0) r_3=(0.6, 0.1, 0.0, 0.3) r_4=(0.5, 0.5, 0.0, 0.0) r_5=(0.1, 0.1, 0.4, 0.4)
PAMAP2	Human activity monitoring	PAMAP (CNN-2) [10]	PAMAP2 [40]	r_1=(1.0, 0.0, 0.0, 0.0) r_2=(0.2, 0.4, 0.0, 0.4) r_3=(0.5, 0.0, 0.0, 0.5) r_4=(0.5, 0.5, 0.0, 0.0)
CIFAR-10	Image recognition	ResNet [18]	CIFAR-10 [6]	r_1=(1.0, 0.0, 0.0, 0.0) r_2=(0.25, 0.25, 0.25, 0.25) r_3=(0.5, 0.25, 0.0, 0.25) r_4=(0.5, 0.0, 0.0, 0.5)

transition protocol; 2) using the naive switching mechanism. By performing this experiment, we evaluate the efficiency of the SBRS-TP transition protocol, proposed in Section 9.

In Step 4 (Section 10.5), we perform a comparative study, where we compare our SBRS methodology with the most relevant existing work. As explained in Section 2 and demonstrated in Section 4, none of the existing works currently can design an adaptive CNN-based application, which considers platform-aware requirements and constraints that are specifically affected by the environment changes at run-time. Within this context, none of the existing works is completely comparable to our methodology. Nonetheless, we perform a partial comparison between our methodology and the most relevant existing work. Among the existing works, reviewed in Section 2 and Section 4, the MSDNet adaptive CNN work [12] is the most relevant to our methodology. Similarly to our methodology and unlike other reviewed existing work, the methodology in [12] associates a CNN-based application with multiple alternative CNNs that are characterized with different trade-offs between accuracy and resources utilization, and can be used to process application inputs of any complexity. Additionally, both the work in [12] and our methodology provide means to reduce the memory cost of a CNN-based application by reusing the memory among the alternative CNNs. In this sense, the methodology in [12] and our SBRS methodology can be compared via 1) CNNs, designed for a specific dataset and edge platform; 2) run-time adaptive trade-offs between application accuracy and resources utilization; 3) memory efficiency. In Section 10.5, we perform such comparison, using the image recognition CIFAR-10 dataset [6].

10.1 Experimental setup

We demonstrate the merits of our methodology through three applications from two different domains, namely Human Activity Recognition (HAR) and image classification. We used the PAMAP2 [40] dataset for HAR and the Pascal VOC [20] and CIFAR-10 [6] datasets for image classification. PAMAP2 has data from body-worn sensors and predicts the activity performed by the wearer, while Pascal VOC and CIFAR-10 are multi-label image classification datasets with 20 classes and 10 classes, respectively. The sensor data in PAMAP2 is downsampled to 30 Hz and a sliding window approach with a window size of 3s (100 samples) and a step size of 660ms (22 samples) is used to segment the sequences.

The main features and requirements for each CNN-based application are listed in Table 2. Column 1 lists applications names, corresponding to the names of the datasets, the applications are using. Hereinafter, we refer to the applications by their names; Column 2 shows the task performed by the applications; Column 3 lists the baseline CNN that was deployed to perform the application tasks; Column 4 lists the real-world datasets, which were used to train

Table 3. VOC Search Space

Cluster Type	Layers		Neurons		Kernel	
	β^{min}	β^{max}	η^{low}	η^{up}	K^{min}	K^{max}
C_1:Conv	1	3	16	96	3x3	7x7
C_2:MaxP	-	-	-	-	2x2	-
C_3:Conv+Res	1	5	16	96	3x3	7x7
C_4:MaxP	-	-	-	-	2x2	-
C_5:Conv+Res	1	5	32	128	3x3	7x7
C_6:MaxP	-	-	-	-	2x2	-
C_7:Conv+Res	1	5	32	128	3x3	7x7
C_8:MaxP	-	-	-	-	2x2	-
C_9:Conv+Res	1	5	64	256	3x3	7x7
C_{10}:MaxP	-	-	-	-	2x2	-
C_{11}:GlbAvgP	-	-	-	-	2x2	-

Table 4. CIFAR-10 Search Space

Cluster Type	Layers		Neurons		Kernel	
	β^{min}	β^{max}	η^{low}	η^{up}	K^{min}	K^{max}
C_1:Conv	1	3	32	64	3x3	7x7
C_2:Conv+Res	2	4	32	128	3x3	7x7
C_3:MaxP	-	-	-	-	2x2	-
C_4:Conv+Res	2	4	64	256	3x3	7x7
C_5:Conv+Res	2	4	64	256	3x3	7x7
C_6:MaxP	-	-	-	-	2x2	-
C_7:Conv+Res	2	5	128	512	3x3	7x7
C_8:Conv+Res	2	5	128	1024	3x3	7x7
C_9:MaxP	-	-	-	-	2x2	-
C_{10}:FC	1	3	256	1024	-	-

Table 5. PAMAP2 Search Space

Cluster Type	Layers		Neurons		Kernel	
	β^{min}	β^{max}	η^{low}	η^{up}	K^{min}	K^{max}
C_1:Conv	2	7	64	128	3x1	7x1
C_2:MaxP	-	-	-	-	2x1	-
C_3:Conv	2	7	96	256	3x1	7x1
C_4:GlbMaxP	-	-	-	-	2x1	-
C_5:FC	1	4	128	512	-	-

and validate the applications' baseline CNNs; Column 5 shows sets of application requirements $r_i, i \in [1, S]$, where every set r_i characterizes a scenario, associated with the CNN-based application, S is the total number of CNN-based application scenarios. The applications use extremely different baseline CNNs (from the deep and complex ResNet based topology [18] to the small and shallow PAMAP topology) and diverse datasets (from the large Pascal VOC [20] dataset to the small PAMAP2 [40] and CIFAR-10 [6] datasets). The ResNet based baseline topologies for VOC and CIFAR-10 application are custom Resnets, both of which are smaller than the popular ResNet-18. This leads to diversity in scenarios and SBRS MoCs, derived for these applications and, thereby providing a sufficient basis for evaluation of the effectiveness of our methodology.

To explore the design space in our experimental study (Step 1), we first define clusters as derived from the baseline CNNs used for all the datasets. These clusters are shown in Table 3 for the VOC dataset, Table 5 for the PAMAP2 dataset, and Table 4 for the CIFAR-10 dataset. In these tables, Column 1 depicts the cluster-ID with the abbreviated layer types. Conv, MaxP, GlbAvgP, GlbMaxP, FC are abbreviations for convolution, max-pool, global average pool, global max pool and fully connected, respectively. Conv+Res is a special cluster where all layers are convolutional, but there is a residual connection [18] from the input edge to the cluster until the output edge. This residual connection is maintained (or repaired) as needed during the architecture modification through evolutionary operators. The Conv+Res cluster is designed based on the ResNet v1 [18] family of neural networks. Since the CNNs are automatically generated based on the provided constraints by the NAS, they are not identical to any popular ResNet variant, such as, ResNet-18 or ResNet-128. The rest of the columns define cluster specific bounds, namely, the number of layers, the neurons per layer, and the kernel sizes.

Once the clusters are defined, the next step is to perform the multi-objective evolutionary NAS using Algorithm 1 as defined in Section 6. Table 6 lists the values for all parameters of Algorithm 1. Column 1 shows the parameters along

Table 6. Algorithm parameters for DSE

Parameter		VOC	PAMAP2	CIFAR10
Mutation change rate	ϱ_m	0.10	0.12	0.12
Mutation probability	P_m	0.3	0.3	0.3
Initial Crossover probability	$P_r(0)$	0.3	0.4	0.3
Population size	N_p	60	50	100
No of iterations	N_g	30	60	120
Population replacement rate	Ω	0.02	0.03	0.02
Training Parameters	τ_{params}			
Training size per iteration		1 epoch	1/5 epoch	1/8 epoch
Optimizer		Adam	Adam	Adam
Learning rate		$1e^{-3}$	$1e^{-4}$	$1e^{-3}$
Batch size		10	50	64

with their symbol in Column 2. Columns 3, 4 and 5 are the respective parameter values used in the experiments for VOC, PAMAP2 and CIFAR-10.

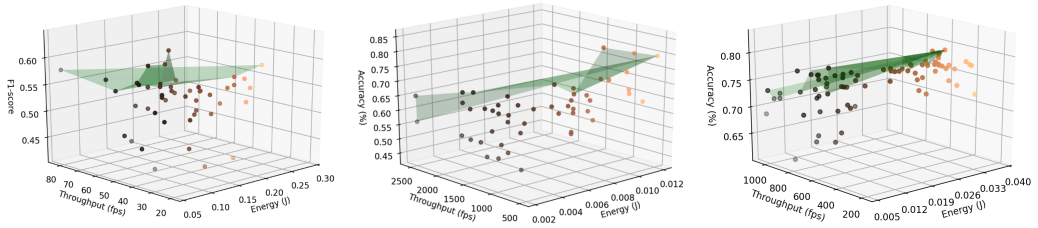
To perform the measurements, required for Step 2 and Step 3 in our experimental study, for every application listed in Table 2, we first use Algorithm 2, explained in Section 7, to automatically derive two SBRS MoCs with different sets of adaptive attributes A. Then for every SBRS MoC, we design an executable application, performing the functionality of the SBRS MoC, and execute this application on the NVIDIA Jetson TX2 embedded platform [37]. To implement the executable applications, we use the TensorRT Deep Learning library [38], providing state-of-the-art performance of deep learning inference on the NVIDIA Jetson TX2 embedded device [37], and custom C++ code. The TensorRT library is used to implement the functionality of CNN layers and edges. The custom C++ code implements the run-time adaptive functionality of the applications.

10.2 Automated scenarios derivation

The scenarios for all the applications were derived using a two step process. First, an exploration of the defined search space was performed using Algorithm 1. This exploration resulted in a pareto front, consisting of CNNs with evaluated objectives, such that an objective can not be improved further without worsening at least one other objective. Figure 9(a), Figure 9(b) and Figure 9(c) illustrate the pareto front for Pascal VOC, PAMAP2 and CIFAR-10, respectively. These pareto fronts do not include memory evaluations to allow for a comprehensible visualization, since the actual pareto fronts created by Algorithm 1 are four dimensional. For the Pascal VOC dataset, which is an imbalanced set, the F1-score was used as the efficiency evaluation metric to compare the partially trained CNNs during the search. The exploration took 6 days with 8 GPUs for the image recognition application (i.e., Pascal VOC dataset). It took 2.5 days on 4 GPUs for the CIFAR-10 dataset, and 10 hours on 1 GPU for the HAR application (PAMAP2 dataset).

The CNNs in the pareto fronts were modified further, by adding a batch normalization layer after every convolutional layer. Subsequently, these models were trained for 250 epochs for Pascal VOC and CIFAR-10 and 100 epochs for PAMAP2. Once the CNNs are trained, all the objectives are evaluated again to make sure they correctly reflect the modifications applied to the CNNs.

Second, all objectives are ranked individually and rank based weighted aggregation was performed, as described in Section 6, using the requirement sets from Table 2 for the three applications. The selected CNNs for each scenario after rank aggregation are presented in Table 7, Table 8, and Table 9 for Pascal VOC, PAMAP2 and CIFAR-10, respectively.



(a) Pascal VOC pareto front (b) PAMAP2 Pareto front (c) CIFAR-10 Pareto front

Fig. 9. Pareto fronts based on 3 evaluation parameters, namely, accuracy (F1-score for Pascal VOC), throughput and energy

Table 7. VOC scenarios

Req. set	PR-AUC	Thr. (fps)	Mem. (MB)	Energy (J)
r_1	77.78	15.41	292.61	0.384
r_2	76.28	21.78	210.69	0.281
r_3	77.69	20.26	242.72	0.291
r_4	73.99	59.27	155.48	0.101
r_5	72.85	75.07	130.21	0.078

Table 8. PAMAP2 scenarios

Req. set	PR-AUC	Thr. (fps)	Mem. (MB)	Energy (J)
r_1	94.17	510.20	10.02	0.0083
r_2	91.34	1333.33	4.30	0.0033
r_3	92.56	970.87	4.86	0.0037
r_4	92.93	1052.63	4.11	0.0039

Table 9. CIFAR-10 scenarios

Req. set	PR-AUC	Thr. (fps)	Mem. (MB)	Energy (J)
r_1	94.86	231.80	52.87	0.0242
r_2	92.84	754.15	13.07	0.0055
r_3	93.46	538.79	18.30	0.0081
r_4	94.46	403.71	28.07	0.0121

The first column in the tables shows the requirements set ID (as already described in Table 2), followed by the evaluation metric, throughput, memory, and energy for the associated CNNs for each scenario. As the evaluation metric, the accuracy was computed for PAMAP2 , and CIFAR-10 , while PR-AUC (Area under precision-recall curve) was used for Pascal-VOC. The PR-AUC is calculated as the average of precision scores calculated for each recall threshold. PR-AUC was chosen over F1-score to evaluate the fully trained CNNs. F1-score is based on threshold based class assignments, and is more useful to perform comparisons between partially trained models (during the NAS). Once a CNN is fully trained, the PR-AUC, which is based on the prediction scores and ordering of these predictions, is more insightful for multi-label classification.

The scenarios that were eventually automatically derived in the experiments, showcase a compelling representation of the application requirements. For instance, the Pascal VOC have contrasting requirements in r_1 and r_5; r_1 demands best possible model efficiency, while on the other hand, r_5 demands low memory and energy usage. In line with the requirements, the scenario for r_1 has the best associated CNN in terms of high PR-AUC score, though with a high memory and energy cost. Whereas, the CNN for r_5 consumes significantly less memory and energy than the former, but with a lower PR-AUC score. In yet another example, if the CNNs for r_1 and r_2 are compared, it is observed that both demand high efficiency, while r_2 additionally demands a lower memory footprint. The scenario that was derived for r_2 requires almost 25% less memory at the cost of a small dip in the PR-AUC score.

For the PAMAP2 application, a similar CNN ensemble with various requirement sets is automatically derived. For example, r_1 and r_2 requirement sets place contradicting demands: r_1 demands higher accuracy, whereas r_2 has more

Table 10. SBRS MoC memory reuse efficiency evaluation

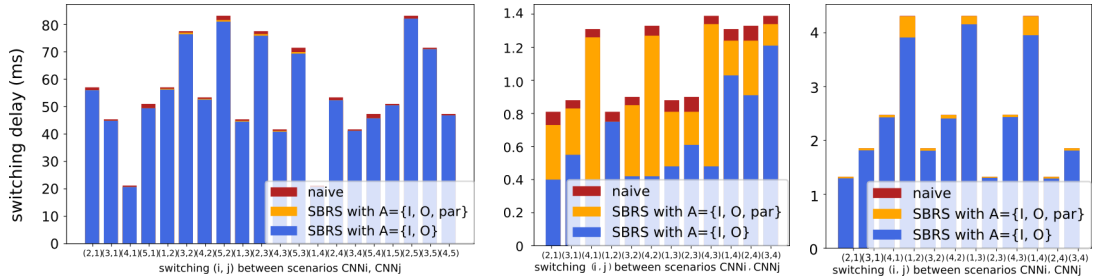
Application	A	Memory use (MB) M^{SBRS}	Memory use (MB) M^{naive}	memory reduction (%)
Pascal VOC	$\{I, O, PAR\}$	230	1032	78
	$\{I, O\}$	547		47
PAMAP 2	$\{I, O, PAR\}$	22.43	23.28	3.64
	$\{I, O\}$	23.21		0.31
CIFAR-10	$\{I, O, PAR\}$	83.3	112.31	25.9
	$\{I, O\}$	107.17		4.57

focus on energy and throughput. The derived CNN for r_1 has high accuracy, while the CNN for r_2 has lower accuracy, but ≈2.5x better throughput and more than halves the energy usage.

Comparably, CNNs are derived for the CIFAR-10 application in the same manner. To illustrate, r_1 and r_2 requirement sets purposefully differ from each other in their demands. r_1 requires high accuracy, whereas r_2 considers all of the measured characteristics to have the same importance. Comparing the derived CNNs for r_1 and r_2, it is clearly observable that r_1 CNN has a high accuracy, while r_2 CNN with a lower accuracy, performs better on all other parameters. These experiments clearly illustrate that our scenario derivation enables automatic generation of diverse CNNs with different ATME characteristics.

10.3 SBRS MoC memory reuse efficiency

In this experiment, we measure and compare the memory cost of every CNN-based application, presented in Table 2 in Section 10, when the application is represented as: 1) an SBRS MoC with a set of adaptive layer attributes $A = \{I, O, par\}$; 2) an SBRS MoC with a set of adaptive layer attributes $A = \{I, O\}$; 3) a set of scenarios, where every scenario is represented as a CNN and no memory is reused within or among the CNNs. The results of this experiment are given in Table 10. In Table 10, Column 1 lists the CNN-based applications with scenarios, explained in Section 10.1. Column 2 shows the sets of adaptive layer attributes A, used by Algorithm 2 to generate the SBRS MoCs for the CNN-based applications. Column 3 shows the memory use M^{SBRS} (in MB) of the CNN-based applications, represented as the SBRS MoCs. As shown in Columns 2 and 3 of Table 10, the more attributes are specified in the set A, the more memory is reused by the application, and the application memory cost is less. For example, as shown in Rows 3-4, Columns 2-3 in Table 10, Pascal VOC uses 230 MB of platform memory, when generated with $A = \{I, O, par\}$ and 547 MB of platform memory, when generated with $A = \{I, O\}$. Column 4 in Table 10 shows the memory use M^{naive} (in MB) of the CNN-based applications, when every application is represented as a set of scenarios and no memory reuse is exploited by the application. Column 5 in Table 10 shows the memory reduction (in %), enabled by the memory reuse, exploited by our proposed SBRS MoC. The memory reduction is computed as $(M^{naive} - M^{SBRS})/M^{naive} * 100\%$, where M^{SBRS} and M^{naive} are listed in Columns 3 and 4, respectively. As shown in Column 5, the memory reuse, exploited by the SBRS MoC, varies for different applications: Pascal VOC (Row 3 to Row 4) demonstrates high (47% - 78%) memory reduction; PAMAP2 (Row 5 to Row 6) demonstrates low (0.31% - 3.64%) memory reduction; CIFAR-10 (Row 7 to Row 8) demonstrates (4.57% - 25.9%) memory reduction, which is higher, compared to PAMAP2 but lower than Pascal VOC. The difference occurs due to the different amounts of components reuse exploited by the Pascal VOC, PAMAP2 , and CIFAR-10 applications . Pascal VOC has 5 scenarios, where every scenario is a deep CNN with a larger number of similar layers. In other words, Pascal VOC is characterised by a large amount of repetitive CNN components, reused by the SBRS MoC (see Section 8), which leads to a significant memory reduction. PAMAP2 has 4 scenarios, compared to 5 scenarios of Pascal VOC, and every scenario in PAMAP2 has less layers and edges than the scenarios of Pascal



(a) Pascal VOC (b) PAMAP2 (c) CIFAR-10

Fig. 10. SBRS-TP efficiency evaluation

VOC. Thus, in PAMAP2, the SBRS MoC can reuse only a small number of components, which leads to a small memory reduction. CIFAR-10 has 4 scenarios, and every scenario in CIFAR-10 has less layers and edges than the scenarios of Pascal VOC, but more layers and edges than the scenarios of PAMAP2. Thus, in CIFAR-10, the SBRS MoC can reuse less components than in Pascal VOC, but more components than in PAMAP2.

10.4 SBRS-TP efficiency

In this experiment, for every CNN-based application, explained in Section 10.1, and represented as two functionally equivalent SBRS MoCs with sets of adaptive attributes $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, we measure and compare the application responsiveness during the scenarios switching, when the switching is performed using: 1) the naive switching mechanism; 2) the SBRS-TP transition protocol. The results of this experiment for Pascal VOC, PAMAP2 and CIFAR-10 are shown as bar charts in Figure 10, subplots (a), (b), and (c), respectively. Every pair (o, n), shown along the horizontal axis in the subplots denotes switching between a pair (CNN^o, CNN^n), $o \neq n$ of the application scenarios, performed upon arrival of a Scenarios Switch Request (SSR) at the first step of the old scenario ($step^o_{SSR}$=1). For example, pair (2, 1) shown in Figure 10(b), denotes switching between scenarios CNN^2 and CNN^1 of PAMAP2, performed at the fist step of scenario CNN^2. Every such switching is associated with 3 bars, showing the switching delay Δ (in milliseconds), when switching is performed: 1) using the naive switching mechanism [1]; 1) using the SBRS-TP for an SBRS MoC with $A = \{I, O, par\}$; 3) using the SBRS-TP for an SBRS MoC with $A = \{I, O\}$. The higher the corresponding bar is (i.e., the larger response delay Δ is), the less efficient is the switching. For example, switching (2, 1), shown in Figure 10(b), is associated with 1) a bar of height 0.8; 2) a bar of height 0.7; 3) a bar of height 0.4. The bar of height 0.8, showing delay Δ of the naive switching, is the highest among the bars. Thus, the switching between scenarios CNN^2 and CNN^1 of PAMAP2 is least efficient, when performed using the naive switching mechanism. The difference in height of bars, corresponding to one switching, shows the relative efficiency of different switching methods expressed via these bars. For example, the switching (2, 1), shown in Figure 10(b), is 0.8 - 0.4 = 0.4 ms less efficient when performed using naive switching (bar of height 0.8) than when performed using SBRS-TP for an SBRS with $A = \{I, O\}$ (bar of height 0.4).

As shown in Figure 10: 1) the switching delay Δ is typically lower when the switching is performed using the SBRS-TP, compared to the switching performed using the naive switching mechanism. Thus, the SBRS-TP is, in general, more efficient than the naive switching mechanism; 2) When the switching is performed under the SBRS-TP, the

[1]One bar is sufficient to show the delay of the naive switching for SBRS MoCs with $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, because, as explained in Section 9, the naive switching is not affected by the application components reuse, determined by the set A

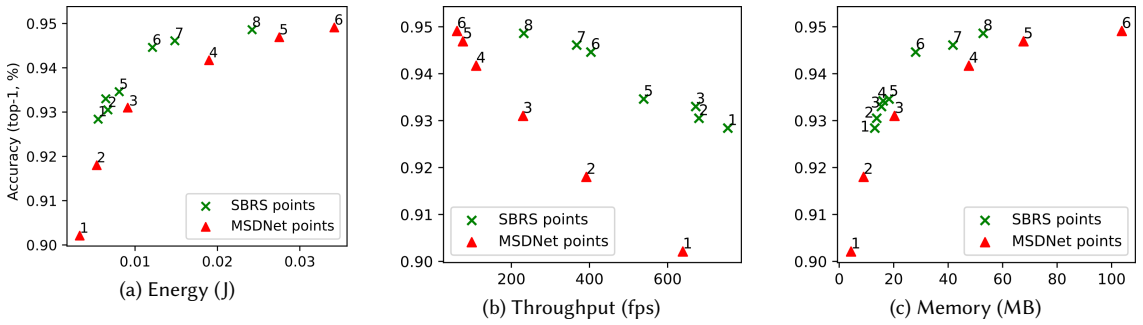


Fig. 11. Comparison among SBRS and MSDNet [12] points

switching delay Δ is typically lower for an SBRS MoC with $A = \{I, O\}$ than for a functionally equivalent SBRS MoC with $A = \{I, O, par\}$. The difference occurs because among these SBRS MoCs, the one with $A = \{I, O, par\}$ typically reuses more CNN components than the one with $A = \{I, O\}$ (see Section 7). As explained in Section 9, reuse of the application components can cause an increase in switching delays, when the switching is performed under the SBRS-TP. Thus, the switching performed under the SBRS-TP is more efficient when performed in an SBRS MoC with $A = \{I, O\}$ than in a functionally equivalent SBRS MoC with $A = \{I, O, par\}$. Analogously, the relative efficiency of the SBRS-TP compared to the naive switching is lower for Pascal VOC than for PAMAP2 or CIFAR-10 because, as explained in Section 10.3, Pascal VOC exploits more components reuse than PAMAP2 or CIFAR-10.

10.5 Comparative study

In this section, we compare our SBRS methodology to the MSDNet adaptive CNN methodology [12]. MSDNet proposes an adaptive CNN-based application which allows multiple exit points in a large neural network, depending upon the input complexity and hardware resources budget allocated to the application. Similarly to our methodology, the methodology in [12] associates a CNN-based application with multiple alternative CNNs that are characterized with different trade-offs between accuracy and resources utilization, and can be used to process application inputs of any complexity. In this sense, the methodology in [12] and our SBRS methodology can be compared via 1) CNNs, designed for a specific dataset and edge platform; 2) run-time adaptive trade-offs between application accuracy and resources utilization; 3) memory efficiency.

First of all, we compare the CNNs, obtained using our SBRS methodology and the MSDNet methodology to perform image classification on the CIFAR-10 dataset [6]. We refer to these CNNs as to SBRS points and MSDNet points, respectively. The MSDNet points, i.e., subgraphs or *exits* of the MSDNet CNN, are derived using the official implementation of the MSDNet methodology [11], executed with design and training parameters specified for the CIFAR-10 dataset in [12]. In total, there are six MSDNet points. The SBRS points are obtained using the platform-aware four-objective NAS, described in Section 6. In total, we obtained eight SBRS points that are pareto-optimal in terms of the ATME characteristics. These points are not the final scenarios as portrayed in Table 9, but the pareto-optimal CNNs resulting from NAS. The scenarios are derived based on a weighted ranking from this pareto set of CNNs, as discussed in Section 6.

To compare the MSDNet points with our SBRS points, we have evaluated the ATME characteristics of all the points on the same hardware. The accuracy characteristic is measured using the cross-validation technique, explained in Section 6.0.1. The platform-aware characteristics (throughput, memory, and energy) are measured on the NVIDIA Jetson TX2 edge platform [37].

The SBRS and MSDNet points comparison is shown in Figure 11. Considering that it is not easy to draw and understand four-dimensional plots, the comparison is represented as three two-dimensional plots, subplots (a), (b) and (c), each comparing one of the platform-aware CNNs characteristics to the CNNs accuracy. The accuracy (the higher the better) is always on the vertical axis with different platform-aware characteristics on the horizontal axis: energy (the lower the better), throughput (the higher the better) and memory cost (the lower the better), respectively. Each subplot shows the six points for MSDNet and those SBRS points that are pareto-optimal in terms of respective platform-aware characteristics.

Beside the visualization, these plots also provide insight into the key difference between our SBRS methodology and MSDNet. It can be clearly observed in Figure 11 that the SBRS points are able to achieve similar accuracy when compared to the MSDNet points, but with lower energy cost, higher throughput, and lower memory cost. We believe that the reason for this direct distinction is caused by the optimization, applied (through the NAS) by our methodology, to every SBRS point to meet the platform-aware needs, while the MSDNet CNN does not provide such optimization. The plots in Figure 11 undoubtedly reveal that our SBRS points are a better choice for using them as scenarios in our SBRS methodology compared to the MSDNet points because none of the MSDNet points pareto-dominates our SBRS points but many of our SBRS points pareto-dominate the MSDNet points.

To further study the efficiency of our proposed methodology, we compare accuracy and throughput characteristics of the MSDNet CNN and the SBRS MoC, both constructed for an example CNN-based application. The example application performs classification on the CIFAR-10 dataset, and is affected by the application environment at run-time.

The MSDNet CNN is constructed according to the design and training parameters specified for the CIFAR-10 dataset in the original MSDNet work [12]. It has six exits, characterized with different accuracy and throughput. During the application run-time, the MSDNet CNN can yield data from different exits, thereby offering various trade-offs between the application accuracy and throughput. We evaluate these trade-offs by executing the MSDNet CNN with an *anytime prediction* setting [12]. This setting allows the MSDNet CNN to switch among its subgraphs (exits), thereby adapting the MSDNet CNN to changes in the application environment. We note that in the original work [12] the switching among the MSDNet CNN exits is driven by a resource budget given in FLOPs, not by a throughput requirement. However, conceptually, it is possible to extend the MSDNet CNN with a throughput-driven adaptive mechanism. In this experiment, we emulate execution of the MSDNet CNN with such a mechanism in order to enable direct comparison of the MSDNet CNN with our SBRS MoC.

The SBRS MoC is obtained by using our methodology, presented in Section 5. As input, our methodology accepts a custom baseline CNN from ResNet [18] family, presented in Table 4, and three sets of application requirements. In the first set $r_1 = \{0.1, 0.9, 0, 0\}$, the application prioritizes high throughput over high accuracy. In the second set $r_2 = \{0.5, 0.5, 0, 0\}$, high throughput and high accuracy are equally important for the application. In the third set $r_3 = \{0.9, 0.1, 0, 0\}$, the application prioritizes high accuracy over high throughput. The obtained SBRS MoC has three scenarios corresponding to the three sets of requirements r_1, r_2, and r_3. During the application run-time the SBRS MoC can switch among its scenarios, thereby offering various trade-offs between application accuracy and throughput, and adapting the application to changes in the application environment at run-time.

The comparison, in terms of accuracy and throughput characteristics of the aforementioned MSDNet CNN and the SBRS MoC, is visualized in Figure 12. The horizontal axis shows throughput (in fps). The vertical axis shows accuracy (in %). The two step-wise curves in Figure 12 represent the relationships between the accuracy and the throughput, exhibited by the MSDNet CNN and SBRS MoC. Each flat segment of the step-wise curves represents a scenario in the SBRS MoC or an exit in MSDNet CNN. For example, the flat segment of the MSDNet curve, characterized with

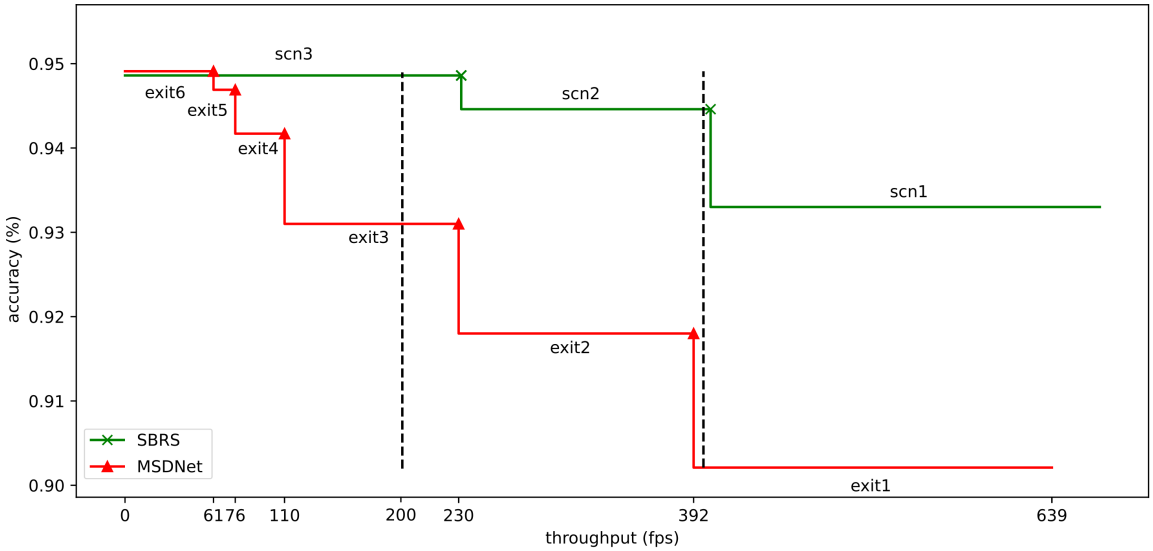


Fig. 12. Comparison between SBRS MoC and MSDNet CNN [12], performing classification on the CIFAR-10 dataset with throughput-driven adaptive mechanism

throughput between 231 and 392 fps and accuracy of 0.918%, represents exit 2 of the MSDNet CNN. Each cross marker or triangle marker represents a switching point between SBRS MoC scenarios or MSDNet CNN exits, respectively. As explained above, run-time switching among the scenarios or exits occurs when the application is affected by changes in its environment at run time. Figure 12 illustrates such changes in the application environment as the two vertical dashed lines, representing demands of minimum throughput, imposed on the application by the environment at run time. For example, at the start of the application execution, the environment demands that the application must have throughput of no less than 200 fps with as high as possible accuracy. In this case, the MSDNet CNN yields data from exit 3, demonstrating 0.931% accuracy, and the SBRS MoC executes in scenario 3, demonstrating 0.949% accuracy. Later, the application environment changes and demands that the application must have throughput of no less than 394 fps. Thus, the MSDNet CNN starts to yield data from exit 1, demonstrating 0.902% accuracy, and the SBRS MoC switches to scenario 2, demonstrating 0.946% accuracy.

As shown in Figure 12, our SBRS MoC exhibits higher accuracy than the MSDNet CNN for any throughput requirement, except when the application has to exhibit throughput lower or equal to 61 fps. In the latter case, the accuracy of our SBRS MoC is comparable (0.05% lower) to the accuracy of the MSDNet CNN. We believe that the difference in accuracy between our SBRS MoC and the the MSDNet CNN occurs because the scenarios in the SBRS MoC are optimized for both high accuracy and high throughput, whereas the exits of MSDNet are only optimized for high CNN accuracy. Optimization for the platform-aware requirements performed during the SBRS MoC design enables for more efficient utilization of the platform resources, and therefore for more efficient execution of the application when high throughput is required.

Finally, we compare the memory efficiency between our SBRS methodology and the MSDNet methodology. To do so, we compare the memory cost of the MSDNet CNN and the SBRS MoC, designed to perform classification on the CIFAR-10 dataset. The memory cost of our final application equals 77.68 MB when the application is designed with adaptive parameters $A = \{I, O, PAR\}$, and 97.6 MB when the application is designed with adaptive parameters

$A = \{I, O\}$. The memory cost of the MSDNet CNN, designed for the CIFAR-10 dataset, is estimated as explained in Section 6.0.2, and is equal to 103.76 MB. Thus, for the CIFAR-10 dataset, the memory efficiency of our methodology is higher than the one of MSDNet. The difference occurs because: 1) unlike the MSDNet methodology, our methodology reuses memory allocated to store intermediate computational results within every CNN as well as among different CNNs; 2) as shown in Figure 11(c), the SBRS points obtained using our methodology and used by our final application require less memory than comparable MSDNet points. It is fair to note that, since our methodology does not enable for reuse of CNN parameters, it may prove less efficient than MSDNet for applications that use CNNs characterized with large sizes of weights. However, such applications are not typical for execution at the edge.

11 CONCLUSION

We have proposed a novel methodology, which provides run-time adaptation for CNN-based applications executed at the edge to changes in the application environment. We evaluated our proposed methodology by designing three real-world run-time adaptive applications in the domains of Human Activity Recognition (HAR) and image classification, and executing these applications on the NVIDIA Jetson TX2 edge device. The experimental results show that for real-world applications our methodology enables: 1) Efficient automated design of CNNs, characterized with different accuracy, throughput, memory cost and energy consumption; 2) A high (up to 78%) degree of platform memory reuse for CNN-based applications that execute CNNs with large amounts of similar components; 3) Efficient switching between the application scenarios, using the novel SBRS-TP transition protocol proposed in our methodology. Additionally, we compared our methodology to the run-time adaptive MSDNet CNN methodology, which is the most relevant to our methodology among the related work. The comparison is performed by CNNs designed for the CIFAR-10 dataset and executed on the Jetson TX2 edge device. The comparison illustrates that the application designed using our methodology outperforms the MSDNet CNN when executed under tight platform-aware requirements, and demonstrates comparable accuracy against the MSDNet CNN when the platform-aware requirements are relaxed. The difference can be attributed to the fact that unlike the MSDNet CNN, our methodology optimizes the application in terms of both high accuracy and platform-aware characteristics.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 Research and Innovation program under grant agreement No. 780788.

REFERENCES

[1] Jungmo Ahn, Jeongyeup Paek, and JeongGil Ko. 2016. Machine learning-based image classification for wireless camera sensor networks. In *IEEE RTCSA*. 103–103.
[2] Brandon Reagen et al. 2018. Weightless: Lossy Weight Encoding For Deep Neural Network Compression. *ICLR*.
[3] Chi-Hung Hsu et al. 2018. MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning. *ArXiv* abs/1806.10332 (2018).
[4] François Chollet et al. 2015. Keras. https://keras.io.
[5] An-Chieh Cheng et al. 2018. Searching toward Pareto-Optimal Device-Aware Neural Architectures. In *ICCAD*. Association for Computing Machinery. https://doi.org/10.1145/3240765.3243494
[6] Alex Krizhevsky et al. 2013. CIFAR-10 (Canadian Institute for Advanced Research). http://www.cs.toronto.edu/ kriz/cifar.html.
[7] Bichen Wu et al. 2019. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *CVPR*. Computer Vision Foundation / IEEE, 10734–10742. https://doi.org/10.1109/CVPR.2019.01099
[8] Chuan-Chi Wang et al. 2020. PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks. RACS, 13–16.
[9] Christos Kyrkou et al. 2018. DroNet: Efficient convolutional neural network detector for real-time UAV applications. In *DATE*. 967–972. https://doi.org/10.23919/DATE.2018.8342149

[10] Fernando Moya Rueda et al. 2018. Convolutional neural networks for human activity recognition using body-worn sensors, Vol. 5. MDPI.
[11] Gao Huang et al. 2018. MSDNet code. https://github.com/gaohuang/MSDNet.
[12] Gao Huang et al. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. ICLR.
[13] Itay Hubara et al. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. 18, 1 (2017), 6869–6898.
[14] Ilias Theodorakopoulos et al. 2017. Parsimonious Inference on Convolutional Neural Networks: Learning and applying on-line kernel activation rules. arXiv:1701.05221 [cs.CV]
[15] Joseph Vinu et al. 2020. A Programmable Approach to Neural Network Compression. *IEEE Micro* 40, 5 (2020), 17–25. https://doi.org/10.1109/mm.2020.3012391
[16] Jiahui Yu et al. 2019. Slimmable Neural Networks. ICLR.
[17] Kalyanmoy Deb et al. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.
[18] Kaiming He et al. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. https://doi.org/10.1109/CVPR.2016.90
[19] Martin Abadi et al. 2017. A Computational Model for TensorFlow: An Introduction. In *MAPL*. ACM, New York, NY, USA. https://doi.org/10.1145/3088525.3088527
[20] Mark Everingham et al. 2012. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.
[21] Mohamed S Abdelfattah et al. 2020. Best of both worlds: Automl codesign of a cnn and its hardware accelerator. In *DAC*. IEEE, 1–6.
[22] Md Zahangir Alom et al. 2018. The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches. *ArXiv* abs/1803.01164 (2018).
[23] Ricardo Bonna et al. 2019. Modeling and Simulation of Dynamic Applications Using Scenario-Aware Dataflow. *ACM TODAES* 24, 5 (2019). https://doi.org/10.1145/3342997
[24] Sergio Branco et al. 2019. Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey. *Electronics* 8, 11 (2019). https://doi.org/10.3390/electronics8111289
[25] Tolga Bolukbasi et al. 2017. Adaptive Neural Networks for Efficient Inference. ICML, 527–536.
[26] Truong-Dong Do et al. 2018. Real-Time Self-Driving Car Navigation Using Deep Neural Network. In *GTSD*. 7–12.
[27] Tien-Ju Yang et al. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. *CVPR* (2017).
[28] Weiwen Jiang et al. 2019. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *DAC*. 1–6.
[29] Yu Cheng et al. 2018. A Survey of Model Compression and Acceleration for Deep Neural Networks. *IEEE Signal Processing Magazine* 35 (2018), 126–136.
[30] Yann LeCun et al. 2015. Deep learning. *Nature* (2015).
[31] Yue Wang et al. 2020. Dual Dynamic Inference: Enabling More Efficient, Adaptive and Controllable Deep Inference. *IEEE Journal of Selected Topics in Signal Processing* (2020).
[32] Yhui Xu et al. 2020. Latency-aware differentiable neural architecture search. *arXiv preprint arXiv:2001.06392* (2020).
[33] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Not All Ops Are Created Equal!. In *SysML*.
[34] Lanlan Liu and Jia Deng. 2018. Dynamic Deep Neural Networks: Optimizing Accuracy-Efficiency Trade-Offs by Selective Execution. In *AAAI*. AAAI Press, 3675–3682.
[35] Mingxing Tan et al. 2020. In *MnasNet: Platform-Aware Neural Architecture Search for Mobile*. ICML.
[36] Orlando Moreira. 2012. *Temporal analysis and scheduling of hard real-time radios running on a multi-processor*. Ph.D. Dissertation. Technical University Eindhoven.
[37] NVIDIA. 2016. Jetson TX2. //https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2.
[38] NVIDIA. 2021. Tensorrt framework. https://developer.nvidia.com/tensorrt.
[39] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. *Cross-Validation*. Springer US, 532–538.
[40] Attila Reiss. 2012. https://archive.ics.uci.edu/ml/datasets/PAMAP2 Physical Activity Monitoring.
[41] Saku Kukkonen and Jouni Lampinen. 2007. Ranking-Dominance and Many-Objective Optimization. In *2007 IEEE Congress on Evolutionary Computation*. 3983–3990. https://doi.org/10.1109/CEC.2007.4424990
[42] Dolly Sapra and Andy D Pimentel. 2020. Constrained evolutionary piecemeal training to design convolutional neural networks. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer.
[43] Mario Vestias. 2019. A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing. *Algorithms* 12, 8 (2019).
[44] Jiali Teddy Zhai, Sobhan Niknam, and Todor Stefanov. 2018. Modeling, Analysis, and Hard Real-Time Scheduling of Adaptive Streaming Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2636–2648. https://doi.org/10.1109/TCAD.2018.2858365