System-level design for efficient execution of CNNs at the edge

Minakova, S.

Chapter 5

Methodology for run-time adaptive inference of CNN-based applications

Svetlana Minakova, Dolly Sapra, Todor Stefanov, Andy Pimentel. "Scenario Based Run-time Switching for Adaptive CNN-based Applications at the Edge". *In ACM Transactions on Embedded Computing Systems (TECS), vol. 21, Iss. 2, Article 14, March 2022.*

IN this chapter, we present our methodology for run-time adaptive inference of CNN-based applications, which corresponds to the third research contribution of this thesis summarized in Section 1.5.3. The proposed methodology is a part of the post-selection optimization component, introduced in Section 1.5, and is aimed at relaxation of Limitation 2, introduced in Section 1.4.2. The reminder of this chapter is organized as follows. Section 5.1 introduces, in more details, the problem addressed by our novel methodology. Section 5.2 summarizes the novel research contributions, presented in this chapter. An overview of the related work is given in Section 5.3. Section 5.4 provides a motivational example. Sections 5.5 to 5.9 present the proposed methodology and its steps. Section 5.10 presents the experimental study performed by using the proposed methodology. Section 5.11 ends the chapter with conclusions.

5.1 Problem statement

As mentioned in Section 1.4.2, a CNN-based application designed using the state-of-the-art design flow shown in Figure 1.3 and explained in Section 1.3,

uses a single CNN to perform its task. This CNN is characterized with certain accuracy and platform-aware characteristics (see Section 1.1) corresponding to requirements posed on the CNN by the application and target edge platform (see Section 1.2). The CNN characteristics remain unchanged during the application run-time. However, the needs of a CNN-based application, and hence the requirements posed on the CNN, may change under the influence of the application environment during the application run-time. For example, a CNN-based road traffic monitoring application, executed on a drone [53], can have different needs, 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 [3, 16, 38, 41, 46, 77, 92, 99, 100, 105, 106] for resource-efficient CNN execution at the Edge, nor existing embedded systems design methodologies [13, 68, 108] for execution of run-time adaptive applications at the edge, provide such a mechanism. Therefore, in this chapter, we propose a novel methodology, which enables to adapt a CNN-based application to changes in the application environment during run-time.

5.2 Contributions

In this chapter, we propose a novel methodology which provides run-time adaptation of a CNN-based application, executed at the Edge, to changes in the application environment. Our methodology, shortly referred as scenario-based run-time switching (SBRS) methodology, is based on the concept of scenarios [15], widely used in embedded systems design. According to this concept, an application can have different internal operation modes, called scenarios, each with its own typical characteristics or/and functionality. During run-time, the application can switch among the scenarios, thereby adapting its

characteristics or functionality to changes in the application environment. In our SBRS methodology a scenario is a CNN designed to conform to a specific set of requirements in terms of accuracy and platform-aware characteristics. During the application execution, the application environment can trigger the application to switch between the scenarios, thereby adapting the application characteristics to changes in the application environment. The SBRS methodology, proposed in Section 5.5, is our main novel contribution. Other important novel contributions within the methodology, are:

- A novel SBRS Model of Computation (MoC) (see Section 5.7). The SBRS MoC captures the functionality of a CNN-based application with multiple scenarios and allows for run-time switching between these scenarios.
- An algorithm for automated derivation of the SBRS MoC from a set of application scenarios (see Section 5.8);
- A transition protocol for efficient switching between the CNN-based application scenarios (see Section 5.9).

5.3 Related Work

The platform-aware neural architecture search (NAS) methodologies, proposed in [3,38,46,92,100,105] and reviewed in survey [16], allow for automated generation of CNNs that solve the same problem, and are characterized with different accuracy and platform-aware characteristic. 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 [3,16,38,46,92,100,105], our methodology proposes such a mechanism, and ensures that application needs are best served at every moment in time.

The methodologies presented in [12,39,61,96,102,107] 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, 39, 61, 96, 102, 107] 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 [12,39,61,96,102] or by the number of floating-point operations (FLOPs), required to perform the CNN functionality [39, 107], 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 [13, 68,108], 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 [13, 68, 108] 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 [2], typically utilized by CNN-based applications. First of all, the MoCs utilized in [13, 68, 108] lack means for explicit definition of various CNN-specific features, such as CNN parameters and hyperparameters, while, as we show in Section 5.7, explicit definition of these features is required for the application analysis. Secondly, the MoCs utilized in methodologies [13, 68, 108] are not accepted as input by existing Deep Learning (DL) frameworks, such as Keras [19] or TensorRT [72], 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 [13, 68, 108], to represent a run-time adaptive CNN-based application and ensure efficient switching between the CNN-based application scenarios. However, unlike the methodologies [13, 68, 108], 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.

5.4 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 5.3, 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 the CNN-based application execution. The distribution of the power budget between the flight and the 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 5.1 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, 38, 46, 92, 100, 105] 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 and power consumption. To

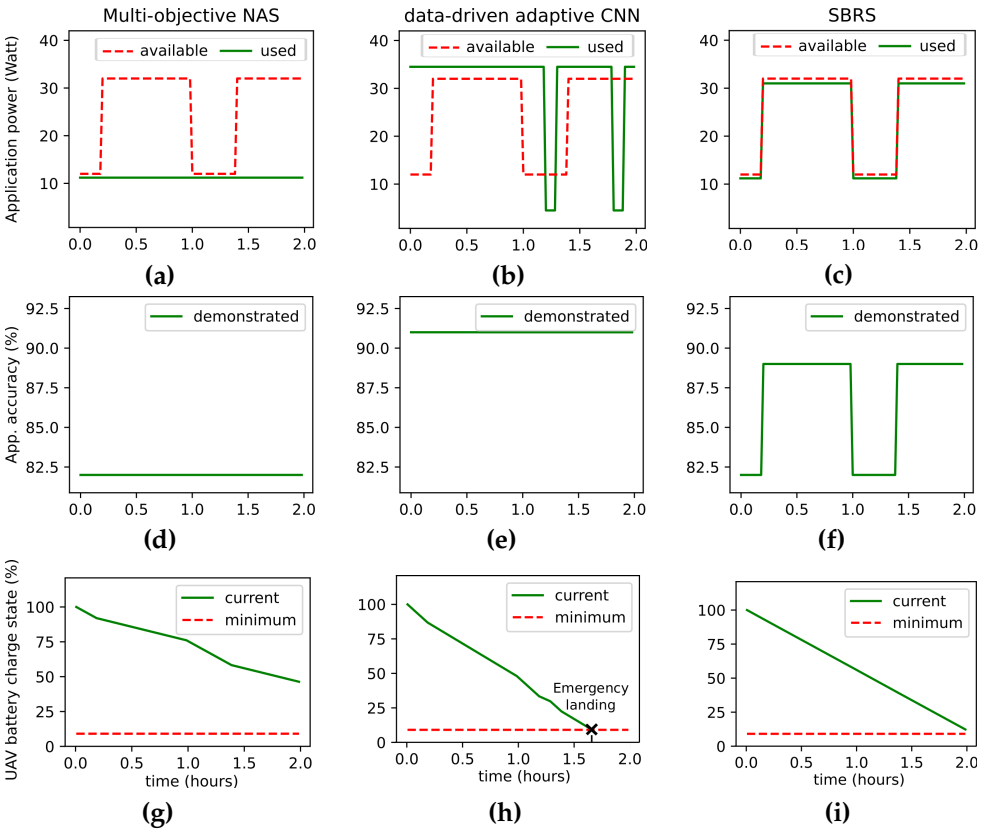


Figure 5.1: *Execution of a CNN-based application, affected by the application environment and designed using different methodologies*

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 5.1(a) and Figure 5.1(d), respectively). As shown in Figure 5.1(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, 38, 46, 92, 100, 105] 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, 39, 61, 96, 102], 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 5.1(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 5.1(h). Thus, *the methodologies in [12, 39, 61, 96, 102] 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 [39, 107]. However, as shown in numerous works [54, 103, 105] 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 [39, 107] 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, 38, 46, 92, 100, 105]. 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 5.1 (c), (f), (i). Particularly, Figure 5.1(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 5.1(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*.

5.5 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 5.2. 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 [4], ResNet [36], 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), 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. In Step 1 (see Section 5.6), for every set of application requirements r, accepted as an input by our method-

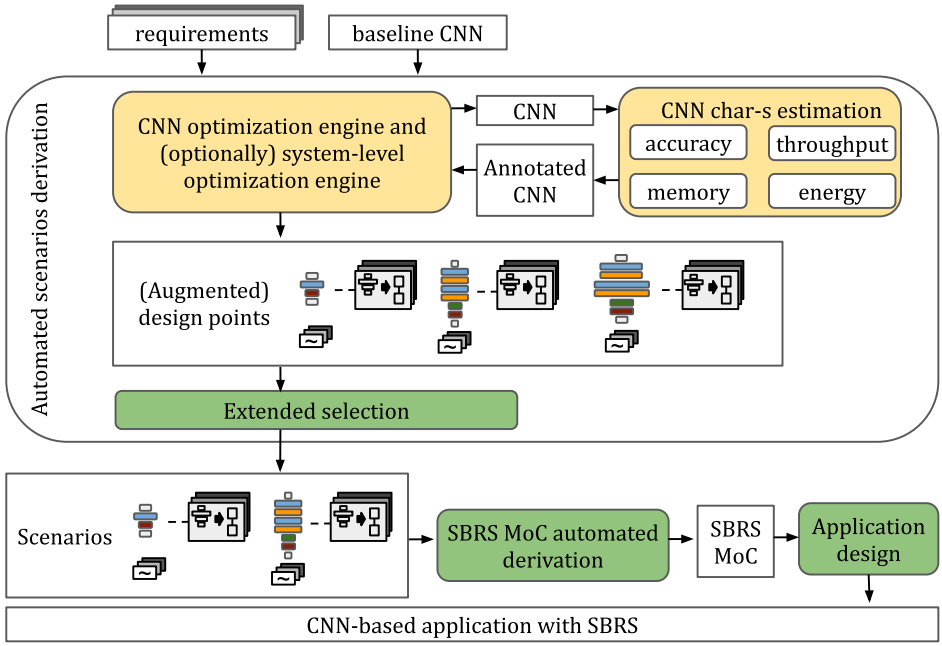


Figure 5.2: *SBRS methodology*

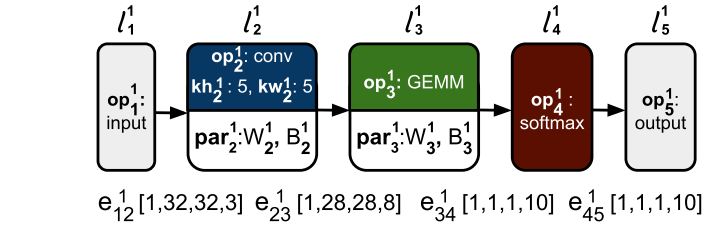
ology, we derive an application scenario, i.e., a CNN which conforms to the given set r of application requirements.

In Step 2, we use the scenarios generated by Step 1, and the algorithm proposed in Section 5.8, to automatically derive a SBRS MoC of a CNN-based application with scenarios. The SBRS MoC, proposed in Section 5.7, 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.

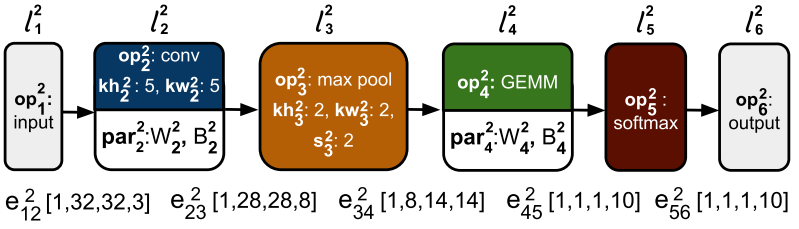
Finally, in Step 3, we use the SBRS MoC derived at Step 2 to design a final implementation of the CNN-based application with SBRS. The final implementation of the CNN-based application performs the application functionality with run-time adaptive switching among the application scenarios, illustrated in Section 5.4, and following the switching protocol presented in Section 5.9.

5.6 Automated scenarios derivation

In this section, we discuss the automated derivation of application scenarios, i.e., CNNs characterized with different accuracy and platform-aware charac-



(a) CNN^1, $r_1 = (1.0, 0, 0, 0)$



(b) CNN^2, $r_2 = (0.2, 0.4, 0, 0.4)$

Figure 5.3: *Application scenarios*

teristics. An example set of $S = 2$ scenarios, derived using our methodology, is shown in Figure 5.3. As mentioned in Section 5.5, every intended scenario $CNN^i, i \in [1, S]$ is first depicted by a user-defined set of requirements $r_i = (r_a, r_t, r_m, r_e)$, where r_a, r_t, r_m, r_e refer to the importance of high CNN accuracy, high CNN throughput, low CNN memory footprint and low CNN energy cost, respectively. Together, these variables constitute the influence factor of each requirement 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 scenario CNN^1 shown in Figure 5.3(a) only high accuracy is pivotal, i.e. $r_a = 1.0$, the requirements set is $r_1 = (1.0, 0, 0, 0)$. For scenario CNN^2 shown in Figure 5.3(b), the throughput and energy are critical factors while accuracy is still moderately significant, and the requirements set is defined as $r_2 = (0.2, 0.4, 0, 0.4)$.

To derive a set of scenarios, depicted by their respective sets of requirements, we use a part of the extended CNN design flow shown in Figure 1.5 and explained in Section 1.5. First, the sets of requirements are passed to the *CNN optimization engine*, introduced in Section 1.3. The CNN optimization engine performs automated search for optimal CNN architecture and weights using techniques such as platform-aware NAS [9, 25, 34, 38, 46, 92, 105] and CNN compression [41, 99, 106]. The search results into a set of CNNs, characterized with different architecture, weights, accuracy, and platform-aware characteristics. The platform-aware characteristics of the CNNs may be further improved by the use of the *system-level optimization engine*, introduced in Section 1.5. Recall, that the system-level optimization engine explores and

exploits alternative manners of CNN execution to improve the CNNs characteristics, and produces as an output a set of *augmented design points*, i.e., CNNs, annotated with a specific manner of execution. Finally, the extended selection component, introduced in Section 1.5, selects scenarios from the (augmented) design points, produced using the CNN optimization engine and (possibly) the system-level optimization engine. The selection of every scenario is based on existing multi-objective ranking algorithms [29], able to score a CNN, based on the CNN accuracy and platform-aware characteristics, and a set of requirements, posed on a CNN.

To estimate the accuracy and platform-aware characteristics, the CNN optimization engine and the system-level optimization engine use the CNN characteristics estimation component, briefly introduced in Section 1.3. In our methodology, the CNN characteristics estimation component provides means to evaluate the CNN accuracy, throughput, memory cost, and energy cost.

To evaluate the *accuracy* of a CNN, we use means of existing DL frameworks, such as Keras [19], Pytorch [75], Tensorlow [1], TensorRT [72] and others [74]. These frameworks offer a wide range of a state-of-the-art techniques for evaluating CNN accuracy. Mainly, we use the widely known cross-validation technique [78]. In this technique, a CNN efficiency metric is measured by application of a CNN to a special set of data, called validation dataset [78]. The 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 estimation of a CNN as accuracy, it is possible to use alternative estimation metrics suitable to the application, and offered by the DL frameworks. For instance, F-1 score, precision, recall, PR-AUC (Area under curve for precision recall) [89] are some of the metrics that can be used for CNNs estimation as well.

To estimate the platform-aware characteristics of a CNN, we use analytical formulas as well as measurements on the platform. The *memory cost* of a CNN is estimated analytically, using Equation 2.5 explained in Section 2.2. To estimate the *CNN throughput and energy cost* that are notoriously hard to evaluate analytically [54, 60, 103, 105], we use direct measurements on the platform.

5.7 SBRS application model

In this section, we propose our 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

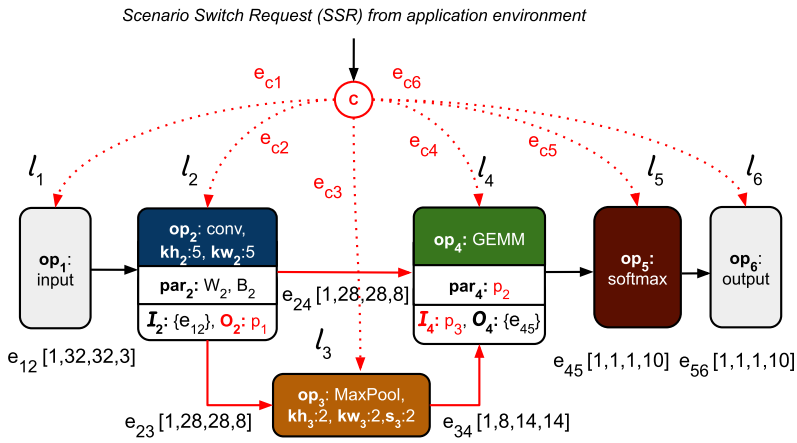


Figure 5.4: *SBRS MoC*

these scenarios. Every scenario in the SBRS MoC is a CNN. Figure 5.4 shows an example of the SBRS MoC, which models a CNN-based application associated with scenarios CNN^1 and CNN^2 shown Figure 5.3 and explained in Section 5.6. In this section, we use the example in Figure 5.4 to explain the SBRS MoC in detail. Formally, the SBRS MoC is defined as a scenarios supergraph, augmented with a control node c and a set of control edges E_c. The scenarios supergraph (see Section 5.7.1), captures all components (layers and edges) in every scenario of a CNN-based application. Therefore, it captures the functionality of every scenario, used by the application. To represent the functionality of a specific scenario, the SBRS MoC uses a sub-graph of the scenarios supergraph. The execution of a specific scenario (i.e., the use of a specific sub-graph of the scenarios supergraph) as well as run-time adaptive switching among the scenarios is determined by the control node c of the SBRS MoC (see Section 5.7.2). Finally, control edges E_c (see Section 5.7.3) specify the communication between the control node c and the scenarios supergraph. The details of the SBRS MoC deployment and inference at the Edge are provided in Section 5.7.4.

5.7.1 Scenarios supergraph

The scenarios supergraph of an SBRS MoC is a graph $SBRS(L, E)$ with a set of layers L which captures the functionality of every layer in every scenario of a CNN-based application, and a set of edges E which captures every data dependency in every scenario of the CNN-based application. Every layer l_i^s of every scenario CNN^s is captured by the functionally equivalent layer $l_n \in L$ of the scenarios supergraph, and every edge e_{ij}^s of every scenario

Table 5.1: *Capturing of scenarios' components (layers and edges) in the scenarios supergraph*

		layers						edges					
SBRS	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	all attri-butes of l_1	op_2,s_2, kw_2,kh_2, par_2, I_2	-	op_4,s_4, kw_4,kh_4, O_4	all attri-butes of l_5	all attri-butes of l_6	e_{12}	-	-	-	e_{45}	e_{56}

CNN^s is captured by the functionally equivalent edge $e_{nk} \in E$ of the scenarios supergraph. Table 5.1 shows how layers and edges of scenarios CNN^1 and CNN^2, shown in Figure 5.3 are captured in the scenarios supergraph of the SBRS MoC shown in Figure 5.4. For example, Column 9 in Table 5.1 shows that edge e^1_{12} of scenario CNN^1 and edge e^2_{12} of scenario CNN^2 are captured by edge e_{12} of the scenarios supergraph. We note that edge e_{12} is used by scenario CNN^1 and scenario CNN^2, i.e., edge e^1_{12} is *reused* among the application scenarios.

The reuse of components (layers and edges) of the scenarios supergraph is shown in Row 7 in Table 5.1. The reuse is introduced in the SBRS MoC because it allows for reduction of the CNN-based application memory cost and efficient utilization of the target edge platform memory by the CNN-based application. For example capturing of edge e^1_{12} of scenario CNN^1 and edge e^2_{12} of scenario CNN^2 by edge e_{12} of the SBRS MoC enables to reuse target edge platform memory allocated to data tensors $e^1_{12}.data$ and $e^2_{12}.data$, thereby reducing the application memory cost. Analogously, reuse of weights among and within application scenarios, enables to reuse the edge platform memory allocated to store these weights, thereby reducing the application memory cost. The reuse of a scenarios supergraph component can be full or partial. When a component is *fully reused*, all attributes of the component are reused. For example, layer l_1 of the scenarios supergraph shown in Column 3 is fully reused between scenarios CNN^1 and CNN^2, because all attributes of layer l_1 are reused between the scenarios[1]. When a component is *partially reused*, only some of its attributes are reused. For example, layer l_4 of the scenarios supergraph shown in Column 6 is partially reused between scenarios CNN^1 and CNN^2 because only attributes op_4, s_4, kw_4, kh_4, and O_4 of layer l_4 are reused among the scenarios.

The attributes that are not reused between the scenarios, are specified via run-time adaptive control parameters, introduced into the scenarios supergraph by the SBRS MoC to support partial components reuse. For example, as shown in Row 4 and Row 6, Column 6 in Table 5.1, 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 $\{\{W^1_3, B^1_3\}, \{W^2_4, B^2_4\}\}$ and control parameter p_3 takes values from the set $\{\{e_{24}\}, \{e_{34}\}\}$. When $p_2 = \{W^1_3, B^1_3\}$ and $p_3 = \{e_{24}\}$, supergraph layer l_4 is functionally equivalent to layer l^1_3 of scenario CNN^1. When $p_2 = \{W^2_4, B^2_4\}$ and $p_3 = \{e_{34}\}$, supergraph layer l_4 is functionally equivalent to layer l^2_4 of scenario CNN^2.

The capturing of scenarios' components (layers and edges) in the scenarios

[1] Attributes of a layer are defined in Table 2.1 in Section 2.1

supergraph (example of capturing is shown in Table 5.1 explained above) is determined at the application design time, and is stored in the control node c of the SBRS MoC during the application run-time.

5.7.2 Control node

The control node c of the SBRS MoC communicates with the application environment, and determines the execution of scenarios in the application supergraph as well as the switching between these scenarios.

Execution of scenario $CNN^s(L^s, E^s), s \in [1, S]$ captured by the SBRS MoC is defined as an execution sequence ϕ^s. The execution sequence is composed of computational steps, performed in a specific order, determined by the CNN topology and manner of execution as explained in Section 2.2. Every computational step $\phi_i^s \in \phi^s, i \in [1, |L^s|]$ involves execution of scenarios supergraph layer l_n, capturing layer l_i^s of scenario CNN^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 $\phi^1 = \{(l_1, \emptyset), (l_2, \{(p_1, \{e_{24}\})\}), (l_4, \{(p_2, \{W_3^1, B_3^1\}), (p_3, \{e_{24}\})\}), (l_5, \emptyset), (l_6, \emptyset)\}$. At step $\phi_1^1 = (l_1, \emptyset)$ of sequence ϕ^1 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.

The switching between the application scenarios is triggered by the application environment, communicating with the control node c. During the application run-time, control node c can receive a scenario switch request (SSR) from the application environment. Upon receiving the SSR, control node c changes old scenario CNN^o, executed by the node, to a new scenario CNN^n, more suitable for the application needs according to SSR. The switching from scenario CNN^o to scenario CNN^n is performed under the SBRS transition protocol, which will be explained in Section 5.9.

5.7.3 Control edges

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.

5.7.4 Deployment and inference

When a CNN-based application represented as the SBRS MoC is deployed on an edge platform, all the MoC SBRS scenarios supergraph components (layers and edges) as well as all associated parameters (weights and biases) are placed in the platform memory. During the application run-time, the control node c of the SBRS MoC uses part of these components and parameters to execute one of the application scenarios, captured by the SBRS MoC. The current scenario, also referred as an old scenario, is executed until the control node c receives a scenario switching request (SSR) from the application environment. Upon receiving the SSR, the control node c switches to a new scenario, more suitable for the application needs according to SSR. After the switching is finished, the scenarios supergraph continues to execute the new scenario, until a new SSR is received. If a new SSR is received during an ongoing scenarios switching, it is ignored.

5.8 SBRS MoC automated derivation

In this section, we propose an algorithm - see Algorithm 5 that automatically derives the SBRS MoC, explained in Section 5.7, from a set of S application scenarios $\{CNN^s\}, s \in [1, S]$, provided by the automated scenarios derivation component explained in Section 5.6. Algorithm 5 accepts as inputs: 1) the set of scenarios $\{CNN^s\}, s \in [1, S]$, where every scenario is a CNN, annotated with a specific manner of execution; 2) 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 5 provides an SBRS MoC, which captures application scenarios $\{CNN^s\}, s \in [1, S]$, and exploits the components reuse specified by set A. Figure 5.4 provides an example of the SBRS MoC, derived using Algorithm 5 for scenarios $\{CNN^1, CNN^2\}$ shown in Figure 5.3, and set $A = \{par, I, O\}$ of adaptive layer attributes.

Algorithm 5: SBRS MoC automated derivation

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

$L \leftarrow \emptyset$; $E \leftarrow \emptyset$; $\Pi \leftarrow \emptyset$; $L^{capt} \leftarrow \emptyset$; $E^{capt} \leftarrow \emptyset$;
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  for $l_i^s \in L^s$ **do**
    find $l_n \in L : eq(l_i^s, l_n, A)$//*Equation 5.1* $\wedge \nexists (l_n, l_j^q) \in L^{capt} : l_j^q$ and l_i^s are executed in parallel;
    if l_n *does not exist* **then**
      $n = |L|$;
      $l_n \leftarrow$ new layer $(type_i^s, op_i^s, -, -, -, -, \Theta_i^s, kh_i^s, kw_i^s, s_i^s, pad_i^s, par_i^s)$;
      $L \leftarrow L + l_n$;
    $L^{capt} \leftarrow L^{capt} + (l_n, l_i^s)$;
  for $e_{ij}^s \in E^s$ **do**
    find $l_k \in L : (l_k, l_i^s) \in L^{capt}$ and $l_n \in L : (l_n, l_j^s) \in L^{capt}$;
    if $\nexists e_{kn} \in E : eq(e_{kn}, e_{ij}^s, A)$ //*Equation 5.2* **then**
      $e_{kn} \leftarrow$ new edge (l_k, l_n);
      $E \leftarrow E + e_{kn}$;
    $E^{capt} \leftarrow E^{capt} + (e_{kn}, e_{ij}^s)$;
for $l_n \in L$ **do**
  if $\exists l_i^s \neq l_j^q : (l_n, l_i^s) \in L^{capt} \wedge (l_n, l_j^q) \in L^{capt}$ **then**
    for $attr \in l_n$ **do**
      for $l_i^s \in L^s : eq(l_i^s, l_n, A), s \in [1, S]$ **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;
          $\Pi \leftarrow \Pi + p$;
for $\text{CNN}^s(L^s, E^s), s \in [1, S]$ **do**
  $\phi^s = \emptyset$;
  for $i \in [1, |L^s|]$ **do**
    find $l_n \in L : (l_n, l_i^s) \in L^{capt}$;
    $P \leftarrow \emptyset$;
    for $attr \in l_n : attr.value = p_q \in \Pi$ **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**
          $e = e_{nk} \in E : (e_{nk}, e_{ij}^s) \in E^{capt}$;
          $value \leftarrow value + e$;
      else
        $value = sattr.value$;
      $P \leftarrow P + (p_q, value)$;
    $\phi^s \leftarrow \phi^s + (l_n, P)$;
$c \leftarrow$ new control node $(\{\phi^1, \phi^2, ..., \phi^S\}, L^{capt}, E^{capt})$;
$E_c \leftarrow \emptyset$;
for $l_n \in L$ **do**
  $e_{cn} \leftarrow$ new control edge (c, l_n);
  $E_c \leftarrow E_c + e_{cn}$;
return $SBRS(L, E, c, E_c)$

In Lines 1 to 23, Algorithm 5 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, an empty set of captured layers L^{capt}, and an empty set of captured edges E^{capt}. The latter two sets represent capturing of scenarios' components (layers and edges, respectively) in the scenarios supergraph.

In Lines 3 to 9, Algorithm 5 adds layers to the supergraph layers set L. For every layer l_i^s of every scenario CNN^s, Algorithm 5 first checks if set L contains a layer l_n that can be reused to capture layer l_i^s. The check is performed in Line 4 and consists of two parts. First, Algorithm 5 checks functional equivalence of layer l_n and layer l_i^s. This check is performed using Equation 5.1, which compares 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). Then, Algorithm 5 ensures than layer l_n does not capture layer l_j^q, executed in parallel with layer l_i^s. This check is performed using annotation of CNN^s which specifies a manner of execution of CNN^s. If condition in Line 4 is met, layer l_n is used to capture the functionality of layer l_i^s (Line 9 in Algorithm 5). Otherwise, a new layer l_n, capturing the functionality of layer l_i^s, is added to the scenarios supergraph (Lines 5 to 9 in Algorithm 5). To define a new layer, Algorithm 5 specifies a value for each attribute given in Table 2.1 and explained in Section 2.1. The values are listed in the order in which they appear in Table 2.1. If Algorithm 5 specifies a value as symbol "$-$", it means that the respective attribute takes the default value.

$$eq(l_i^s, l_n, A) = \begin{cases} true & \text{if } attr_n = attr_i^s, \forall attr \notin A \\ false & \text{otherwise} \end{cases} \tag{5.1}$$

In Lines 10 to 15, Algorithm 5 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 5 uses Equation 5.2.

$$eq(e_{ij}^s, e_{nk}, A) = \begin{cases} true & \text{if } eq(l_i^s, l_n, A) \wedge eq(l_j^s, l_k, A) \\ false & \text{otherwise} \end{cases} \tag{5.2}$$

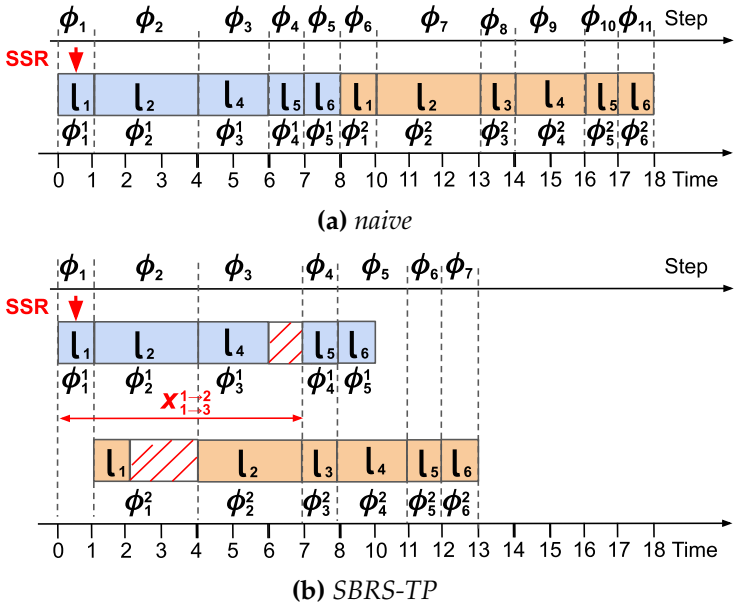
In Lines 16 to 23, Algorithm 5 introduces control parameters into the partially reused layers of the scenarios supergraph to capture those attributes that cannot be reused among the scenarios. For example, to capture attribute I_4

of scenarios supergraph layer l_4, shown in Figure 5.4, Algorithm 5 introduces control parameter p_3 into layer l_4 (as explained in Section 5.7).

In Lines 24 to 44, Algorithm 5 augments the scenarios supergraph, derived in Lines 2 to 23, with a control node c and a set of control edges E_c. In Lines 24 to 39, it generates execution sequence ϕ^s for every scenario CNN^s, captured by the scenarios supergraph. Every computational step $\phi_i^s, i \in [1, |L^s|]$ of the sequence ϕ^s is derived in Lines 26 to 38. In Line 27, Algorithm 5 determines layer l_n of scenarios supergraph, capturing functionality of layer l_i^s of scenario CNN^s. In Lines 28 to 38, Algorithm 5 derives set P of parameter-value pairs that specifies the values for every control parameter p_q associated with layer l_n. In Lines 29 to 38, Algorithm 5 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_i^s. In Line 30, Algorithm 5 finds attribute $sattr$ of layer l_i^s, corresponding to the attribute $attr$ of layer l_n. For example, if attribute $attr \in l_n$ is a set of parameters par of layer l_n, Algorithm 5 finds attribute $sattr \in l_i^s$, which is a set parameters par_i^s of layer l_i^s. 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 31 is met), the value for parameter p_q is specified in Lines 32 to 35 of Algorithm 5, 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 37 of Algorithm 5 as the value of attribute $sattr$ of layer l_i^s. In Line 40, Algorithm 5 creates a new control node c, which stores the execution sequence ϕ^s of every scenario as well as sets L^{capt} and E^{capt} that specify capturing of the components (layers and edges) of every scenario by the scenario supergraph. In Lines 41 to 44, Algorithm 5 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 45, Algorithm 5 returns the SBRS MoC, capturing the functionality of every scenario $CNN^s, s \in [1, S]$, associated with the CNN-based application.

5.9 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 5.7, 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



(a) *naive*

(b) *SBRS-TP*

Figure 5.5: *Switching from scenario* CNN^1 *to scenario* CNN^2

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 5.5(a), where the CNN-based application represented by the SBRS MoC from Figure 5.4 switches from scenario CNN^1 to scenario CNN^2 upon receiving a SSR at the first execution step of scenario CNN^1. The layers of scenario CNN^1 and scenario CNN^2 are executed in a sequential manner, explained at the end of Section 2.2. The upper axis in Figure 5.5(a) shows steps $\phi_i, i \in [1, 11]$, performed by the control node c during the scenarios switching. For example, Figure 5.5(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 5.5(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 5.5(a), shows the time needed to execute layer l_n. The response delay Δ of the naive switching, shown in Figure 5.5(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_6^2.

We note that this response delay can be reduced. Figure 5.5(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 5.7, 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 5.7, 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 5.5(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 5.5(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 5.5(a). Thus, the switching mechanism shown in Figure 5.5(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 5.5(b). The SBRS-TP is carried out in two phases: the analysis phase, and the scheduling phase. The analysis phase is performed during

Algorithm 6: 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}
```

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 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 5.5(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 6. Algorithm 6 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 6 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 6 for the scenario switching, shown in Figure 5.5(b), is the set $X^{1\rightarrow 2} = \{1, 2, 3, 4, 5, 6\}$. In Line 1, Algorithm 6 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 6. In Lines 2 to 10, Algorithm 6 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 6 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

Algorithm 7: SBRS-TP scheduling phase

Input: $\phi^o, \phi^n, X^{o \to n}$
1 $q = 1; i = 1; j = step^o_{SSR}$;
2 wait until step ϕ^o_j is finished; $j = j + 1; q = q + 1$;
3 **while** $j \leq |L^o|$ **do**
4 start ϕ^o_j; $j = j + 1$;
5 **if** $q \geq x^{o \to n}_{1 \to i} - step^o_{SSR} + 2$ **then**
6 start ϕ^n_i; $i = ((i + 1) \mod |L^n|)$;
7 wait until started scenarios' steps are finished; $q = q + 1$;
8 **while** $i \leq |L^n|$ **do**
9 start ϕ^n_i;
10 wait until ϕ^n_i finishes; $i = i + 1; q = q + 1$;

met), it delays the execution of step ϕ^n_i until step ϕ^o_j is finished. In Line 9, Algorithm 6 adds the delay of step ϕ^n_i, stored in variable x, to the set $X^{o \to n}$. In Line 10, Algorithm 6 increases the delay by one step, thereby defining an initial delay for the next step ϕ^n_{i+1} of the new scenario CNN^n. Finally, in Line 11, Algorithm 6 returns the set $X^{o \to n}$. The set $X^{o \to n}$ derived using Algorithm 6 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 7. 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 $X^{o \to n}$ derived by Algorithm 6 for scenarios CNN^o and CNN^n at the SBRS-TP analysis phase. In Line 1, Algorithm 7 defines variables i, j, and q, representing indexes of the current step ϕ^n_i of the new scenario CNN^n, current step ϕ^o_j 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 5.5(b), $step^o_{SSR} = 1$ because SSR arrives at step ϕ^1_1 of the old scenario CNN^1. In Line 2, Algorithm 7 performs the first step ϕ_1 of the scenarios switching. During this step, Algorithm 7 waits until step ϕ^o_j, during which the SSR

arrived, finishes. In Lines 3 to 7, Algorithm 7 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 $x_{1\rightarrow i}^{o\rightarrow n}$ between steps ϕ_1^o and ϕ_i^n is observed (the condition in Line 5 is met). In Line 7, Algorithm 7 waits until the steps of scenarios CNN^o and CNN^n, started in Lines 4 to 6, finish. In Lines 8 to 10, Algorithm 7 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 7 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.

5.10 Experimental Study

To evaluate our novel SBRS methodology, we perform an experiment, where we apply our methodology to 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 three real-world CNN-based applications from different domains. The fourth step compares our methodology to the most relevant existing work.

In Step 1 (Section 5.10.1), we derive scenarios for three real-world CNN-based applications with scenarios. We illustrate the diversity among the selected scenarios and compare the use of multiple scenarios (CNNs), enabled by our methodology, to use of a single CNN, adopted by the state-of-the-art design flow, introduced in Section 1.3 and shown in Figure 1.3. By performing this experiment, we evaluate the effectiveness of run-time adaptivity, offered by our methodology.

In Step 2 (Section 5.10.2), we use Algorithm 5, proposed in Section 5.7.1, 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 2.1. By performing this experiment, we evaluate the efficiency of the memory reuse, exploited by the SBRS MoC, proposed in Section 5.7.

In Step 3 (Section 5.10.3), 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 our SBRS-TP 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 5.9.

Finally, in Step 4 (Section 5.10.4), we compare our SBRS methodology with the most relevant existing work. As explained in Section 5.3 and demonstrated in Section 5.4, none of the currently existing works 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 5.3 and Section 5.4, the MSDNet adaptive CNN work [39] is the most relevant to our methodology. Similarly to our methodology and unlike other reviewed existing work, the methodology in [39] 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 [39] 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 [39] and our SBRS methodology can be compared via 1) run-time adaptive trade-offs between application accuracy and resources utilization; 2) memory efficiency. In Section 5.10.4, we perform such comparison, using the image recognition CIFAR-10 dataset [51].

To perform the measurements, required for Step 1 to Step 4, we implement the executable CNN-based applications, using the TensorRT Deep Learning framework and operators library [72], providing state-of-the-art performance of deep learning inference on the NVIDIA Jetson TX2 embedded device [71], 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.

5.10.1 Automated scenarios derivation

In this section, we derive scenarios for three CNN-based applications from two different domains, namely Human Activity Recognition (HAR) and image classification. We used the PAMAP2 [79] dataset for HAR and the Pascal VOC [27]

Table 5.2: *CNN-based applications*

App	task	baseline CNN	dataset	app. requirements sets
Pascal VOC	Image recongition	ResNet [36]	Pascal VOC [27]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.7, 0.0, 0.3, 0.0)$ $r_3 = (0.6, 0.1, 0.0, 0.3)$ $r_4 = (0.5, 0.5, 0.0, 0.0)$ $r_5 = (0.1, 0.1, 0.4, 0.4)$
PAMAP2	Human activity monitoring	PAMAP (CNN-2) [69]	PAMAP2 [79]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.2, 0.4, 0.0, 0.4)$ $r_3 = (0.5, 0.0, 0.0, 0.5)$ $r_4 = (0.5, 0.5, 0.0, 0.0)$
CIFAR-10	Image recognition	ResNet [36]	CIFAR-10 [51]	$r_1 = (1.0, 0.0, 0.0, 0.0)$ $r_2 = (0.25, 0.25, 0.25, 0.25)$ $r_3 = (0.5, 0.25, 0.0, 0.25)$ $r_4 = (0.5, 0.0, 0.0, 0.5)$

and CIFAR-10 [51] 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 down-sampled 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 5.2. Column 1 lists the 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 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 [36] to the small and shallow PAMAP topology) and diverse datasets (from the large Pascal VOC [27] dataset to the small PAMAP2 [79] and CIFAR-10 [51] 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

Table 5.3: *Algorithm parameters for platform-aware NAS [82]*

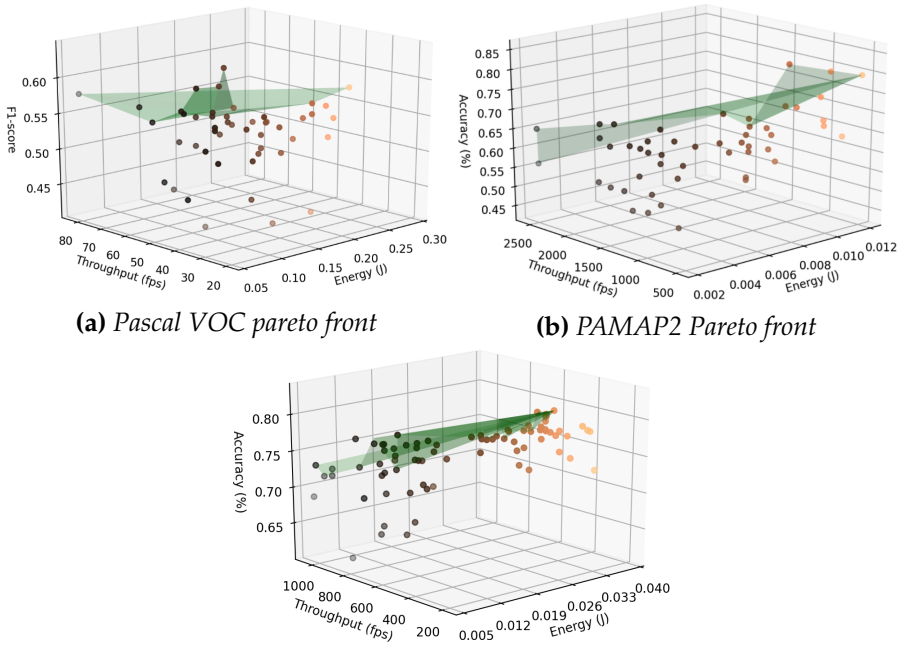
Parameter		VOC	PAMAP2	CIFAR10
Mutation change rate	ϱ_m	0.10	0.12	0.12
Mutation probability	P_m	0.3	0.3	0.3
Initial Crossover probability	$P_r(0)$	0.3	0.4	0.3
Population size	N_p	60	50	100
No of iterations	N_g	30	60	120
Population replacement rate	Ω	0.02	0.03	0.02
Training Parameters	τ_{params}			
Training 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

our methodology.

To derive the scenarios for each application, described in Table 5.2, we used the multi-objective platform-aware NAS methodology proposed in [82] and the scenarios selection based on the ranking dominance concept introduced in [52]. In addition to the baseline CNN and the dataset, specified in Table 5.2, the platform-aware NAS methodology in [82] accepts as input a set of NAS hyper-parameters. The NAS hyper-parameters used in our experiments are summarized in Table 5.3. For the explanation of the NAS hyper-parameters, we respectfully refer the reader to work [82].

The methodology in [82] performs automated search for optimal CNNs, which arrives at a set of CNNs pareto-optimal in terms of accuracy, memory, throughput and energy characteristic. Every CNN in the set is executed in a sequential manner, explained in Section 2.2. The pareto-fronts obtained for Pascal VOC, PAMAP2 and CIFAR-10 applications listed in Table 5.2, are shown in Figure 5.6(a), Figure 5.6(b) and Figure 5.6(c), respectively. The pareto-fronts are shown in three-dimensional (accuracy, throughput and energy cost) space to allow for a comprehensible visualization, while the actual design points exist in four-dimensional (accuracy, throughput, energy cost, and memory cost) space.

The scenarios selected from the pareto-fronts shown in Figure 5.6 for the three multi-scenario applications given in Table 5.2, are presented in Table 5.4, where Column 1 shows the CNN-based applications; Column 2 shows the requirements sets, depicting scenarios, associated with every application; Column 3 shows the scenarios, derived for the requirements sets given in Column 2; Columns 4 to 7 show the accuracy and platform-aware characteristics of every scenario given in Column 3. For the PAMAP2 and CIFAR-10 applica-



(a) *Pascal VOC pareto front*

(b) *PAMAP2 Pareto front*

(c) *CIFAR-10 Pareto front*

Figure 5.6: *Pareto-fronts based on 3 evaluation parameters, namely, accuracy (F1-score for Pascal VOC), throughput and energy*

Table 5.4: *Scenarios derived from pareto-fronts shown in Figure 5.6 for three applications shown in Table 5.2*

App.	app. req. set	scenario	Accuracy (PR-AUC or %)	Throughput (fps)	Memory (MB)	Energy (J)
Pascal VOC	r_1	CNN^1	77.78	15.41	292.61	0.384
	r_2	CNN^2	76.28	21.78	210.69	0.281
	r_3	CNN^3	77.69	20.26	242.72	0.291
	r_4	CNN^4	73.99	59.27	155.48	0.101
	r_5	CNN^5	72.85	75.07	130.21	0.078
PAMAP2	r_1	CNN^1	94.17	510.20	10.02	0.0083
	r_2	CNN^2	91.34	1333.33	4.30	0.0033
	r_3	CNN^3	92.56	970.87	4.86	0.0037
	r_4	CNN^4	92.93	1052.63	4.11	0.0039
CIFAR-10	r_1	CNN^1	94.86	231.80	52.87	0.0242
	r_2	CNN^2	92.84	754.15	13.07	0.0055
	r_3	CNN^3	93.46	538.79	18.30	0.0081
	r_4	CNN^4	94.46	403.71	28.07	0.0121

tions, the accuracy is estimated using the cross-validation technique [78] and measured in percent. For Pascal-VOC, accuracy was estimated as the PR-AUC (Area under precision-recall curve) [89]. Columns 5 to 7 show the scenario throughput (in frames per second), memory (in MegaBytes) and Energy (in Joules), respectively.

Columns in 4 to 7 in Table 5.4 clearly demonstrate that scenarios (CNNs) obtained for different application requirements have vastly different characteristics. If Pascal VOC, PAMAP2, and CIFAR-10 CNN-based applications would use only one of the selected scenarios, as proposed in the state-of-the-art design flow, shown in Figure 1.3 and explained in Section 1.3, the applications' needs would not be optimally served.

For example, let us assume that the Pascal VOC application, shown in Row 2 in Table 5.2 and associated with scenarios CNN^1, CNN^2, CNN^3, and CNN^4, shown in Row 2 in Table 5.4: 1) only uses scenario CNN^1 when is designed using the state-of-the-art design flow; 2) adaptively switches between scenarios CNN^1, CNN^2, CNN^3 and CNN^4, when designed using our proposed methodology. Under requirements set r_4, specifying that high CNN throughput is as important as high CNN accuracy, the application would use CNN^1 when is designed using the state-of-the-art design flow, and CNN^4 when is designed using our proposed methodology. Compared to CNN^1, CNN^4 demonstrates 3.8 times higher throughput and only 4% lower accuracy. Thus, the Pascal VOC application would better serve the application needs, specified in the requirements set r_4.

The example above shows that our methodology ensures more efficient serving of CNN-based applications affected by the environment at run-time when compared to the the state-of-the-art design flow shown in Figure 1.3 and explained in Section 1.3.

5.10.2 SBRS MoC memory reuse efficiency

In this experiment, we measure and compare the memory cost of every CNN-based application, presented in Table 5.2 in Section 5.10.1, 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 5.5.

In Table 5.5, Column 1 lists the CNN-based applications with scenarios, explained in Section 5.10.1. Column 2 shows the sets of adaptive layer attributes A, used by Algorithm 5 to generate the SBRS MoCs for the CNN-based appli-

Table 5.5: *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

cations. 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 5.5, 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 5.5, 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 5.5 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 5.5 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 5.7.1), 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 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

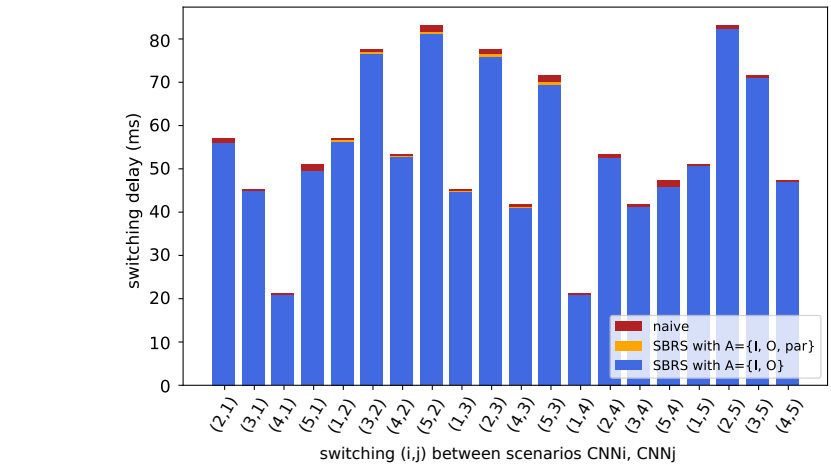
PAMAP2. Thus, in CIFAR-10, the SBRS MoC can reuse less components than in Pascal VOC, but more components than in PAMAP2.

5.10.3 SBRS-TP efficiency

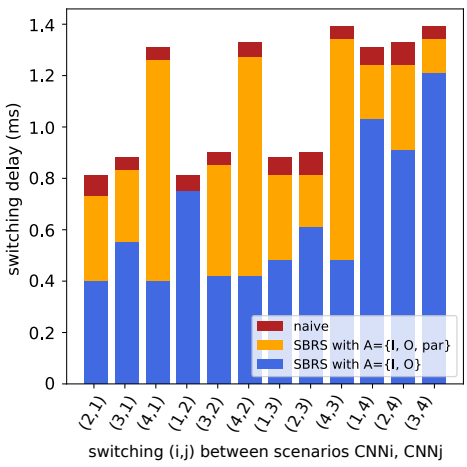
In this experiment, for every CNN-based application, explained in Section 5.10.1, and represented as two functionally equivalent SBRS MoCs with sets of adaptive attributes $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, we measure and compare the application responsiveness during the scenarios 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 5.7, 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 5.7(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 [2]; 2) 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 the switching is. For example, switching (2, 1), shown in Figure 5.7(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 the 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 5.7(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 5.7: 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

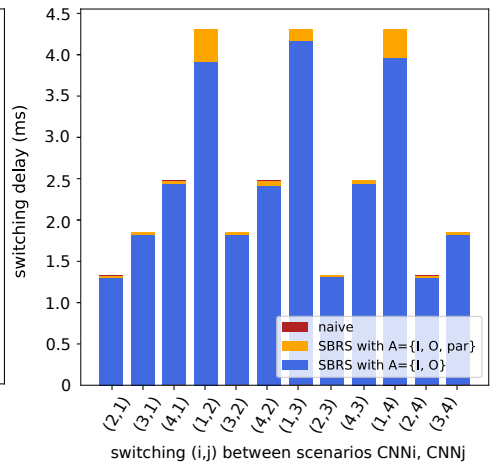
[2] One bar is sufficient to show the delay of the naive switching for SBRS MoCs with $A = \{I, O\}$ and $A = \{I, O, par\}$, respectively, because, as explained in Section 5.9, the naive switching is not affected by the application components reuse, determined by the set A



(a) *Pascal VOC*



(b) *PAMAP2*



(c) *CIFAR-10*

Figure 5.7: *SBRS-TP efficiency evaluation*

is performed under the SBRS-TP, the 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 5.7). As explained in Section 5.9, 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 5.10.2, Pascal VOC exploits more components reuse than PAMAP2 or CIFAR-10.

5.10.4 Comparative study

In this section, we compare our SBRS methodology to the most relevant related work called the MSDNet adaptive CNN methodology [39]. As explained in Section 5.10, the MSDNet methodology and our SBRS methodology can be compared via: 1) the run-time adaptive trade-offs between the application accuracy and resources utilization; 2) the memory efficiency.

In this section, we perform such a comparison 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 comparison is performed in three steps. In Step 1, we construct the MSDNet CNN and the SBRS MoC for the example application. In Step 2, we compare the accuracy-throughput trade-off offered by the MSDNet methodology and our SBRS methodology, using the applications obtained at Step 1. Finally, in Step 3, we compare the memory efficiency of the MSDNet methodology and our SBRS methodology, using the applications obtained at Step 1.

The MSDNet CNN is constructed according to the design and training parameters specified for the CIFAR-10 dataset in the original MSDNet work [39]. 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 [39]. 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 [39] 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 5.5. As input, our methodology accepts a ResNet-18 [36] baseline CNN 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 impor-

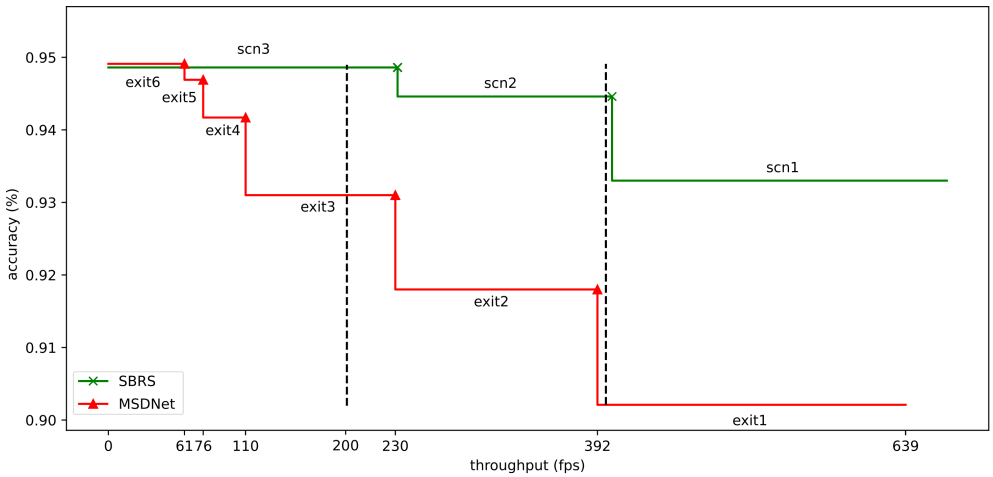


Figure 5.8: *Comparison between SBRS MoC and MSDNet CNN [39], performing classification on the CIFAR-10 dataset with throughput-driven adaptive mechanism*

tant 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 the 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 5.8. The horizontal axis shows the throughput (in fps). The vertical axis shows the accuracy (in %). The two step-wise curves in Figure 5.8 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 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 5.8 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 5.8, 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 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 unlike the MSDNet methodology, our methodology reuses memory allocated to store intermediate computational results within every CNN as well as among different CNNs. 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.

5.11 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: 1) Enables to adapt a CNN-based application to changes in the application environment during run-time and therefore ensures that the application needs are served at every moment in time; 2) Achieves a high (up to 78%) degree of platform memory reuse for CNN-based applications that execute CNNs with large amounts of similar components; 3) Enables for 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.