Deep Learning for Effective and Efficient Reduction of Large Adaptation Spaces in Self-adaptive Systems

DANNY WEYNS, Katholieke Universiteit Leuven, Belgium, Linnaeus University Sweden
OMID GHEIBI and FEDERICO QUIN, Katholieke Universiteit Leuven, Belgium
JEROEN VAN DER DONCKT, Ghent University (imec), Belgium

Many software systems today face uncertain operating conditions, such as sudden changes in the availability of resources or unexpected user behavior. Without proper mitigation these uncertainties can jeopardize the system goals. Self-adaptation is a common approach to tackle such uncertainties. When the system goals may be compromised, the self-adaptive system has to select the best adaptation option to reconfigure by analyzing the possible adaptation options, i.e., the adaptation space. Yet, analyzing large adaptation spaces using rigorous methods can be resource- and time-consuming, or even be infeasible. One approach to tackle this problem is by using online machine learning to reduce adaptation spaces. However, existing approaches require domain expertise to perform feature engineering to define the learner and support online adaptation space reduction only for specific goals. To tackle these limitations, we present "Deep Learning for Adaptation Space Reduction Plus"—DLASeR**+** for short. DLASeR**+** offers an extendable learning framework for online adaptation space reduction that does not require feature engineering, while supporting three common types of adaptation goals: threshold, optimization, and set-point goals. We evaluate DLASeR**+** on two instances of an Internet-of-Things application with increasing sizes of adaptation spaces for different combinations of adaptation goals. We compare DLASeR**+** with a baseline that applies exhaustive analysis and two state-of-the-art approaches for adaptation space reduction that rely on learning. Results show that DLASeR**+** is effective with a negligible effect on the realization of the adaptation goals compared to an exhaustive analysis approach and supports three common types of adaptation goals beyond the state-of-the-art approaches.

CCS Concepts: • **Software and its engineering → Software design engineering**; • **Theory of computation → Online learning algorithms;**

Additional Key Words and Phrases: Self-adaptation, adaptation space reduction, analysis, planning, deep learning, threshold goals, optimization goal, set-point goal, Internet-of-Things

ACM Reference format:
Danny Weyns, Omid Gheibi, Federico Quin, and Jeroen Van Der Donckt. 2022. Deep Learning for Effective and Efficient Reduction of Large Adaptation Spaces in Self-adaptive Systems. *ACM Trans. Auton. Adapt. Syst.* 17, 1-2, Article 1 (July 2022), 42 pages.
https://doi.org/10.1145/3530192

Authors' addresses: D. Weyns, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium, and Linnaeus University, 35252 Vaxjo, Sweden; email: danny.weyns@kuleuven.be; O. Gheibi, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium; email: omid.gheibi@kuleuven.be; F. Quin, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium; email: federico.quin@kuleuven.be; J. Van Der Donckt, Ghent University (imec), Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium; email: jeroen.vanderdonckt@ugent.be.

1556-4665/2022/07-ART1
https://doi.org/10.1145/3530192

1 INTRODUCTION

Many software systems today face changing and uncertain operating conditions. For such systems, employing a stationary approach that does not adapt to changes may jeopardize the quality goals of the system. Consider for instance a web-based system that runs a fixed number of servers based on an average load. This configuration will result in a waste of resources when the load is very low, but the number of servers may be insufficient to handle the peak demand [43].

Self-adaptation is one prominent approach to tackle such problems [14, 71]. A self-adaptive system reasons about itself and its environment, based on observations, to determine whether adaptation is required. In the case that adaptation is required, the system adapts itself to meet its adaptation goals, or gracefully degrade if the goals may temporarily not be achievable. Self-adaptation has been applied in a wide range of application domains, ranging from service-based systems to cyber-physical systems, Internet-of-Things, the Cloud, and robotics [4, 10, 20, 38, 52]. In the example of the web-based system above, enhancing the system with self-adaptation enables it to increase and decrease the number of servers dynamically based on the monitored load. This results in higher user satisfaction as well as improved economical and ecological use of resources.

In this research, we apply architecture-based adaptation [25, 45, 80] that adds an external feedback loop to the system. The feedback loop uses up-to-date architectural models as first-class citizens to reason about changes. The feedback loop forms a managing system on top of managed software system and is structured according to the **MAPE-K** reference model, short for **Monitor - Analyzer - Planner - Executor - Knowledge** [40]. MAPE-K divides adaptation in four principal functions [40, 79, 84]. The monitor monitors the system and its environment. The analyzer determines whether adaptation is required or not and if so, then it analyzes the *adaptation options* for adapting the system. An adaptation option is a configuration of the system that can be reached from the current configuration by changing elements of the system through *adaptation actions*. Adaptation actions can range from adjusting a parameter of the system up to an architectural re-configuration of the system. We use the term *adaptation space* as the set of all the possible adaptation options at some point in time, i.e., all the possible configurations that can be reached from the current configuration of the system by applying a set of adaptation actions to the system. The size of the adaptation space (i.e., the number of adaptation options) may be constant over time, or it may change dynamically. The planner then selects the best adaptation option according to the adaptation goals and composes a plan for adapting the managed system. Finally, the executor applies the adaptation actions of the plan on the managed system. The four MAPE functions share common **knowledge (K)**, e.g., models of the system, the adaptation goals, the set of adaptation options, an adaptation plan, among others.

This article focuses on the analysis of the adaptation options of the adaptation space, which is a task of the analyzer, and selecting the best option based on the analysis results and the adaptation goals, which is a task of the planner. Both tasks are essential for the decision-making in self-adaptive systems. During the execution of these tasks the feedback loop estimates a set of quality properties for each adaptation option of the adaptation space, each quality property corresponding to an adaptation goal. We consider threshold goals that require a system parameter to stay above/below a given threshold, optimization goals that require to minimize or maximize a system parameter, and set-point goals that require a system parameter to stay as close as possible to a given value.

Selecting an adaptation option from a large adaptation space is often computationally expensive [14, 18, 73]. A common technique used to find the best adaptation option is runtime verification of formal runtime models that represent the system and its environment for one or more quality properties. These quality models have parameters that can be instantiated for a particular adaptation option and the actual values of the uncertainties. A classic approach to estimate the

qualities of the adaptation option is runtime quantitative verification; see for example References [8, 50, 76]. It is important to note that the adaptation space exhibits dynamic behavior that is difficult to predict upfront. On the one hand, the estimated quality properties of the adaptation options vary over time, as the uncertainties the system is exposed to change over time. On the other hand, the system configuration itself dynamically changes as the system is adapted over time.

Different techniques have been studied to find the best adaptation option in a large adaptation space. One particular approach to deal with the problem is adaptation space reduction that aims at retrieving a subset of relevant adaptation options from an adaptation space that are then considered for analysis. An adaptation option is labeled relevant when it is predicted to satisfy the adaptation goals. Techniques have been applied in this approach including search-based techniques [53] and feature-based techniques [49]. Recently, different machine learning techniques have been investigated to reduce the adaptation space at runtime; see for instance References [22, 37, 58]. Among the applied learning techniques are decision trees, classification, and regression [29]. However, most of these techniques rely on domain expertise to perform feature engineering to define the learner, which may hamper the applicability in practice. Further, most existing approaches are limited to threshold goals and optimization goals. In this article, we tackle the following research question:

> *How to reduce large adaptation spaces and rank adaptation options effectively and efficiently at runtime for self-adaptive systems with threshold, optimization, and set-point goals?*

With *effectively*, we mean the solution should ensure that: (1) the reduced adaptation space is significantly smaller, (2) the relevant adaptation options should be covered well, that is, the adaptation options that satisfy (most of) the adaptation goals should be included, (3) the effect of the state space reduction on the realization of the adaptation goals is negligible. With *efficiently*, we mean the solution should ensure that: (4) the learning time is small compared to the time needed for analysis, (5) there is no notable effect on (1), (2), (3), and (4) for larger sizes of the adaptation space.

To answer the research question, we propose a novel approach for adaptation space reduction called **Deep Learning for Adaptation Space Reduction Plus (DLASeR+)**. DLASeR**+** leverages on deep learning that relies on deep artificial neural networks, i.e., neural networks with many layers. DLASeR**+** offers an extendable framework that performs effective and efficient online adaptation space reduction for threshold, optimization, and set-point goals. While DLASeR**+** can handle an arbitrary mix of goals, the concrete architecture we present in this article is tailored to a set of threshold and set-point goals and one optimization goal. DLASeR**+**'s learning pipeline consists of an offline and online part. During the offline part, the learning model is selected and configured using training data. During the online part, the running system uses the model to reduce adaptation spaces and exploits newly obtained data from analysis to continue the training and to update the learning model enabling the system to deal with changing operating conditions.

We studied deep learning for four important reasons. First, classic learning techniques usually require some form of human input for feature engineering, whereas deep learning can handle raw data without the need for feature engineering. Second, besides some exceptions, classic machine learning models are usually linear in nature, whereas deep learning can work with non-linear models. Third, learned features and even entire models can be reused across similar tasks. This type of transfer learning is a consequence of representation learning, which is the basic core concept that drives deep learning. We exploit representation learning in the DLASeR**+** neural network architecture. Fourth, given the success of deep learning in various other domains, e.g., computer vision [65] and natural language processing [60], we were curious to explore how well deep learning could perform for an important runtime problem in self-adaptive systems.

In initial work, we explored the use of deep learning to reduce adaptation spaces for threshold and optimization goals [70].[1] The goal of that initial work was to investigate the usefulness of deep learning for adaptation space reduction. Compared to that exploratory work, DLASeR**+** supports besides threshold and optimization goals also set-point goals. Whereas the initial approach of Reference [70] used a distinct model per goal, DLASeR**+** works with an integrated learning architecture that uses a single model, where layers are shared and reused across different adaptation goals. Furthermore, DLASeR**+** requires a single grid search process and a single prediction step in each adaptation cycle, whereas the initial approach required grid search and prediction for each goal in each cycle.

We evaluate DLASeR**+** on two instances of DeltaIoT, an artifact for evaluating self-adaptive systems [35]. The **Internet-of-Things (IoT)** is a challenging domain to apply self-adaptation, given its complexity and high degrees of uncertainties [81]. The two instances of DeltaIoT differ in the size of their adaptation space, enabling us to evaluate the different aspects of effectiveness and efficiency. To that end, we define appropriate metrics to evaluate DLASeR**+** and compare it with a baseline that applies exhaustive analysis, and two existing learning-based approaches for adaptation space reduction: ML4EAS [58], which uses classic learning techniques, and the initial DLASeR [70].

The contributions of this article are: (1) DLASeR**+**, a novel modular approach for adaptation space reduction in self-adaptive systems that is able to handle threshold, optimization, and set-point goals, and (2) a thorough evaluation of the effectiveness and efficiency of the approach in the domain of IoT, including a comparison with a baseline and two state-of-the-art approaches.

Given the specific domain we use in the evaluation with relatively limited sizes of adaptation spaces, we want to emphasize that additional validation is required to generalize the findings.

The remainder of this article is structured as follows: In Section 2, we provide relevant background: We introduce DeltaIoT, we present a high-level architecture of self-adaptation with adaptation space reduction, we zoom in on the different types of adaptation goals, the adaptation space, and we introduce the essential concepts of deep learning. Section 3 gives a high-level overview of the research methodology. In Section 4, we introduce a set of metrics that we use to measure the effectiveness and efficiency of DLASeR**+** and compare the approach with alternative approaches. Sections 5 and 6 present the core technical contribution of this article: the architecture and learning pipeline of DLASeR**+** architecture, respectively. In Section 7, we use the metrics to evaluate DLASeR**+** for two instances of DeltaIoT. Section 8 positions DLASeR**+** in the landscape of other related work. Finally, we draw conclusions and look at future work in Section 9.

2 BACKGROUND

This section introduces the necessary background for this article. We start with introducing DeltaIoT. Then, we explain the basic architecture of a self-adaptive system that integrates a verifier for runtime analysis and a learning module for online adaptation space reduction. Next, we introduce the different types of adaptation goals that are supported by DLASeR**+**. Then, we elaborate on the concept of adaptation space. Finally, we introduce the relevant concepts of deep learning.

2.1 DeltaIoT

DeltaIoT is a reference IoT application that has been deployed at the Campus Computer Science of KU Leuven [35]. DeltaIoT has been developed to support research on self-adaptation, i.e., evaluate new self-adaptive approaches, e.g., to evaluate tradeoffs between non-functional requirements in self-adaptive systems [21] or perform cost-benefit-analysis in runtime decision-making for

[1]This initial version was denoted DLASeR; the **+** emphasizes that DLASeR**+** significantly extends DLASeR.

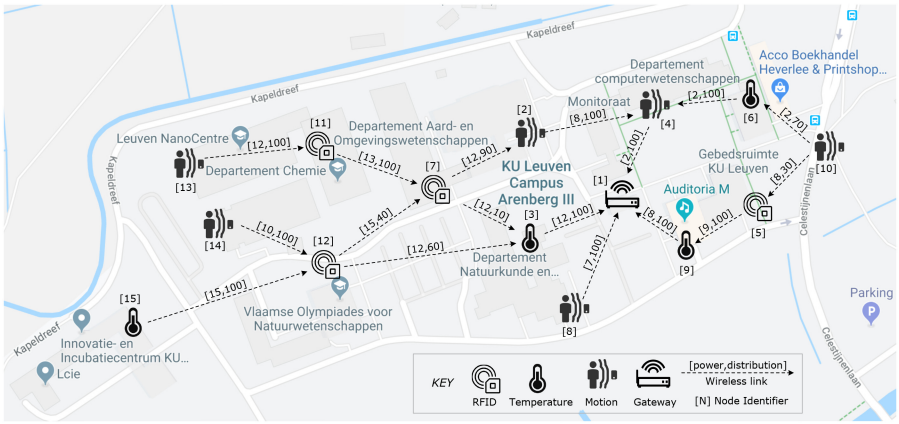


Fig. 1. DeltaIoT deployment at the KU Leuven campus (borrowed from Reference [69]). The gateway is marked by the blue icon in the center. The data collected by the sensors is sent over multiple wireless links to this gateway.

self-adaptation [69]. Next to the real physical setup deployed by VersaSense,[2] DeltaIoT also offers a simulator for offline experimentation. We use DeltaIoT as evaluation case, but also as running example to illustrate the different parts that follow in this section.

Figure 1 shows the physical setup of DeltaIoT. RFID sensors are used to provide access control to labs, passive infrared sensors monitor the occupancy of several buildings, and heat sensors are employed to sense the temperature. The data of these sensors is relayed to the gateway by means of wireless multi-hop communication. Each sensor is plugged into a battery-powered mote, i.e., a networked tiny embedded computer. The motes take care of the routing of sensor data to the gateway. The communication in DeltaIoT is time-synchronized and organized in cycles with a fixed number of slots. Neighboring motes are assigned such slots during which they can exchange packets. Motes collect data (locally generated or received from other motes) in a buffer. When a mote gets a turn to communicate with another mote, it forwards the packets to the other mote. Packets that cannot be sent remain in the buffer until the mote is assigned a next slot.

DeltaIoT has three main quality requirements: packet loss, latency, and energy consumption. For these qualities, we define corresponding adaptation goals, e.g., average latency of packages delivery should not exceed a predefined percentage and energy consumption of the network should be minimized. Ensuring such goals is challenging, since the IoT network is subject to various types of uncertainties. Two main uncertainties are interference along network links caused by external phenomena and changing load in the network, that is, motes only send packets when there is useful data, which may be difficult to predict. The IoT network can be adapted using two parameters: power setting and link distribution. The first parameter refers to the setting of the transmission power of each mote. The options are discretized as an integer in the range of 1 to 15. Increasing the transmission power will reduce packet loss but increase energy consumption. The second parameter refers to the way data packets are relayed from the motes to the gateways. Observe in Figure 1 that several motes have multiple links over which data can be sent. We refer to the distribution of packets sent by a mote to its parents as the link distribution. If a mote has only one parent, then it is obvious that it relays 100% of its packets through that single parent. But when a mote has multiple parents, the distribution of packets over the different links to the

[2]VersaSense website: www.versasense.com.

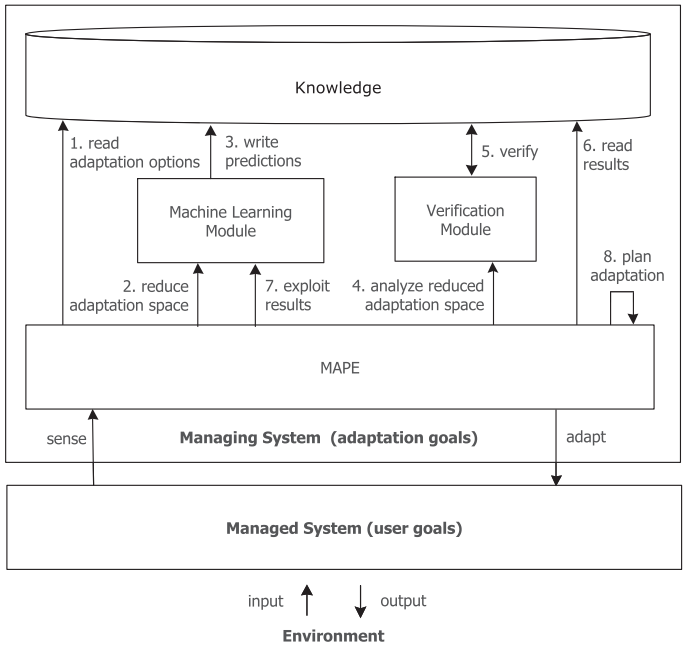


Fig. 2. A self-adaptive system that uses learning for adaptation space reduction.

parents can be selected and changed. Note that the sum of the distributions should remain 100% (to optimize energy consumption). By changing the link distribution, paths with more interference can be avoided. However, this may cause delays at the buffers of the motes along these paths.

For practical reasons, we use the DeltaIoT simulator for the evaluation of DLASeR**+**, since extensive experimentation on a physical IoT deployment is particularly time-consuming. The DeltaIoT simulator offers a realistic alternative for the physical network where parameters of the simulator are based on field experiments. We consider two instances of DeltaIoT. The first one, referred to as *DeltaIoTv1*, consists of 15 motes (shown in Figure 1); the second instance, referred to as *DeltaIoTv2*, consists of 37 motes. The larger IoT network is more challenging in terms of the number of configurations that are available to adapt the system representing the adaptation space. In particular, the adaptation space of DeltaIoTv1 contains 216 possible adaptation options, while DeltaIoTv2 has 4,096 adaptation options. These numbers are determined by the parameters of the IoT network that can be used for adapting the system: power setting and link distribution.[3] Hence, for both versions, the number of adaptation options is constant. However, as will explain in Section 2.4, the properties of the adaptation options change dynamically with changing conditions.

2.2 Basic Architecture Self-adaptive System with Adaptation Space Reduction

Figure 2 shows the basic architecture of a self-adaptive system that uses learning for adaptation space reduction. As explained in the introduction, in this research, we apply architecture-based

[3]Technically, we apply the following approach to determine the adaptation options for the IoT settings: First, we determine the required power settings for each mote along the links to its children such that the signal-to-noise ratio is at least zero. These settings are determined based on the actual values of signal-to-noise along the links. The settings are then fixed for all adaptation options. The number of adaptation options is then determined by the combinations of all possible settings of link distributions in the network. This number is 216 for DeltaIoTv1 and 4,096 for DeltaIoTv2.

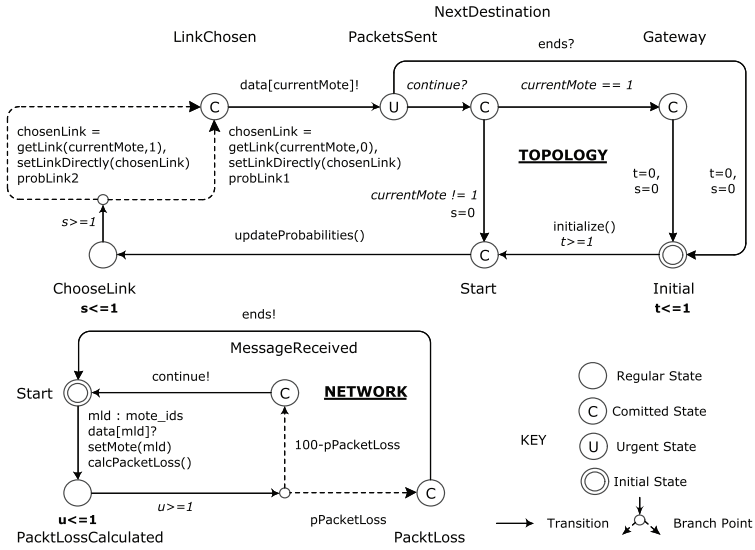


Fig. 3. Runtime quality model for packet loss.

adaptation with a MAPE-K feedback loop. The figure highlights the main elements of architecture and high-level flow of interactions between the elements to realize adaptation space reduction. The managing system is responsible for adapting the managed system to achieve a set of adaptation goals. The *managing system* manages the managed system to achieve a set of *adaptation goals*. Central to the managed system are the *MAPE* elements that share *knowledge* and realize a feedback loop. The feedback loop *senses* the managed system and *adapts* it to achieve the adaptation goals.

When the MAPE feedback loop detects that the adaptation goals are violated or may no longer be achievable, it reads the adaptation options from the knowledge (1). The feedback loop then instructs the *machine learning module* to reduce the adaptation space (2). The machine learning module will use its learning model to make predictions about the expected quality properties for the different adaptation options (3). Based on these predictions, the adaptation space is reduced to the most relevant adaptation options. Next, the MAPE feedback loop instructs the *verifier module* to analyze the adaptation options of the reduced adaptation space (4). When the verifier completes the verification (5), the MAPE feedback loop reads the results (6). It then forwards the results to the machine learning module that exploits the results to continue the training of its learning model (7). Finally, the MAPE feedback loop generates a plan (8) that is used to adapt the managed system.

Analysis adaptation option. We illustrate how an adaptation option is analyzed for DeltaIoT. In particular, we explain how packet loss is estimated by the analyzer of the MAPE loop using a statistical model checker (we apply this approach in the evaluation in Section 7). Figure 3 shows the quality model for packet loss that consists of two interacting automata: Topology and Network.

The automata have two sets of parameters: (i) parameters to configure the model for an adaptation option and (ii) parameters of the uncertainties that need to be set based on the current conditions.

To configure an adaptation option, the power settings of the motes per link need to be set (with values between 0 to 15) and the distribution factors for links of motes with two parents need to be set (each with a value of 0%, 20%, 40%, 60%, 80%, or 100%). The power of the links is set based

on the current levels of interference along the links that are available in the knowledge repository. The power settings are applied to all adaptation options. The values for the distribution factors are different for each adaptation option; these values are assigned to the variables *probLink*1 and *probLink*2 of the topology model. Furthermore, the values of the uncertainties, network interference (SNR), and traffic load, need to be set. These values are available in the knowledge repository and are based on the recent observations. The uncertainties apply to all adaptation options.

After initializing the model for a particular adaptation option, the *Topology* automaton simulates the communication of data along a path selected for verification, i.e., a sequence of links from one mote via other motes to the gateway (see also Figure 1). The current link to send data is selected probabilistically based on the distribution factors (*probLink*1 and *probLink*2). The model then signals the *Network* automaton. Next, the probability for packet loss is calculated (based on the SNR). Depending on the result, either the packet is lost or the message is received. In the latter case, the network automaton returns to the start location, continuing with the next hop of the communication along the path that is currently checked, until the gateway is reached. If the packet is lost, then the verification of the communication along that path ends. The quality model allows determining the packet loss of the adaptation options using the following query:

$$Pr[<=1](<> Network.PacketLoss).$$

This query determines the probability that the state *Network.PacketLoss* is reached for the different paths of an adaptation option (see the Network automaton in Figure 3). To that end, the verifier performs a series of simulations and returns the expected packet loss with a required accuracy and confidence. These estimates together with the estimates of other quality properties for the different adaptation options are then used to select an adaptation option using the adaptation goals.

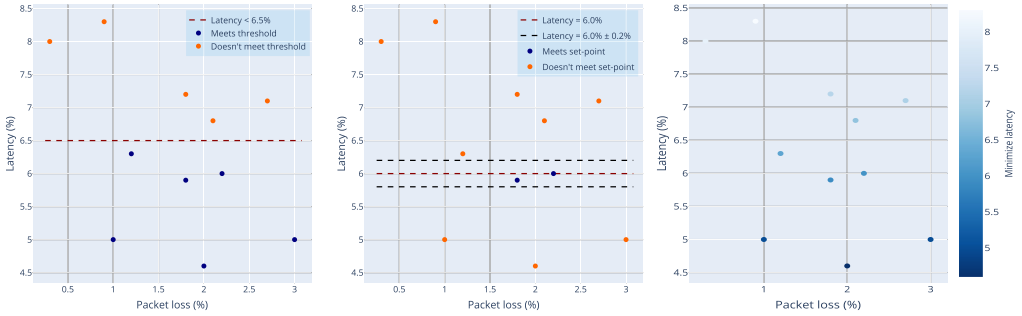
This example illustrates that rigorous runtime analysis can be a resource- and time-consuming activity, implying the need for adaptation space reduction if the number of adaptation options, i.e., the size of the adaptation space, is too big to be completely verifiable within the available time period to make an adaptation decision. We elaborate on the adaptation space below in Section 2.4.

2.3 Adaptation Goals

An adaptation goal refers to a quality requirement of the managed system that the managing system should realize. A violation (or an expected violation) of one or more of the adaptation goals triggers an adaptation, aiming to satisfy all the adaptation goals again.

In this research, we consider three types of adaptation goals: (1) threshold goals, (2) set-point goals, and (3) optimization goals. Intuitively, a threshold goal states that the value of some quality property of the system should be below (or above) a certain threshold value. A set-point goal states that some value of a quality property of the system should be kept at a certain value, i.e., the set-point, with a margin of at most ϵ. Finally, an optimization goal states that some value of a quality property of the system should be minimized (or maximized). Formally, we define the satisfaction of goals by the adaptation options as follows[4]: Consider C the set of possible configurations, each configuration representing an adaptation option of the adaptation space. We refer to a particular quality property q_x of an adaptation option $c_i \in C$ as $c_i[q_x]$, with $x \in \{t, s, o\}$ referring to quality properties related to threshold, set-point, and optimization goals, respectively. Further, consider a threshold goal g_t, a set-point goal g_s, and an optimization goal g_o. The set of adaptation options T

[4]For the explanation, we consider only threshold goals below a value and optimization goals that minimize a value; the other variants are defined similarly.



(a) Threshold goal. (b) Set-point goal. (c) Optimization goal.

Fig. 4. Illustration of the three types of adaptation goals.

that satisfies g_t, the set S that satisfies g_s, and the set O that satisfies g_o are then defined as follows[5]:

$$T = \{c_i \in C \mid c_i[q_t] < g_t\}, \tag{1}$$

$$S = \{c_i \in C \mid g_s - \epsilon \leq c_i[q_s] \leq g_s + \epsilon\}, \tag{2}$$

$$O = \{c_i \in C \mid c_i[q_o] < c_j[q_o],\ \forall c_j \in C \setminus \{c_i\}\}. \tag{3}$$

While DLASeR**+** can handle an arbitrary mix of adaptation goals, in this article, we focus on systems that can have multiple threshold and set-point goals, but only one optimization goal. In particular, the DLASeR**+** architecture we present in this article maps each adaptation goal to a rule. Decisions are made by first applying the rules of the threshold and set-point goals, and then the rule of the optimization goal. Handling multiple optimization goals with different decision-making techniques, e.g., based on Pareto optimality [34], are outside the scope of this article.

As an illustration, Figure 4 shows a latency goal for DeltaIoT specified in three formats corresponding with the three types of goals (the diagrams are simplified for didactic reasons).[6] Each dot in a diagram represents an adaptation option based on the values of latency and packet loss (we are here mainly interested in the values of latency). Figure 4(a) shows the latency as a threshold goal `Latency < 6.5%`. All the dots below the red dotted line represent adaptation options that satisfy the threshold goal. The adaptation options above the red dotted line do not meet this goal. Figure 4(b) shows the latency as a set-point goal `Latency = 6% ± 0.2%`. The two blue dots are the only adaptation options that meet this set-point goal, since both are in the range defined by the set-point value and the margin ϵ. The other adaptation options have values out of the range [5.8%, 6.2%] and do not satisfy the goal. Finally, Figure 4(c) shows the latency as an optimization goal. The darker the dot, the lower the latency, hence, the dot at the bottom has the lowest value for latency and this adaptation option optimizes the latency.

For the evaluation with DeltaIoT in Section 7, we consider three combinations of adaptation goals: (1) two threshold goals: packet loss and latency, with energy consumption as optimization goal, (2) the same threshold goals, with energy consumption as set-point goal, and (3) latency as threshold goal, energy consumption as set-point goal, and packet loss as optimization goal.

[5]While a set-point goal may conceptually be modeled as two threshold goals, there are good arguments to differentiate them as a distinct type of goal. In particular, using a set-point to express the corresponding goal is more straightforward and natural for stakeholders compared to using two thresholds. Further, it makes maintenance easier, e.g., when updating the set-point value of the goal. Last, from a learning perspective, if we use two threshold goals instead of one set-point goal, then we require two times the processing resources to train and infer.

[6]In DeltaIoT, latency is defined as a relative part of the cycle time of the time-synchronized communication in the network. E.g., a latency of 5% means that the average time packets stay in the network is 5% more as the cycle time.

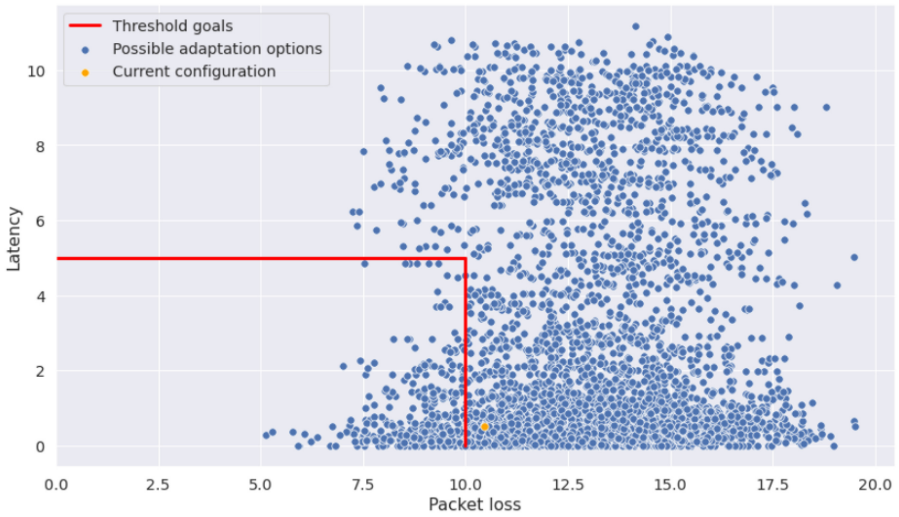


Fig. 5. Adaptation space of DeltaIoTv2 at some point in time.

2.4 Adaptation Space

An adaptation space comprises the set of adaptation options at some point in time. The adaptation space is determined by the effectors (also called actuators or "knobs") that are available to adapt the managed system. The actuators allow to set system variables. These variables are usually characterized by a discrete domain (e.g., start/stop a number of servers, launch a number of new virtual machines, select a power setting, send a number of messages over a link). For variables with a continuous domain, we assume in this work that the domain can be discretized.

If adaptation is required, then it is the task of the analyzer and the planner of the MAPE-K feedback loop to find the best adaptation option. This task involves estimating the expected quality properties related to the adaptation goals for all or at least a relevant subset of the adaptation options. Different techniques can be used to make these predictions. One approach is to represent the quality of the system as a network of stochastic timed automata [48, 77, 82] as we illustrated in Section 2.2. In this approach, the model is configured for the effector settings of a particular adaptation option, and parameters that represent uncertainties are assigned up-to-date values. By utilizing statistical model checking, one can determine the expected quality of an adaptation option. Another approach is to represent the quality of the system as a parameterized Markov model [8, 9, 50] and apply probabilistic model checking [48, 82] to estimate the quality property of interest. Statistical model checking is more efficient than probabilistic model checking, but offers results bounded to a given accuracy and confidence, depending on the number of simulation runs.

In the case of DeltaIoT, we are interested in the failure rate of the network, its latency and the energy consumed to communicate the data to the gateway. Figure 5 shows a representation of the adaptation space for DeltaIoTv2 at some point in time.

The red lines denote two threshold goals for this particular scenario (latency and packet loss). Each blue dot in this graph represents one adaptation option. The dot in orange represents the current configuration. For this particular instance, the number of adaptation options is over 4,000. Analyzing all these options within the available time slot may not be feasible. Hence, the analysis should focus on the relevant options for adaptation, i.e., those that are compliant with the goals, represented by the dots in the box bottom left as determined by the two threshold goals. This article is concerned with analyzing such a large adaptation space in an effective and efficient manner.

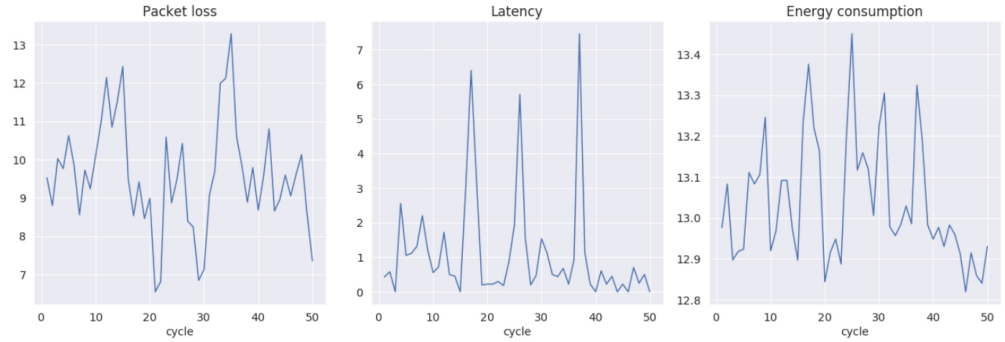


Fig. 6. Dynamic behavior of an adaptation option over 50 cycles reflected in changes of its quality properties.

It is important to notice that the adaptation space is dynamic, i.e., the qualities of each adaptation option may change over time. In Figure 5, this means that the position of the adaptation options on the diagram move over time. The main cause for this dynamic behavior are the uncertainties the system is subjected to. It is due to this dynamic behavior that the analysis and verification needs to be performed at runtime, when the actual values of the uncertainties are available. Figure 6 illustrates the dynamic behavior for an instance of DeltaIoT. The figure illustrates how three qualities of one of the adaptation options change over a series of cycles. These graphs are based on field tests.

2.5 Deep Learning in a Nutshell

Deep learning (DL) refers to a subset of machine learning mechanisms in which the learning methods are based on ***deep artificial neural networks* (ANNs)** with *representation learning* [32].

2.5.1 Artificial Neural Networks. ANNs are the core of deep learning. These networks are commonly described in terms of their input and output, which are the only externally observable parts of an ANN. Both the input and the output are represented as a vector of numbers. A neural network applies a chain of mathematical operations aiming to transform the input to the desired output.

The basic building block of a neural network is a neuron, also called a perceptron. In the case of a fully connected neural network, a neuron is connected to all the inputs and produces a single output. To obtain its output, the neuron applies two operations:

$$z = \vec{w} \cdot \vec{x} + b, \tag{4}$$

$$y = f(z), \tag{5}$$

with $\vec{x}$ the current input vector, $\vec{w}$ the weights associated with the inputs, b a constant, z an intermediate computed value, y the output, and $f(\cdot)$ an activation function. First, the weighted sum z of all the inputs is calculated (Equation (4)). The weights associated with the inputs are learnable. Intuitively, these weights represent the relative importance of the inputs. When the weighted sum is computed, the neuron applies an activation function f (Equation (5)). This function allows introducing non-linearity in the neural network. Since a weighted sum is a linear function, and a linear combination of linear functions is still a linear combination, a neural network without a non-linear activation function will only be able to learn linear combinations of input values. Non-linearity in learning is very important to learn more complex concepts. Examples of common activation functions are hyperbolic tangent (tanh), sigmoid, and **rectified linear unit (ReLU)**.

A deep neural network is structured in two dimensions: width and depth. The depth of a network corresponds to the number of layers, whereas each layer is defined by its width, i.e., the number of neurons present in that layer.

2.5.2 Representation Learning. A key concept of ANNs is representation learning [32]. Representation learning learns representations of input data, typically through transformations, which makes it easier to perform certain tasks [6]. An important advantage of representation learning is that it enables transfer learning, i.e., reuse of the learned representations, complemented with fine-tuning for specific learning tasks [54]. In deep learning, representations are learned through multiple non-linear transformations on the input data. These learned representations provide abstract and useful representations for the learning task. A simple linear layer can then be stacked on top of the complex block with many non-linear layers to tailor the learning network to the specific learning task at hand, e.g., classification, regression, or prediction [19].

2.5.3 Training a Deep Neural Network. We distinguish three steps in the training of a ANN: (1) forward propagation, (2) loss calculation, and (3) back propagation.

In the *forward propagation* step, the input data is passed through the layers of the network, from input to output. The neurons apply their transformations to the data they receive from all the connected neurons of the previous layer and pass the result to the connected neurons of the next layer. In the final layer, the neurons apply their transformations to obtain the (predicted) output.

In the second step, a loss function estimates the loss (or error). The loss captures how good or bad the predicted result is (i.e., the output predicted by the network) compared to the correct result (i.e., the expected output). To obtain a loss as close as possible to zero, the weights of the connections between the neurons (that determine the weighted sums) are gradually adjusted in the next step.

As final step, the loss value is propagated backwards through the network, hence *back propagation*. You can think of this step as recursively applying the chain rule to compute the gradients all the way from the output to the input of the network. These gradients tell in which direction the loss is increasing and indicate the influence of the computations of the neurons on the loss. An optimizer exploits these gradients to update the weights of the neuron connections aiming to minimize the loss. Examples of commonly used optimizers are Adam, RMSprop, and Nadam [64].

2.5.4 Classification and Regression. Classification and regression are two important learning tasks that require different predictions. Suppose that we have some machine learning model that is described by the following equation:

$$\vec{y} = M(\vec{x}), \tag{6}$$

with $\vec{x}$ the input and $\vec{y}$ the output of the learner, and M a function that maps input to output.

In the case of classification, M needs to map the input to a set of classes that are represented by different labels with different encodings (e.g., rainy = 0, and sunny = 1). A classification with only two labels is called binary classification; while multi-class classification has more than two labels. Regression, however, maps the input data to continuous real values instead of classes or discrete values. Whereas for classification the predicted output is not ordered, for regression the output is ordered. In sum, the main difference between both tasks is that the output variable $\vec{y}$ for classification is categorical (or discrete), while for regression it is numerical (or continuous).

3 OVERVIEW OF THE RESEARCH METHODOLOGY

To tackle the research question, we followed a systematic methodology as shown in Figure 7.

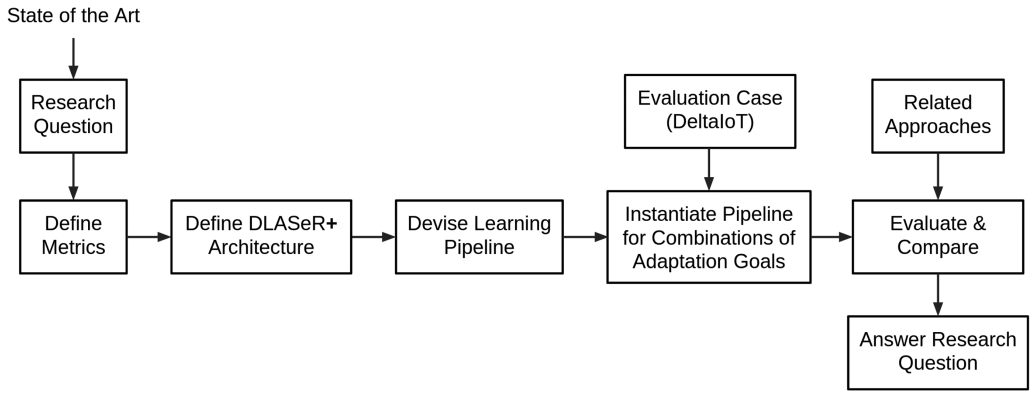


Fig. 7. Overview of research methodology.

Table 1. Left: Different Aspects of Research Question to Be Evaluated; Right: Metrics for Each Aspect

Effectiveness	
Coverage relevant adaptation options	F1-score (for threshold and set-point goals) Spearman's rho (for optimization goal)
Reduction adaptation space	Average adaptation space reduction (AASR) Average analysis effort reduction (AAER)
Effect on realization adaptation goals	Differences in mean values of quality properties of the goals with and without learning
Efficiency	
Learning time	Time used for learning (training + prediction, expressed in s) Time reduction for analysis with learning (expressed in %)
Scalability	Metrics above for increasing size of adaptation space

Based on a study of the state-of-the-art and our own experiences with applying machine learning in self-adaptive systems, we defined the research question (see Section 1). Once we determined the research question, we specified the metrics that enabled us to measure the effectiveness and efficiency of DLASeR**+** and compared it with other approaches. Then, we defined the DLASeR**+** architecture that is able to deal with threshold goals, set-point goals, and an optimization goal. Next, we devised DLASeR**+**'s learning pipeline that applies deep learning for the three types of adaptation goals. We instantiated this pipeline for various setups for the two evaluation cases of DeltaIoT applying different combinations of adaptation goals. We then evaluated DLASeR**+** and compared it with representative related approaches. Finally, we answered the research question and reflected and discussed the extent to which we met our objectives.

In the next sections, we zoom in on the different steps of the methodology. We start with explaining the metrics we used to design and evaluate DLASeR**+** (Section 4). Then, we zoom in on the DLASeR**+** architecture (Section 5) and we explain how to engineer a solution with DLASeR**+** (Section 6). Next, we present the evaluation of DLASeR**+**, answering the research question (Section 7). Finally, we discuss related work (Section 8) and conclude the article (Section 9).

4 METRICS

To answer the research question, we determined metrics that allow us to evaluate the different aspects of effectiveness and efficiency of the proposed solution. Table 1 summarizes the metrics.

We use different metrics to capture the coverage of relevant adaptation options for classification and regression. In particular, we use the F1-score for classification and Spearman's rank correlation coefficient (Spearman's rho) for regression.[7] We define **average adaptation space reduction (AASR)** and **average analysis effort reduction (AAER)** to capture and compare the reduction of adaptation spaces. To capture the effect on the realization of the adaptation goals, we use the differences in mean values over time for the corresponding quality properties with and without learning. We measure the time used for learning[8] and compare this with the time that would be necessary to analyze the complete adaptation space. Finally, for scalability, we apply the different metrics for scenarios with an increasing size of adaptation spaces. We further elaborate on F1-score, Spearman's rho, AASR, and AAER. The other basic metrics are further explained in the evaluation section.

4.1 F1-score: Precision and Recall

The quality of a classification algorithm is often expressed in terms of precision and recall. Applied to the classification of adaptation options (relevant or not relevant according to the adaptation goals), precision is the fraction of selected adaptation options that are relevant, while recall is the fraction of the relevant options that are selected from the total number of relevant options. Hence, precision and recall are a measure of relevance. The F1-score combines both metrics with equal importance into a single number. Concretely, the F1-score is defined as the harmonic mean of precision and recall [27]:

$$F1 = 2 * \frac{precision * recall}{precision + recall}. \tag{7}$$

4.2 Spearman Correlation

Spearman correlation, or Spearman's rho, is a non-parametric metric that can be used to measure rank correlation. We use this as a metric to capture the ranking of the predicted values of quality properties of regression models. The Spearman correlation essentially converts the predicted values to numeric ranks and then assesses how well a monotonic function describes the relationship between the predicted ranks and the true ranks. Spearman's rho is defined as [44]:

$$\rho_{xy} = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}. \tag{8}$$

This formula computes the Spearman correlation between x (the predictions of the selected adaptation options) and y (the true values for the selected adaptation options), with n the number of observations (the number of selected adaptation options). The result is a value between 0 and 1 (for an increasing monotonic trend, or -1 for a decreasing monotonic trend). Large errors are penalized harder. For example, a swapping of the first and third rank in the prediction results is worse (lower Spearman's rho) compared to a swapping of the first and second rank.

4.3 Average Adaptation Space Reduction

To capture how well the adaptation space is reduced, we define a new metric called **average adaptation space reduction (*AASR*)**. *AASR* is defined as:

$$AASR = \left(1 - \frac{selected}{total}\right) \times 100, \tag{9}$$

[7] A common metric to evaluate regression models is **mean squared error (MSE)**. However, since we are more interested in the ranking of the regressed output, we use Spearman's rho to capture this ranking, which is not covered by MSE.

[8] The learning time refers to the time the system uses for training and making predictions of the quality properties of interest for the set of adaptation options that are considered for the respective quality properties.

with *selected* the number of adaptation options selected by learning (over multiple adaptation cycles) and *total* the total number of adaptation options (of multiple adaptation cycles). For instance, an average adaptation space reduction of 70% means that after learning, only 30% of the original adaptation space is considered for analysis. *AASR* is a particularly suitable metric for stakeholders, as it covers the high-end goal of adaptation space reduction and allows comparing solutions.

Note that the average adaptation space reduction is determined by the system's adaptation goals. In particular, for the three types of goals considered in this work, the *AASR* is determined by the threshold and set-point goals, corresponding to the percentage of adaptation options that are predicted to conform with these goals. For systems with only optimization goals, the *AASR* is zero (since the selected and total number of adaptation options are the same).

It is also interesting to note that the *AASR* depends on the restrictiveness of both threshold and set-point goals. Suppose that the threshold and set-point goals are not very restrictive, thus many adaptation options will comply with the adaptation goals. In this case, the reduction will be rather small. In the other case, for very restrictive threshold and set-point goals, the opposite is true. A larger reduction for these goals can be expected.

4.4 Average Analysis Effort Reduction

To capture the effect of the adaptation space reduction on the effort required for analysis, we define a new metric called **average analysis effort reduction (***AAER***)**. *AAER* is defined as:

$$AAER = \left(1 - \frac{analyzed}{selected}\right) \times 100, \tag{10}$$

with *analyzed* the number of adaptation options that have been analyzed (over multiple adaptation cycles) and *selected* the number of adaptation options selected by learning (over multiple adaptation cycles). For instance, an average analysis effort reduction of 90% means that only 10% of the adaptation options selected by learning were analyzed to find an option to adapt the system. Similarly to *AASR*, *AAER* also covers a high-end goal of adaptation space reduction.

Note that the *AAER* depends on the analysis approach that is used to find an adaptation option from the reduced adaptation space (and not on the constraints imposed by the adaptation goals as for *AASR*). In particular, the *analysis reduction* corresponds to 100% minus the percentage of selected adaptation options that have to be analyzed until an adaptation option is found that meets all the threshold and set-point goals. For systems that include threshold and set-point goals and an optimization goal, the selected adaptation options are analyzed in the order predicted for the optimization goal. For systems with only threshold and set-point goals, the most basic method to analyze the selected adaptation options is random order. For the evaluation of this specific case, we randomly shuffle the selected options per cycle, resulting in a more representative *AAER* score. Note that for systems with a large adaptation space and only an optimization goal, *AAER* $\approx$ 100% (since *analyzed* is 1 and *selected* is a large number equal to the size of the adaptation space).

5 DLASER+ ARCHITECTURE

We introduce now the novel adaptation space reduction approach "Deep Learning for Adaptation Space Reduction Plus"—DLASeR**+**. We start with outlining how this approach deals with different types of adaptation goals and how learning for these adaptation goals is combined into a unified architecture. Finally, we zoom in on the neural network architecture of DLASeR**+**.

5.1 Adaptation Space Reduction for Different Types of Adaptation Goals

DLASeR**+** can reduce the adaptation space for adaptation goals based on thresholds, set-points, and optimization goals.

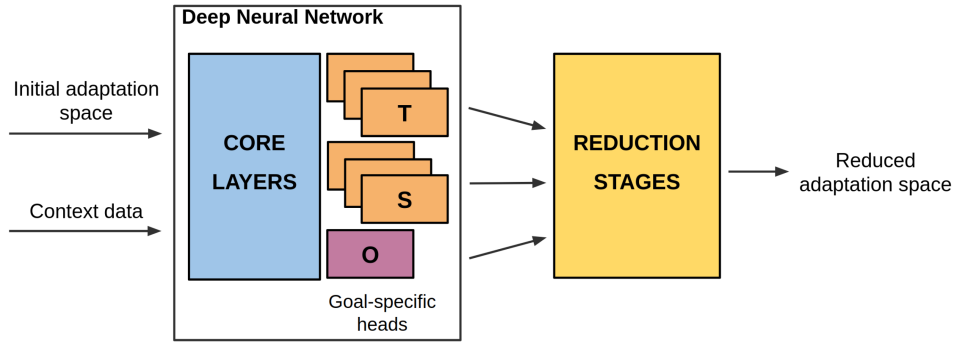


Fig. 8. Unified architecture of DLASeR+ for a mix of threshold and set-point goals and one optimization goal.

5.1.1 Threshold Goals. To deal with threshold goals, DLASeR**+** relies on classification deep learning. Concretely, we apply binary classification using class 1 (true) if an adaptation option meets the threshold goal and class 0 (false) otherwise. We say that an adaptation option meets a threshold goal when the associated quality property of that adaptation option is below (or above) the given threshold value g_t of that goal. Hence, DLASeR**+** reduces the adaptation space to the adaptation options that are valid, i.e., that are classified as 1. In case of multiple threshold goals, the adaptation space is reduced to the intersection of the subsets of adaptation options with predicted values that are classified as 1 for each of the different threshold goals.

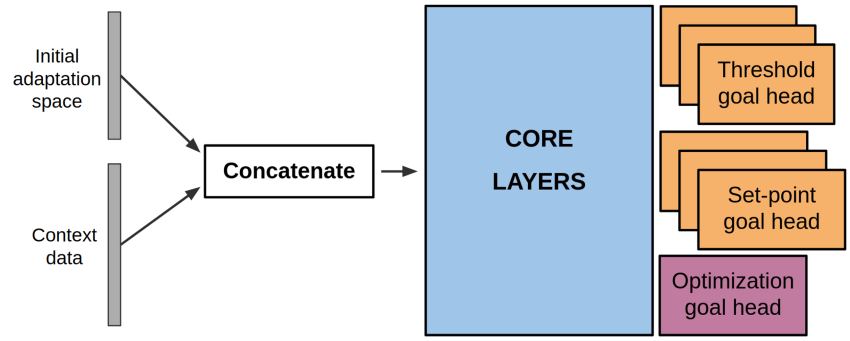
5.1.2 Set-point Goals. For set-point goals, DLASeR**+** also relies on classification deep learning. When a user defines a set-point goal, he or she has to specify a set-point value g_s and a bound ϵ. This bound defines the range $[g_s - \epsilon, g_s + \epsilon]$ in which adaptation options are considered valid.[9] Hence, DLASeR**+** again applies binary classification; class 1 for adaptation options that are predicted within the interval and class 0 otherwise. In case the self-adaptive system has multiple set-point goals, the reduced adaptation space is the intersection of the subsets of adaptation options with predicted values that are classified as 1 for each of the different set-point goals.

5.1.3 Optimization Goals. DLASeR**+** handles optimization goals using regression deep learning. Based on the regressed (predicted) values of a quality property, the adaptation options are ranked. From this ranking the adaptation option that maximizes or minimizes the adaptation goal can be derived. The adaptation space reduction is then determined by the number of adaptation options that need to be analyzed to make an adaptation decision. In the case of a single optimization goal, as we consider in this work, the adaptation space reduction is determined by the number of ranked adaptation options that need to be analyzed before one is found that complies with the other goals.

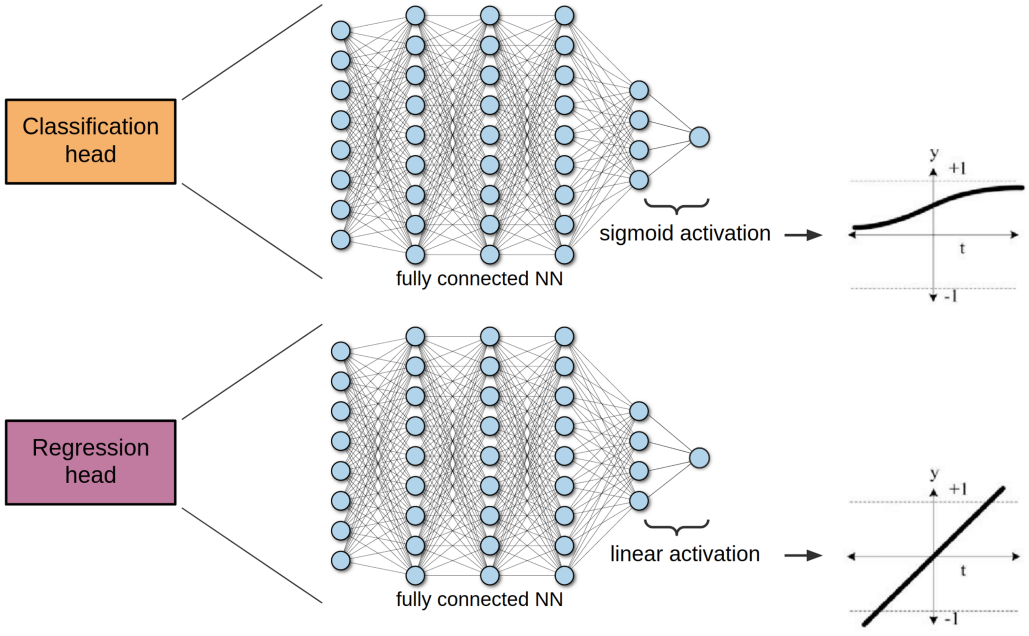
5.2 Unified DLASeR+ Architecture

We explained how DLASeR**+** handles single types of adaptation goals, however, practical systems usually combine different types of adaptation goals. Over the years, different techniques have been developed to combine adaptation goals for the decision-making of self-adaptation. Classic examples are goals, utility functions, and rules [15, 28, 68]. DLASeR**+** offers a unified architecture for adaptation space reduction for a class of systems that combine multiple threshold and set-point goals with a single optimization goal. Our focus is on rule-based goals that are representative for a large number of practical self-adaptive systems. Figure 8 shows the unified DLASeR**+** architecture.

[9]From a control theoretic perspective, this bound corresponds to the steady state error.



(a) Architecture deep neural network.



(b) Internal structure of the classification and regression heads in the DLASeR+ architecture.

Fig. 9. Overview of the internal deep neural network (NN) architecture of DLASeR+.

The deep neural network takes as input the initial adaptation space that consists of all the adaptation options, together with context data, such as the actual values of uncertainties in the environment and the current values of the relevant qualities. Internally, the DLASeR+ architecture is centered on a single deep neural network that covers all the adaptation goals. The deep neural network consists of a number of core layers, complemented with goal specific heads. The core layers are shared among the different goals. Each head deals with a specific goal. The output produced by the deep neural network is then combined to produce the reduced adaptation space. The layered structure of DLASeR+ with shared core layers and goal specific heads adds to the modularity of the learning architecture supporting modifiability and extensibility.

5.2.1 Internal Neural Network Architecture. The core of the DLASeR+ approach is a deep neural network that is kept updated at runtime. Figure 9 shows the internal architecture of the neural

network. The figure at the top shows the structure of the network and the flow of the data through the network. The network starts with concatenating the input into input vectors, one vector per adaptation option. The input vectors include high-level data relevant to adaptation in two parts: data of the adaptation options, such as the settings of the system, and data of the context, such as the current configuration, the recent load of the system, and recent values of uncertainty parameters. The data of the input vectors are then fed to the core layers of the network that are modeled as a fully connected network of neurons. The output of the last layer of the core layers encodes the input of goal-specific heads. DLASeR**+** supports two types of goal-specific heads that can be added on top of the core layers: classification heads for threshold and set-point goals and a regression head for an optimization goal. Figure 9 at the bottom illustrates the difference between the two types. Both types of heads are fully connected neural networks that produce a single output, i.e., the output layer has a dimensionality of 1. However, the heads differ in the output they produce. Classification heads that use a sigmoid activation function produce values between 0 and 1. These values are classified based on predefined thresholds; for instance, all values below 0.5 are classified as class 0 and all values above (and including) 0.5 as class 1. The regression head that uses a linear activation function produces values that predict the quality property that needs to be optimized.

5.2.2 Adaptation Space Reduction Stages. By using a single deep neural network, the input data passes only once through the neural network to produce the output for the different adaptation goals. This output is then combined to reduce the adaptation in two stages. The first stage uses classification deep learning to reduce the adaptation space to the adaptation options that are classified as being valid according to the threshold and set-point goals. We refer to the first stage as the *classification stage*. The second stage that deals with the optimization goal further reduces the adaptation space obtained from stage 1 by ranking the regressed values of the adaptations options. The adaptation options are ranked from low to high in the case of a minimization goal and vice versa for a maximization goal. The ranked adaptation options can then be analyzed one-by-one until an option is found that satisfies (e.g., determined by verification) all the other goals. We refer to the second stage as the *regression stage*.

It is important to note that either of these stages can be omitted when the corresponding type(s) of goals are not present for a problem at hand. For instance, for a system with only threshold goals (for which DLASeR**+** uses classification), the second stage can be omitted.

Figure 10 schematically illustrates the two reduction stages, showing how the predictions of the neural network for different adaptation goals are combined to reduce an adaptation space.

In stage 1, all the adaptation options for all the threshold and set-point goals are classified relevant or not relevant. Then, the results are aggregated, reducing the initial adaptation space to the intersection of the adaptation options that were classified relevant for all the aforementioned goals. In stage 2, DLASeR**+** ranks the subset of adaptation options obtained from stage 1. This ranking is based on the regressed quality for which the optimization goal was defined and depends on the optimization goal, i.e., ascending and descending order for, respectively, minimization and maximization. The adaptation options of the reduced adaptation space can then be analyzed in the order of the ranking until one is found that satisfies all the adaptation goals.

6 ENGINEERING WITH DLASER+

We explain now how to engineer a solution for adaptation space reduction with DLASeR**+**. Central to this is the learning pipeline that consists of an offline and online part. During the offline part, an engineer selects a model for the deep neural network model and a scaler. During the online part, the running system uses the model and scaler to reduce adaptation spaces and exploits newly obtained data from analysis to update the model and the scaler.

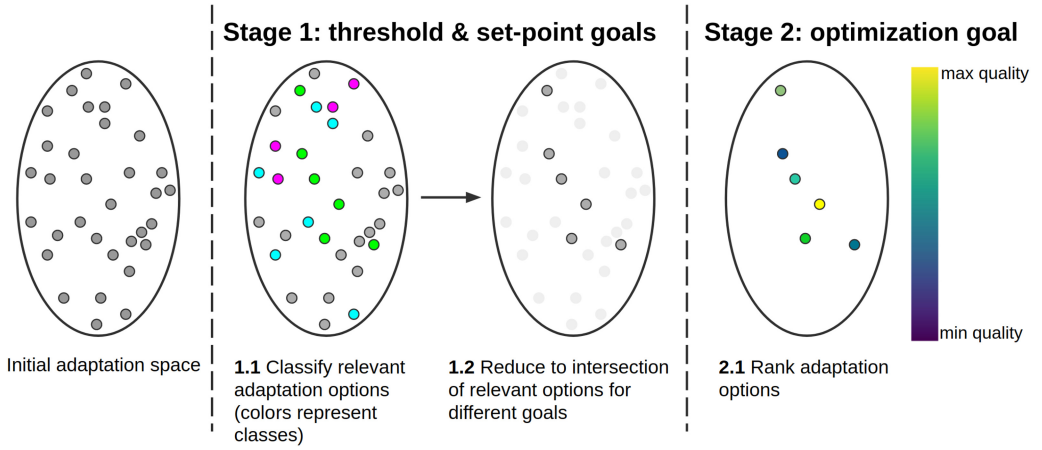


Fig. 10. Illustration of the two stages of DLASeR**+**. Each dot represents an adaptation option. In the first stage the initial adaptation options are classified for threshold and set-point goals (**1.1**): Blue dots satisfy the threshold goals, pink dots satisfy the set-point goals, and green dots satisfy both types of goals. Then, the intersection of adaptation options that satisfy all threshold and set-point goals are kept, i.e., the green dots (**1.2**). In the second stage, the relevant subset of adaptation options obtained in stage 1 are ranked (**2.1**).

6.1 Offline Part of the DLASeR+ Learning Pipeline

During the offline part of the pipeline, shown in Figure 11, the engineer collects data of a series of adaptation cycles (via observation or simulation of the system). This data comprises the adaptation options, context info, and also the associated qualities of the adaptation options. The collected data is then concatenated in *input vectors*, one per adaptation option. Each item of an input vector refers to a measurable property or characteristic of the system that affects the effectiveness of the learning algorithms. As an example, an input vector for DeltaIoT contains:

- Data of an adaptation option, i.e., the settings of the transmission power for each link and the distribution factors per link that determine the percentage of messages sent over the links;
- Context data, i.e., the traffic load generated per mote and the signal-to-noise ratio over the links (uncertainties), the current system configuration.

The qualities associated with the adaptation options are used to validate the output of the generated goal-specific heads. As an example, the current qualities (one per adaptation goal) for DeltaIoT are:

- The packet loss along the links;
- The energy consumed by the motes.

To successfully train a deep neural network, it is important that all relevant data is collected.[10]

The aggregated data is then used to perform so-called back-testing on the application to evaluate the performance of candidate learning solutions. To that end, the aggregated data is split in two sets: training data and validation data. It is crucial that both sets do not overlap. An overlap of the

[10]It is important to note that deep learning can work with "raw data" without the need for transforming and aggregating features, and so on, as with classic machine learning that requires a substantial effort of engineers. Using the raw data, the deep neural network will learn complex relations automatically.

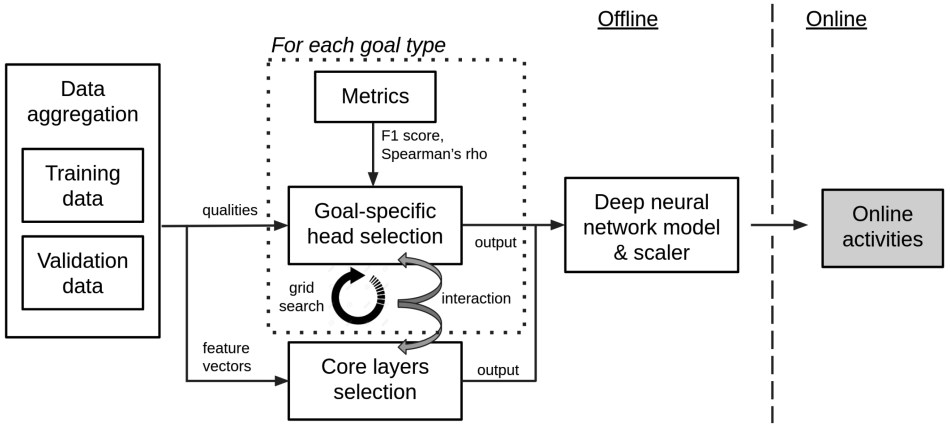


Fig. 11. The offline part of the DLASeR**+** learning pipeline. During model selection the optimal hyper-parameters are determined for the shared layers (*Core layers selection*) and for the goal-specific heads (*Goal-specific head selection*). The output of the offline part of the pipeline is a completely configured deep neural network model (of core layers complemented with the various goal-specific heads) and a scaler.

two sets would introduce data-leakage, where knowledge of the test set leaks into the dataset used to train the model. This would jeopardize the validity of the results [7].

The main activity of the offline stage of the pipeline is selecting a deep neural network model based on finding optimal hyper-parameter settings. Hyper-parameters are non-learnable parameters that control the learning process. The main hyper-parameters are the number of layers of the deep neural network and the number of neurons per layer. Other hyper-parameters that apply for deep learning architectures are the scaler algorithm (normalizes the data, improving the learning process), the batch size (defines how much of the data samples are considered before updating the model affecting the learning speed), the learning rate LR (determines the impact of on update, affecting the learning speed), and the optimizer (influences the updates of model parameters utilizing the learning rate and the local gradient at the neurons).

In DLASeR**+**, we distinguish between hyper-parameters for the shared core layers and for goal-specific heads. *Core layers selection* deals with the hyper-parameters of the shared core layers, while *goal-specific head selection* deals with the hyper-parameters of the goal-specific heads. To determine the optimal values of the hyper-parameters for the DLASeR**+** neural network model, we applied grid search [27]. With grid search, the neural network model is trained and then evaluated for every combination of hyper-parameters.[11] We used different metrics during evaluation: (i) F1-score that combines precision and recall of predicted classes of adaptation options and (ii) Spearman correlation that measures the ranking of predicted values of quality properties of regression models; see the explanation in Section 4. Once the models are trained (on the training dataset), they are evaluated on the validation dataset, i.e., the predictions are evaluated using the validation data. The best model is then selected based on the validation loss that is determined by a loss function that sums the losses of the heads, capturing the overall quality of the neural network model.

When the core layers and goal-specific heads are fine-tuned and a proper scaler is found the integrated solution can be deployed, which brings us to the second stage of the learning pipeline.

[11]Technically, we consider the layout of the core layers and the goal-specific heads (number of layers and neurons per layer) as distinct hyper-parameters. However, these hyper-parameters are optimized together in a single grid search process, since the loss function from the goal-specific heads guides the learning process, including the learning of core layers.

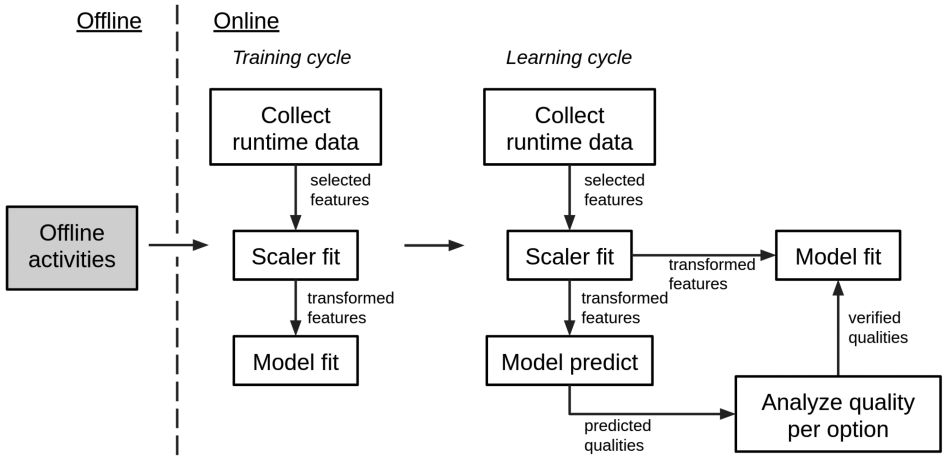


Fig. 12. The online part of the DLASeR**+** learning pipeline. In the training phase (training cycles), the deep neural network model is initialized for the problem at hand. In the learning phase (learning cycles), adaptation space reduction is applied and new data is used to update the deep neural network model (online learning).

6.2 Online Part of the DLASeR+ Learning Pipeline

During the online part of the learning pipeline, the deep neural network supports the decision-making process of a self-adaptive system with reducing large adaptation spaces. The online part, shown in Figure 12, consists of two consecutive phases. In the training phase, which consists of a series of training cycles, the deep learning model is initialized based on the current state of the system exploiting all available runtime data. In the learning phase, which consists of learning cycles, the deep learning model performs adaptation space reduction and is updated using online learning.

6.2.1 Training Phase. The goal of the training phase is to initialize the learnable parameters of the model for the problem at hand. During the training cycles, relevant runtime data is collected to construct input vectors. This data is used to update the scaler (e.g., updating min/max values of parameters, the means) and the input vectors are adjusted accordingly. The deep neural network model is then trained using the transformed data; i.e., the parameters of the model are initialized, e.g., the weights of the contributions of neurons. The training phase ends when the validation accuracy stagnates, i.e., when the difference between the predicted values with learning and the actual verification results are getting small.[12] During the training cycles, the complete time slot available for adaptation is used for analyzing adaptation options to generate data that is used to initialize the deep neural network model; hence, there is no reduction of the adaptation space yet.

6.2.2 Learning Phase. In the learning phase, the deep neural network model is actively used to make predictions about the adaptation options for the various adaptation goals aiming to reduce the adaptation space before the analysis of the reduced adaptation space. Furthermore, the results of the analysis of the reduced adaptation space are then used to incrementally update the learning model. Concretely, in each learning cycle, an input vector is composed for each adaptation option that is scaled using the scaler. Scaling normalizes the range of independent variables of data, ensuring that each element contributes approximately proportionately to the result. The

[12]For the DeltaIoT, the training phase ended after 45 training cycles (one per adaptation cycle); see Section 7.

ALGORITHM 1: Stage 1: classification stage

```
1:  pred_subspace ← K.adaptation_options
2:  DNN_output ← K.DNN_model.predict(input_vectors)
3:  if K.threshold_goals == None & K.set_point_goals == None then
4:      Proceed to the regression stage: Algorithm 2
5:  end if
6:  for each class_goal ← K.classification_goals do
7:      pred_subspace ← pred_subspace ∩ DNN_output.class_goal
8:  end for
9:  if K.optimization_goal ≠ None then
10:     Proceed to the regression stage: Algorithm 2
11: end if
12: pred_subspace.shuffle()
13: valid_found ← False, idx ← 0
14: verified_subspace ← ∅
15: while not valid_found & idx < pred_subspace.size do
16:     adapt_opt ← pred_subspace[idx]
17:     Analyzer.analyzeAdaptationOptions(adapt_opt)                ▷ Knowledge
18:     _, qualities ← K.verification_results[idx]
19:     if qualities meet all threshold and set-point goals then
20:         valid_found ← True
21:     end if
22:     verified_subspace.add(adapt_opt)
23:     idx ← idx + 1
24: end while
25: Select valid adaptation option adapt_opt, if none valid use fall back option
26: unselected ← K.adaptation_options \ pred_subspace
27: explore ← unselected.randomSelect(exploration_rate)
28: Analyzer.analyzeAdaptationOptions(explore)                      ▷ Knowledge
29: input_vectors, qualities ← K.analysis_results
30: K.DNN_model.update(input_vectors, qualities)
```

deep neural network model then makes predictions for the input vectors. Based on these predictions, the adaptation space is reduced and the adaptation options of the reduced adaptation space are then analyzed. In the evaluation of DLASeR**+** (Section 7), we use runtime statistical model checking [2, 17, 74, 75, 78] to analyze the adaptation options, however, other analysis techniques can be applied.

The analysis of the reduced adaptation space depends on the types of adaptation goals at hand. For a self-adaptive system with a mix of threshold and set-point goals and an optimization goal, analysis consists of two stages: the classification stage and the regression stage; see Section 5.2.2. For systems without optimization goal, only the classification stage applies. Similarly, for systems with only an optimization goal, only the regression stage applies.

In the *classification stage,* the adaptation space is reduced to the relevant subset of adaptation options that comply with the threshold and set-point goals. Algorithm 1 shows how the adaptation space reduction is applied and how analysis is performed in the classification stage.

In lines 1 and 2, we initialize the variable *pred_subspace* to the complete adaptation space and store the predicted output of the deep neural network in the *DNN_output* variable. Both these

variables are used in Algorithm 1 and Algorithm 2. *K.DNN_model* refers to the deep neural network model that is stored in the Knowledge module (*K*). Since DLASeR**+** uses a single deep learning model for all the goals, prediction can be done in a single step. Note that the *predict()* function in line 2 scales the elements of the input vectors before making predictions. Lines 3 to 5 check whether there are any threshold or set-point goals. If there are no threshold nor set-point goals, then we proceed to the regression stage, i.e., Algorithm 2. In the other case, i.e., there are threshold and/or set-points goals, the adaptation space (*pred_subspace*) is reduced in lines 6 to 8. Concretely, we reduce the adaptation space to the intersection of the predicted subspaces that are relevant for each of these two types of classification goals. This step represents the actual adaptation space reduction for all threshold and set-point goals. The adaptation option is analyzed to check whether it complies with the threshold and set-point goals. If this is the case, then this option is selected for adaptation. If not, then analysis is continued until an adaptation option is found that satisfies the threshold and set-point goals. Line 9 checks whether there is an optimization goal. If there is such a goal, then the classification stage ends and the system continues the regression stage using the reduced subset, i.e., *pred_subspace* (see line 10). If there is no optimization goal, then the adaptation options for analysis are selected. We first shuffle the relevant subspace to avoid bias in the way the adaptation options are determined (see line 12), prepare the analysis (see lines 13 and 14), and then iterate over the adaptation options top-down (see lines 15 to 24). In this iteration, an adaptation option is analyzed to check whether it complies with the threshold and set-point goals. If this is the case, then this option is selected for adaptation and the iteration halts. If not, then analysis is continued until an adaptation option is found that satisfies the threshold and set-point goals. After iterating over the subspace of adaptation options, line 25 selects a valid option to adapt the system, i.e., an adaptation option that satisfies all the threshold and set-point goals. If no valid adaptation option is found according to the goals, then a fall-back option is used that implements a graceful degradation strategy. Then, lines 26 and 27 use the *exploration_rate* to select a random sample of adaptation options from the options that were not analyzed. Adding this random sample aims at anticipating potential concept drifts that might occur in dynamic environments after a large number of adaptation cycles.[13] These additional options are then also analyzed (see line 28) and the analysis results are stored in the knowledge. Finally, the analysis results are exploited to update the deep neural network model (see lines 29 and 30). To that end, the input vectors (configurations, etc.) and the analysis results (i.e., the qualities per goal obtained by verification) of the analyzed adaptation options are retrieved from the knowledge. Based on this data, the neural network model is updated using the same learning mechanism as used in the training cycle.

The *regression stage* starts either from the adaptation options selected in the classification stage or from the complete adaptation space in case there is only an optimization goal. The set of adaptation options are then ranked according to the predicted quality of the optimization goal. Algorithm 2 shows how the adaptation options are ranked and how one of the options is selected based on the result of the analysis and its compliance with the threshold and set-point goals.

In lines 2 to 4, the predictions of the previous stage are reused to obtain a ranking of the relevant adaptation options. Since we have a single deep neural network model, we require only a single prediction (see Algorithm 1, line 2). Lines 5 to 16 iterate over the ranked adaptation options in descending order of the predicted value for the quality of the maximization goal (the opposite order is used for a minimization goal). The adaptation option is analyzed to check whether it complies with the threshold and set-point goals. If this is the case, then this option is selected for adaptation. If not, then analysis is continued until an adaptation option is found that satisfies the

[13]Intuitively, one may argue to select this sample nearer to the boundaries set by the thresholds rather than random, yet, this may reduce the intended effect on potential concept drifts. Further study is required to clarify this issue.

ALGORITHM 2: Stage 2: regression stage

```
1: pred_subspace retrieved from Algorithm 1
2: opt_goal ← K.optimization_goal
3: ranking ← DNN_output.opt_goal                    ▷ Use prediction of Algorithm 1, line 2
4: ranked_subspace ← pred_subspace.sort(ranking)
5: valid_found ← False, idx ← 0
6: verified_subspace ← ∅
7: while not valid_found & idx < ranked_subspace.size do
8:     adapt_opt ← ranked_subspace[idx]
9:     Analyzer.analyzeAdaptationOptions(adapt_opt)                    ▷ Knowledge
10:    _, qualities ← K.verification_results[idx]
11:    if qualities meet all thresholds and set-point goals then
12:        valid_found ← True
13:    end if
14:    verified_subspace.add(adapt_opt)
15:    idx ← idx + 1
16: end while
17: Select valid adaptation option adapt_opt, if none valid use fall back option
18: unselected ← K.adaptation_options \ verified_subspace
19: explore ← unselected.randomSelect(exploration_rate)
20: Analyzer.analyzeAdaptationOptions(explore)                          ▷ Knowledge
21: input_vectors, qualities ← K.verification_results
22: K.DNN_model.update(input_vectors, qualities)
```

threshold and set-point goals. After iterating through the ranked adaptation (sub)space, a valid option is selected to adapt the system, i.e., an adaptation option that satisfies all the threshold and set-point goals (see line 17). If no valid adaptation option is found according to the goals, then a fall-back option is used, ensuring graceful degradation of the system. Then, lines 18 and 19 use the *exploration_rate* to select a random sample of adaptation options from the options that were not analyzed. These options are then also analyzed in line 20 and the analysis results are stored in the knowledge. Finally, lines 21 and 22 exploit the analysis results to update the deep neural network model. This enables the model to cope with the dynamic behavior of the adaptation space.

6.2.3 Runtime Integration of DLASER+ with MAPE-K. Figure 13 shows how DLASER+ is integrated with a MAPE-K feedback loop.

We follow here the MAPE model and the tasks associated with the different MAPE elements as specified in Reference [72]. When the monitor finishes an update of the knowledge with new runtime data, it triggers the analyzer. The analyzer reads the data that are necessary for the deep neural network to determine the relevant adaptation options. The deep neural network then produces predictions that are written to the knowledge (steps 1 to 4). Next, the analyzer triggers the planner that initiates adaptation space reduction; the ranked adaptation options are then written to the knowledge (steps 5 to 7). Next, the planner reads the ranked options and invokes the model verifier to analyze the options one-by-one and writes the results to the knowledge (steps 8 to 10). When a suitable adaptation is found, the planner invokes the deep neural network to read the verification results and update the learning model accordingly (steps 11 to 13). During these steps, the deep neural network trains the learning model using the most recent verification results that are generated by the model verifier. This online learning in each adaptation cycle helps improve

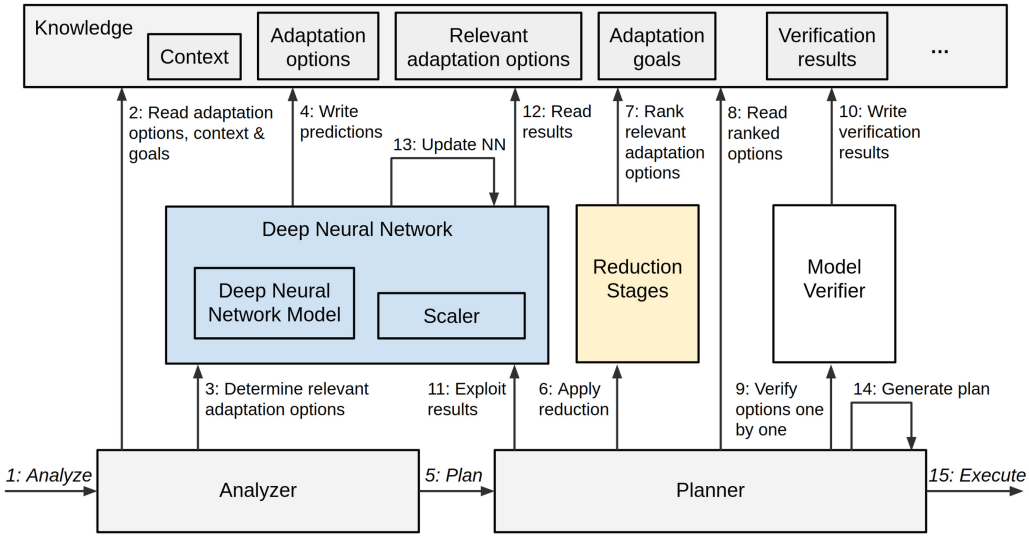


Fig. 13. Runtime integration of DLASER+ with MAPE.

the learner over time to deal with new conditions that the system encounters. Finally, the planner generates a plan for the adaptation option that is selected to adapt the system and triggers the executor to enact the actions of the plan on the managed system (steps 14 and 15).

7 EVALUATION

We start with describing the evaluation setup and specify the various combination of adaptation goals that are evaluated. Then, we explain the offline results we obtained. Finally, we zoom in on the online results for effectiveness and efficiency of DLASeR+. All material used for the evaluation, including all configurations and settings and all evalution results, are available at the DLASeR website.[14]

7.1 Evaluation Setup

We evaluated DLASeR+ on two instances of DeltaIoT, as described in Section 2. For both instances, we used the same type of network settings. The stochastic uncertainty profiles for the traffic load generated by the motes ranged from 0 to 10 messages per mote per cycle, while the network interference along the links varied between −40 dB and +15 dB. The configurations of these profiles are based on field tests. The MAPE-K feedback loop was designed using a network of timed automata. These runtime models were executed by using the ActivFORMS execution engine [36, 78]. The quality models were specified as stochastic timed automata. These models are used to determine the quality estimates for the adaptation options. Section 2 explains an example model for packet loss. We applied runtime statistical model checking using Uppaal-SMC for the verification of adaptation options [17]. The exploration rate was set to 5%. For both instances of DeltaIoT, we considered 275 online adaptation cycles, corresponding with a wall clock time of 77 hours. We used 45 cycles to train the network parameters. The remaining 230 cycles are evaluated as learning cycles.

[14] https://people.cs.kuleuven.be/danny.weyns/software/DLASeR/index.html.

Table 2. Overview of Combinations of Adaptation Goals We Evaluated for Both Instances of DeltaIoT

Reference	Goals DeltaIoTv1	Goals DeltaIoTv2
TTO	**T1**: $PL < 10\%$ **T2**: $LA < 5\%$ **O1**: minimize *EC*	**T1**: $PL < 10\%$ **T2**: $LA < 5\%$ **O1**: minimize *EC*
TTS	**T1**: $PL < 15\%$ **T2**: $LA < 10\%$ **S1**: $EC \in [12.9 \pm 0.1]$	**T1**: $PL < 15\%$ **T2**: $LA < 10\%$ **S1**: $EC \in [67 \pm 0.3]$
TSO	**T1**: $LA < 10\%$ **S1**: $EC \in [12.9 \pm 0.1]$ **O1**: minimize *PL*	**T1**: $LA < 10\%$ **S1**: $EC \in [67 \pm 0.3]$ **O1**: minimize *PL*

The adaptation goals are defined for packet loss (PL), latency (LA), and energy consumption (EC).

To evaluate the effect on the realization of the adaptation goals using DLASeR**+**, we compare the results with a reference approach that analyzes the whole adaptation space without learning. For the evaluation of coverage of relevant adaptation options and the reduction of the adaptation space, we could only compare the results for settings with threshold goals and an optimization goal with the basic DLASeR approach [70] and **ML4EAS (Machine Learning for Efficient Adaptation Space Reduction)** proposed by Quin et al. [58]. The latter approach applies classic machine learning techniques to reduce adaptation spaces for threshold goals.

Table 2 summarizes the combinations of adaptation goals we evaluated for both instances of DeltaIoT: *TTO* (2 Threshold goals and 1 Optimization goal), *TTS* (2 Threshold goals and 1 Set-point goal), and *TSO* (1 Threshold, 1 Set-point, and 1 Optimization goal).[15]

For the implementation of DLASeR**+**, we used the scalers from *scikit-learn* [56] and the neural networks from *Keras* and *Tensorflow* [1]. The simulated IoT networks are executed on an i7-3770 CPU @ 3.40 GHz with 12 GB RAM; the deep learning models are trained and maintained on an i7-3770 CPU @ 3.40 GHz with 16 GB RAM.

We start with presenting the results of the offline stage of the DLASeR**+** learning pipeline. Then, we present the results for effectiveness and efficiency of the online stage of the learning pipeline for each combination of goals.

7.2 Results Offline Settings

As explained in Section 6.1, DLASeR**+** uses grid search for configuring and tuning the deep neural network. We performed grid search on 30 sequential adaptation cycles.[16] Table 3 shows the best parameters of the network for each of the three combinations of adaptation goals. Each row in the table corresponds to one grid search process. In total, six grid search processes were completed; one for each of the three combinations of adaptation goals and this for the two instances of DeltaIoT.

Given that DLASeR**+** comprises a single integrated neural network architecture with multiple classification and regression heads, we use the validation loss to select the best hyper-parameter configuration. The validation loss captures how good the predictions are of the neural network

[15]For TTO setting, ML4EAS applies classifiers to reduce the adaptation space for the two threshold goals and then searches within the reduced adaptation space to find the best adaptation option for the optimization goal.

[16]Concretely, we used the data of all adaptation options with their verification results over 30 adaptation cycles, i.e., 216 adaptation options with verification results per cycle for DeltaIoTv1 and 4,096 adaptation options with verification results per cycle for DeltaIoTv2. Figures 5 and 6 illustrate the performance of different adaptation options (for DeltaIoTv2).

Table 3. Best Grid Search Results for DLASeR**+** on *TTO*, *TTS*, and *TSO* Goal Combinations (See Table 2)

Problem	Goals	Hyper parameters						
		Scaler	Batch size	LR	Optimizer	Core layers	Class. layers	Regr. layers
DeltaIoTv1	**TTO**	Standard	64	5e-3	Adam	[50, 25, 15]	[20, 10, 5]	[40, 20, 10, 5]
DeltaIoTv2	**TTO**	Standard	512	5e-3	Adam	[150, 120, 100, 50, 25]	[40, 20, 10, 5]	[30, 40, 50, 40, 15, 5]
DeltaIoTv1	**TTS**	MaxAbsScaler	64	5e-3	Adam	[50, 25, 15]	[40, 20, 10, 5]	/
DeltaIoTv2	**TTS**	MaxAbsScaler	512	5e-3	Adam	[200, 100, 50, 25]	[40, 20, 10, 5]	/
DeltaIoTv1	**TSO**	StandardScaler	16	5e-3	Adam	[50, 80, 35, 15]	[20, 10, 15]	[40, 20, 10, 5]
DeltaIoTv2	**TSO**	StandardScaler	512	2e-3	RMSprop	[150, 120, 100, 50, 25]	[40, 20, 10, 5]	[30, 40, 50, 40, 15, 5]

LR refers to learning rate. The values between brackets for the different layers represent the number of neurons.

Table 4. Results for the Coverage of Relevant Adaptation Options for the Different Evaluation Settings

Problem	**Setting**	**Method**	**F1 threshold**	**F1 set-point**	**Spearman's rho optimization**
DeltaIoTv1	TTO	DLASeR**+**	86.92%	/	83.43%
		DLASeR	83.92%	/	43.34%
		ML4EAS	75.02%	/	/
	TTS	DLASeR**+**	57.12%	33.77%	/
	TSO	DLASeR**+**	75.60%	43.75%	96.49%
DeltaIoTv2	TTO	DLASeR**+**	63.16%	/	21.53%
		DLASeR	62.35%	/	5.81%
		ML4EAS	89.87%	/	/
	TTS	DLASeR**+**	60.48%	35.14%	/
	TSO	DLASeR**+**	64.19%	35.20%	95.99%

model compared to the true data (of the validation set). Here, the validation loss corresponds to the sum of the losses for each head. In total, grid search evaluated 4,120 configurations for DeltaIoTv1 (1,728 for TTO and TSO goals and 768 for TTS goals) and 3,456 configurations for DeltaIoTv2 (1,296 for TTO and TSO goals and 864 for TTS goals). The average time that was required for the offline training of the deep neural network for a configuration was on 25 s for DeltaIoTv1 and 90 s for DeltaIoTv2. We observe that the loss for the configuration with TTO goals is significantly lower for DeltaIoTv1. Overall, the loss for DeltaIotv2 is somewhat lower. Yet, as we will show in the following subsection, the differences do not lead to significantly inferior results.

We used the configurations with the best results for the different instances of DLASeR**+** shown in Table 3 to perform adaptation space reduction for the three configurations with different adaptation goals (see Table 2) for both versions of DeltaIoT.

7.3 Results Online Setting

We present the results for the different settings shown in Table 2, starting with the metrics for effectiveness, followed by the metrics for efficiency. The results allow us to to answer the different aspects of the research question in a structured manner.

7.3.1 Effectiveness - Coverage of Relevant Adaptation Options. To assess the first aspect of the effectiveness of DLASeR**+**, we look at the F1-score for the threshold and set-point goals, and Spearman's rho for the optimization goal. Table 4 presents the results.

Note that the result of DLASeR**+** for the classification task is a continuous value that needs to be rounded to an integer class number. Hence, the classification error can be computed in two ways. The first method measures the error before rounding, e.g., if the real class of an input data is "1" and the predicted class is "0.49," then the error will be "$1 - 0.49 = 0.51$." The second method

measures the error after rounding, e.g., if the real class of an input data is "1" and the predicted class is "0.49," then the error will be "1 − 0 = 1," as "0.49" has been fixed to "0." Depending on which method is selected for computing the classification error, the F1-score can have different values. We use the second method for calculating the classification error, enabling a comparison of the results of DLASeR**+** with ML4EAS, where the output is an integer class number instead of a continuous value.

For DeltaIoTv1, we notice an F1-score for the threshold goals of 86.92% for the setting TTO, 57.12% for TTS, and 75.60% for TSO (with 100% being perfect precision and recall). The F1-scores for the set-point goal for TTS and TSO are, respectively, 33.77% and 43.75%. The differences can be explained by the constraints imposed by the types of goals combined in the different settings, with setpoint goals being most constraining, followed by threshold goals and then optimization goals. Consequently, the F1-score is highest for TTO with two threshold goals, followed by TSO with one threshold goal and one setpoint goal, and finally TTS that combines two threshold goals with a setpoint goal. For Spearman's rho, we observe a difference between the settings with an optimization goal, with 83.43% and 96.49% for TTO and TSO, respectively. This shows that regression is more difficult for energy consumption (TTO) compared to packet loss (TSO).

For DeltaIoTv2, we observe an F1-score for the threshold goals of 63.16% for the TTO setting, 60.48% for TTS, and 64.19% for TSO. The F1-score for the two set-point goals, are around 35%. These results for F1-score are slightly lower compared to the results for DeltaIoTv1, indicating that the setting is more challenging for the learners. For Spearman's rho, we observe values of 21.53% and 95.99% for TTO and TSO, respectively. The weak score for the setting with TTO may point to a negative effect on the optimization goal caused by the training of the core layers for the two threshold goals. In particular, this may indicate that the knowledge shared in the core layers for the threshold goals and the optimization goal is limited. While this may seem problematic, the evaluation will show that lower Spearman's rho values do not necessarily imply inferior results.

Overall, we obtained acceptable to excellent results for the F1-score and Spearman's rho (with one exception). Compared to the results obtained for the initial version of DLASeR [70] and ML4EAS [58], we observe similar results for the TTO setting (the results with ML4EAS were somewhat better for DeltaIoTv2, but somewhat worse for DeltaIoTv1). However, it is important to emphasize that these approaches do not consider the other combinations of adaptation goals.

7.3.2 Effectiveness - Reduction Adaptation Space. Table 5 presents the results for adaptation space reduction obtained with the different approaches for the different configurations. We observe that the highest **average adaptation space reduction (AASR)** with DLASeR**+** is achieved for the TTO setting and this for both IoT instances. The results are slightly better compared to the initial DLASeR but slightly worse compared to ML4EAS. The AASR is lower for the other combinations of goals, which indicates that reducing the adaptation space with DLASeR**+** for settings with a setpoint goal is more challenging. However, for the **average analysis effort reduction (AAER)**, we notice that only a limited number of adaptation options need to be analyzed from the selected subspace before a valid option is found that complies with the classification goals. This is particularly the case for TTS and TSO settings that include a setpoint goal. This means that the adaptation space reductions captured by AASR is of high quality.

Besides AASR and AAER, we also measured the fraction of the adaptation space that was analyzed of the total adaptation space, which combines *AASR* and *AAER*,[17] defined as:

$$Total\ Reduction\ = \left(1 - \frac{analyzed}{total}\right) \times 100. \tag{11}$$

[17] *The definition of total reduction can be rewritten as: Total Reduction* $= 100 - (100 - AASR) \times (1 - \frac{AAER}{100})$.

Table 5. Adaptation Space Reductions on the Three Configurations of Table 2

Problem	Setting	Method	AASR	AAER	Total Reduction
DeltaIoTv1	TTO	DLASeR+	56.77%	90.11%	95.72%
		DLASeR	54.84%	88.88%	94.98%
		ML4EAS	62.61%	0.00%	62.61%
	TTS	DLASeR+	42.88%	98.54%	99.16%
	TSO	DLASeR+	38.27%	94.18%	96.41%
DeltaIoTv2	TTO	DLASeR+	89.03%	95.57%	99.52%
		DLASeR	84.55%	72.41%	95.73%
		ML4EAS	92.86%	0.00%	92.86%
	TTS	DLASeR+	41.79%	99.90%	99.94%
	TSO	DLASeR+	51.00%	99.95%	99.98%

For DeltaIoTv1, we measured a total reduction of 95.72%, 99.16%, and 96.41% for TTO, TTS, and TSO, respectively, while the total reduction for DeltaIoTv2 was 99.52%, 99.94%, and 99.98%. These excellent results show that deep learning with DLASeR+ is particularly effective in reducing adaptation spaces, i.e., the total reduction is near to the optimum of what can be achieved.

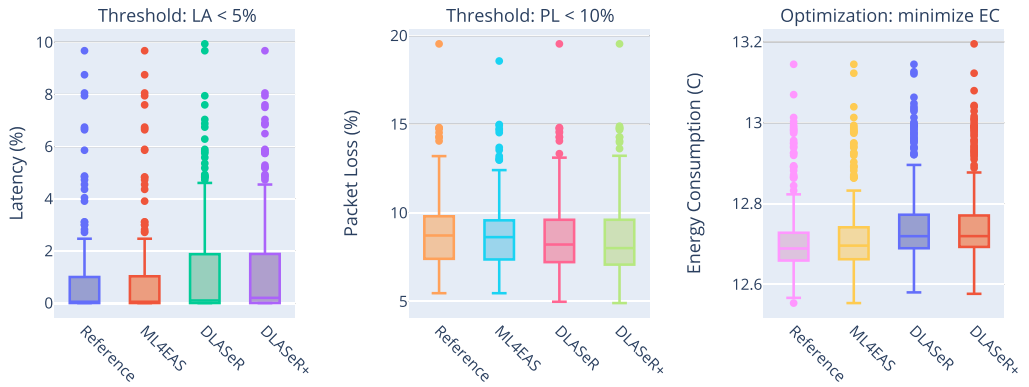
7.3.3 Effectiveness - Effect on Realization of Adaptation Goals. To evaluate the effectiveness, we compare the median values of the quality properties that correspond to the adaptation goals over 230 learning cycles (i.e., representing about three days of operation of the IoT networks; see evaluation setup). Note that a threshold goal is satisfied if the median of the values over 230 cycles satisfies the goal. This does not necessarily mean that the system satisfies the goal in all cycles.

The boxplots of Figures 14–16 show the results for the quality properties of the adaptation goals with DLASeR+ and other approaches. It is important to note that the reference approach exhaustively analyzes the whole adaptation space. This is the ideal case, but practically not always feasible (as in DeltaIoTv2) due to time constraints on the time available to perform adaptation.

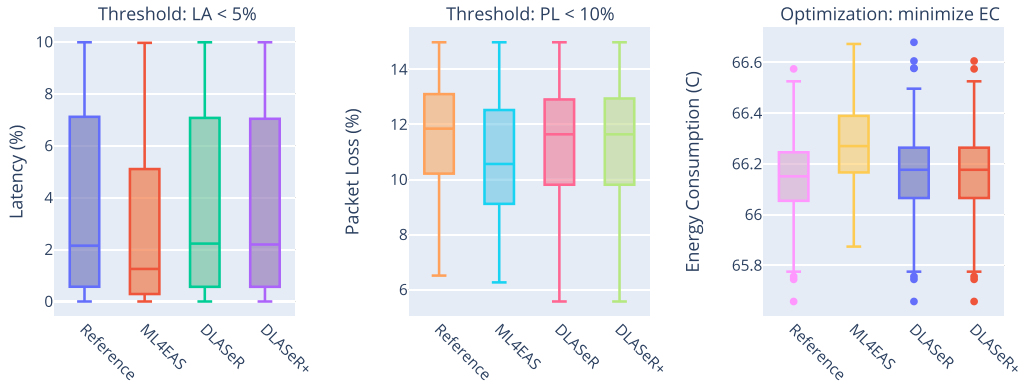
For the setting with TTO goals (see Figure 14), the results for packet loss and latency (threshold goals) are similar for all approaches. We observe that DLASeR+ always satisfies the goals (i.e., all median values are below the thresholds). For some adaptation cycles, no configurations are available that satisfy the threshold goals (as shown by the reference approach that exhaustively searches through the complete adaptation space). For energy consumption (optimization goal), the results for DLASeR+ are slightly higher compared to the reference approach, i.e., an increase of 0.03 C (0.24%) for DeltaIoTv1 (median of 12.69 C for the reference approach; 12.72 C for DLASeR+), and 0.03 C (0.05%) for DeltaIoTv2 (66.15 C for the reference approach; 66.18 C for DLASeR+).

For the setting with TTS goals (see Figure 15), we observe similar results for DLASeR+ and the reference approach. The threshold goals for packet loss and latency are again always satisfied by DLASeR+. The set-point goal of energy consumption is also satisfied for both instances of DeltaIoT (DLASeR+ shows slightly more variability around the set-point).

Finally, for the setting with TSO goals (see Figure 16), the results show that the threshold goal for latency is always satisfied with DLASeR+, and the same applies for the set-point goal for energy consumption. As for the optimization goal, the packet loss of DLASeR+ is slightly higher compared to the reference approach, respectively, 0.57% for DeltaIoTv1 (median of 6.33% for the reference approach versus 6.90% for DLASeR+) and 0.62% for DeltaIoTv2 (6.33% for the reference approach versus 6.95% for DLASeR+).



(a) DeltaIoTv1



(b) DeltaIoTv2

Fig. 14. Effect on the realization of the adaptation goals for the TTO setting.

In summary, the results for the quality properties using DLASeR+ and the reference approach are similar. For settings with two threshold goals and one optimization goal, the results are similar as for two state-of-the-art approaches, the initial DLASeR [70] and ML4EAS [58]. However, contrary to these state-of-the-art approaches, DLASeR+ realizes adaptation space reduction for combinations of threshold, set-point goals, and optimization goals with a small to negligible effect on the qualities compared to the reference approach.

7.3.4 Efficiency - Learning Time. Table 6 presents the results for the analysis time, learning time, and overall time reduction of DLASeR+ compared to the reference approach. As we can see, the major part of the time is spent on analysis, i.e., the time used for the verification of adaptation options. This is in contrast with the learning time[18]; on average, 16.45% of the time is spent on learning for DeltaIoTv1 (total time for verification of the three settings is 4.45 s versus 0.894 s for learning) and 1.56% for DeltaIoTv2 (102.95 s in total for verification of the three settings versus 1.63 s for learning). We observe high numbers for time reduction, on average 92.80% for the three settings of DeltaIoTv1 and 94.84% for the settings of DeltaIoTv2. For TTO settings, DLASeR+ realizes better time reductions than the other approaches. Note that the optimal time reduction is 95%,

[18]Recall that the learning time is the sum of the time used for online prediction and online training, cf. Table 1.

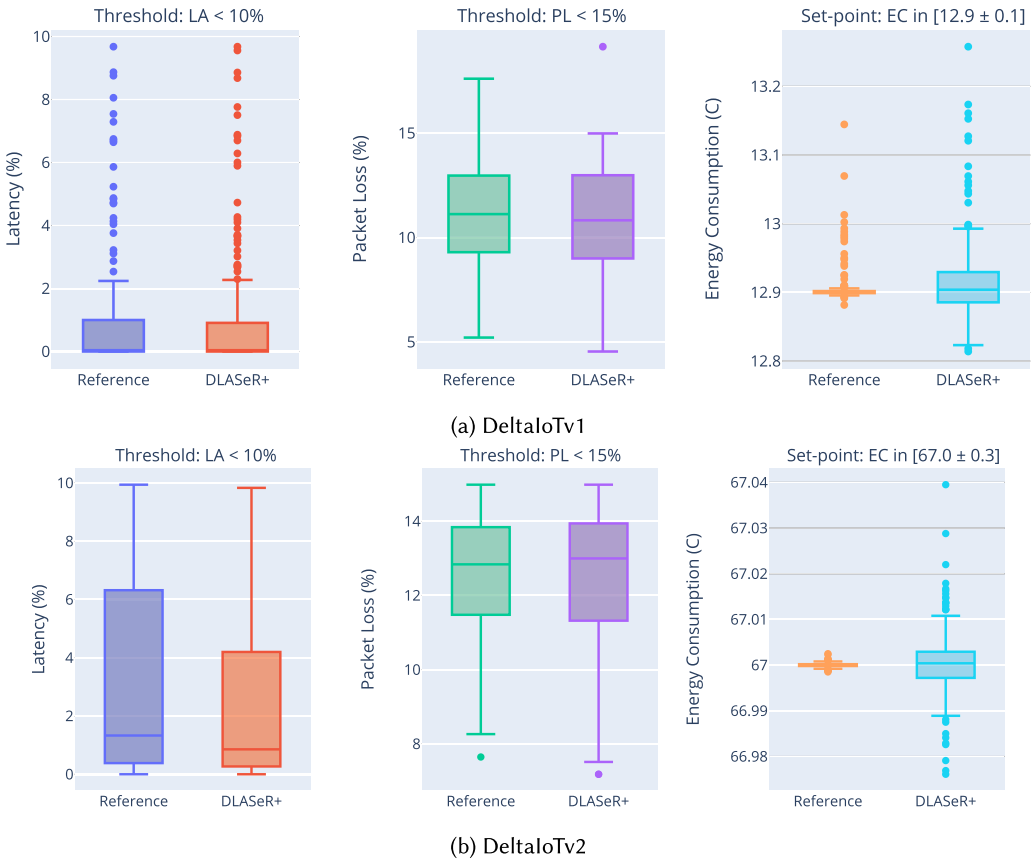


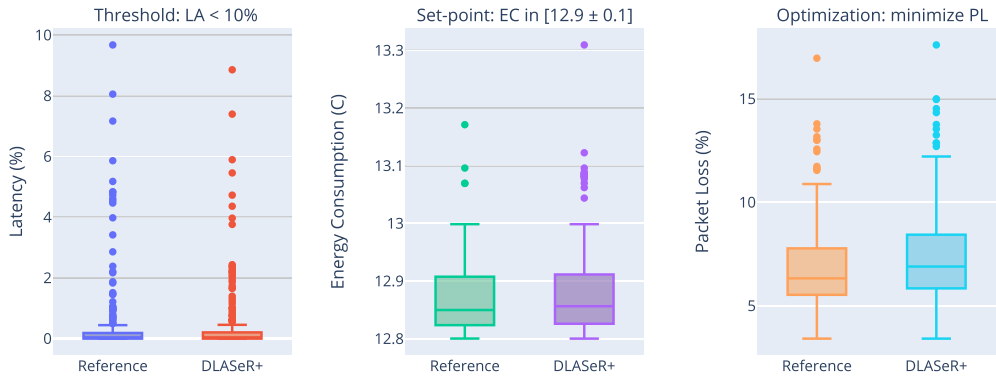
Fig. 15. Effect on the realization of the adaptation goals for the TTS setting.

since we use an exploration rate of 5%, i.e., in each adaptation cycle, 5% of the adaptation space is explored.

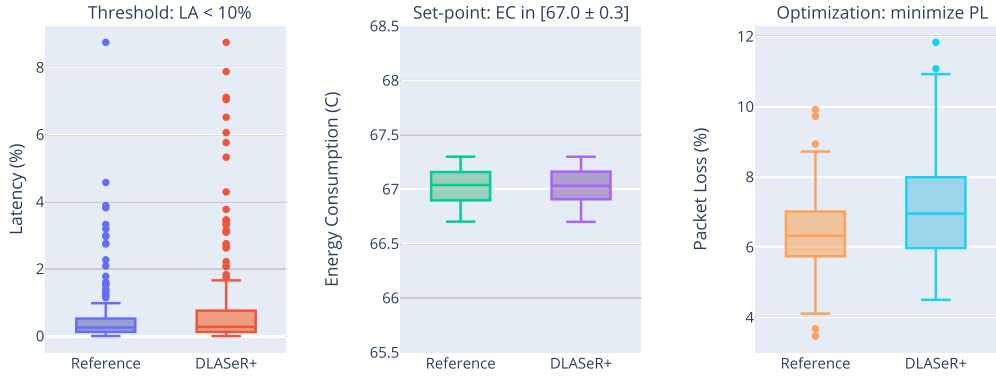
In sum, the learning time of DLASeR**+** is only a fraction of the total time used for analysis. DLASeR**+** realizes an impressive reduction of the time required for analysis and decision-making compared to the reference approach.

7.3.5 Efficiency - Scalability. To evaluate the scalability of DLASeR**+**, we discuss the difference for the metrics when scaling up the evaluation setting from DeltaIoTv1 with 216 adaptation options to DeltaIoTv2 with 4,096 adaptation options; an increase of adaptation space with a factor around 20.

For the *coverage of relevant adaptation options* (see Table 4), we observe a decrease in F1-score for the threshold goals of 23.76% for the TTO setting (86.92% for DeltaIoTv1 versus 63.16% for DeltaIoTv2) and a decrease of 11.41% for the TSO setting (75.60% versus 64.18%). For the TTS settings, we notice a small increase of the F1-score for the threshold goals of 3.36% (57.12% versus 60.48%). For set-point goals, we observe a small increase in F1-score of 1.37% for the setting TTS (33.77% versus 35.14%) and a small decrease of 8.55% for the setting TSO (43.75% versus 35.20%). The results show that DeltaIoTv2 is more challenging, but DLASeR**+** scales well for threshold goals. For the optimization goal, we notice a large decrease for Spearman's rho with 61.9% for the setting



(a) DeltaIoTv1



(b) DeltaIoTv2

Fig. 16. Effect on the realization of the adaptation goals for the TSO setting.

Table 6. Verification and Learning Time for DLASeR+

Problem	Setting	Method	Verification Time	Learning Time	Time Reduction
DeltaIoTv1	TTO	DLASeR+	1.95 s	0.004 s	90.72%
		DLASeR	2.13 s	0.003 s	89.87%
		ML4EAS	6.99 s	0.002 s	66.80%
	TTS	DLASeR+	1.11 s	0.45 s	94.73%
	TSO	DLASeR+	1.48 s	0.44 s	92.96%
DeltaIoTv2	TTO	DLASeR+	36.44 s	0.45 s	94.52%
		DLASeR	61.21 s	1.62 s	90.79%
		ML4EAS	68.44 s	0.04 s	89.71%
	TTS	DLASeR+	33.27 s	0.63 s	95.00%
	TSO	DLASeR+	33.24 s	0.55 s	95.00%

Time reduction compares the total time used by DLASeR+ compared to the time used by the reference approach that verifies the complete adaptation space.

with TTO goals (from 83.43% for DeltaIoTv1 to 21.53% for DeltaIoTv2) and a small decrease of 0.5% for the setting with TSO goals (from 96.49% to 95.99%). These results suggest that the optimization goal in TTO (energy consumption) might be significantly harder to predict.

For the *average adaptation space reduction* (see Table 5), we measured a substantial increase for two configurations and a small decrease for one configuration when scaling up from DeltaIoTv1 to DeltaIoTv2. On average, we measured an increase of 14.63% over the three configurations (from 56.77% to 89.03% for TTO and from 38.27% to 51.00% for TSO; from 42.88% to 41.79% for TTS). The *average analysis effort reduction* also improved for DeltaIoTv2, with an average of 4.20% (from 90.11% to 95.57% for TTO, from 98.54% to 99.90% for TTS, and from 94.18% to 99.95% for TSO. This resulted in an average increase of 2.72% for the *total reduction* (from 95.72% to 99.52% for TTO, from 99.16% to 99.94% for TTS, and from 96.41% to 99.98% for TSO).

For the *effect on realizing the adaptation goals* (see Figures 14–16), we observe that DLASeR**+** realizes the threshold and set-point goals for all configurations of both DeltaIoTv1 and DeltaIoTv2. For the optimization goal compared to the reference approach, we observe a slight increase of energy consumption for the TTO setting with DLASeR**+** from 0.24% extra with DeltaIoTv1 (medians 12.69 C versus 12.72 C) to 0.05% extra with DeltaIoTv2 (medians 66.15 C versus 66.18 C). Similarly, we observe a small increase of packet loss for TSO with DLASeR**+** from 0.57% extra with DeltaIoTv1 (medians 6.33% versus 6.90%) to 0.62% extra with DeltaIoTv2 (medians 6.33% versus 6.95%).

Finally, the results for the *learning time* (see Table 6) show that relative part of the time required for learning compared to the time required for analysis decreases from 16.45% for DeltaIoTv1 to 1.56% for DeltaIoTv2, while the *total time reduction* improves from 92.80% to 94.84% (the numbers are averages over the three settings TTO, TTS, and TSO). The results show that with increasing scale, the total time reduction improves substantially and gets close to optimum for DeltaIoTv2.

In summary, while some of the indicators for the coverage of relevant adaptation options are slightly inferior for configurations with larger adaptation spaces, the other metrics show no negative effects on the effectiveness and efficiency of DLASeR**+** when the size of the adaptation space is increased (in the evaluating setting with a factor of about 20).

7.4 Threats to Validity

The evaluation results show that DLASeR**+** is an effective and efficient approach to reduce large adaptation spaces. However, the evaluation of DLASeR**+** is subject to a number of validity threats.

External validity. Since we evaluated the approach in only one domain with a particular adaptation spaces and adaptation goals, we cannot generalize the conclusions, including the configuration of DLASeR**+** and the effectiveness and efficiency of adaptation space reduction. We mitigated this threat to some extent by applying DLASeR**+** to two IoT networks that differ in their topology and size of adaptation space. Nevertheless, more extensive evaluation is required in different domains to strengthen the validity of the results. Furthermore, the reduction stages of the unified architecture of DLASeR**+** presented in this article target only a single optimization goal. Hence, the approach is not directly applicable to systems with multi-objective optimization goals. Additional research will be required to extend DLASeR**+** for systems with such types of goals. We also tested DLASeR**+** for scenarios with up to 4,000 adaptation options in one domain. Further research is required to study and evaluate other types of systems with much larger adaptation spaces.

Internal validity. We evaluated the effectiveness and efficiency of DLASeR**+** using different metrics. It might be possible that the specifics of the evaluation settings of the applications that we used—in particular, the topology of the network, the uncertainties, and the choices for the specific goals that we considered—may have an effect on the complexity of the problem of adaptation space

reduction. To mitigate this threat to some extent, we applied DLASeR**+** to simulated settings of real IoT deployments that were developed in close collaboration with an industry partner in IoT.

Reliability. For practical reasons, we performed the evaluation of DLASeR**+** in simulation. The uncertainties used in this simulation are based on stochastic models. This may cause a threat that the results may be different if the study would be repeated. We minimized this threat in three ways: (i) the profiles are designed based on field tests, (ii) we evaluated DLASeR**+** over a period of several days, and (iii) the complete package we used for evaluation is available to replicate the study.[19]

8 RELATED WORK

Over the past years, we can observe an increasing interest in using machine learning and related techniques to support self-adaptation [29]. Three examples of initial work are Richert W. et al. [63], who apply reinforcement learning to robots that learn to select tasks that balance their own benefit with the needs of other robots; Sykes et al. [67], who use a probabilistic rule learning approach to update the environment models of a self-adaptive system at runtime relying on execution traces; and Bencomo et al. [5], who use dynamic decision networks to model and support decision-making in self-adaptive systems explicitly taking into account uncertainty.

In this section, we focus on the use of machine learning and other related techniques that are used for adaptation space reduction and efficient decision-making of systems with complex adaptation spaces. We have structured the related work in four groups based on their main focus: (1) learning used for the reduction of adaptation spaces, (2) learning used for efficient decision-making, (3) search-based techniques to efficiently explore large adaptation spaces, and finally (4) approaches for efficient verification. For each group, we present a representative selection of related work.

8.1 Learning Used for Adaptation Space Reduction

Elkhodary et al. [22] propose the FUSION framework, which learns the impact of adaptation decisions on the system's goals. In particular, FUSION utilizes M5 decision trees to learn the utility functions that are associated with the qualities of the system. The results show a significant improvement in analysis. Whereas DLASeR**+** targets the adaptation space, FUSION targets the feature selection space, focusing on proactive latency-aware adaptations relying on a separate model for each utility.

Chen et al. [11] study feature selection and show that different learning algorithms perform significantly different, depending on the types of quality of service attributes considered the way they fluctuate. The work is centered on an adaptive multi-learners technique that dynamically selects the best learning algorithms at runtime. Similar to our work, the authors focus on efficiency and effectiveness, but the scope of that work is on the features instead of adaptation options.

Quin et al. [58] apply classical machine learning techniques, in particular, classification and regression, to reduce large adaptation spaces. These techniques require domain expertise to perform feature engineering, which is not required in DLASeR**+** (which only requires model selection). That work also only considers threshold goals based on linear learning models. In contrast, our work considers threshold, optimization, and set-point goals based on non-linear deep learning models.

Jamshidi et al. [37] present an approach that learns a set of Pareto optimal configurations offline that are then used during operation to generate adaptation plans. The approach reduces adaptation spaces, while the system can still apply model checking with PRISM [46] at runtime to quantitatively reason about adaptation decisions. Compared to that work, DLASeR**+** is more versatile by reducing the adaptation space at runtime in a dynamic and learnable fashion.

[19]DLASeR website: https://people.cs.kuleuven.be/danny.weyns/software/DLASeR/index.html.

Metzger et al. [49] apply online learning to explore the adaptation space of self-adaptive systems using feature models. The authors demonstrate a speedup in convergence of the online learning process. The approach is centered on the adaptation of rules, whereas DLASeR**+** is centered on model-based self-adaptation. Furthermore, the work also looks at the evolution of the adaptation space, while DLASeR**+** only considers dynamics in the adaptation space (not its evolution).

Camara et al. [9] use reinforcement learning to select an adaptation pattern relying on two long-short-term memory deep learning models. Similar to our work, these authors demonstrate the benefits of integrating machine learning with runtime verification. However, the focus differs in the type of goals considered (they only consider threshold goals) and the type of verification used (they use runtime quantitative verification); that work also does not consider scalability.

Stevens et al. [66] present Thallium, which exploits a combination of automated formal modeling techniques to significantly reduce the number of states that need to be considered with each adaptation decision. Thallium addresses the adaptation state explosion by applying utility bounds analysis. The (current) solution operates on a Markov decision process, which represents the structure of the system itself, independent of the stochastic environment. The authors suggest future work in combining learning-based approaches employed on the reduced adaptation space from Thallium.

8.2 Learning Used for Efficient Decision-making

Kim et al. [41] present a reinforcement learning-based approach that enables a software system to improve its behavior by learning the results of its behavior and dynamically changing its plans under environmental changes. Compared to DLASeR**+**, the focus is on effective adaptation decision-making, without considering guarantees or the scalability of the proposed approach.

Anaya et al. [3] present a framework for proactive adaptation that uses predictive models and historical information to model the environment. These predictions are then fed to a reasoning engine to improve the decision-making process. The authors show that their approach outperforms a typical reactive system by evaluating different prediction models (classifiers). Whereas they focus on leveraging proactive techniques to make decision-making more effective, our work focuses on both the efficiency and effectiveness of adaptation space reduction to improve the decision-making.

Qian et al. [57] study goal-driven self-adaptation centered on case-based reasoning for storing and retrieving adaptation rules. Depending on requirements violations and context changes, similar cases are used, and if they are not available, then goal reasoning is applied. This way, the approach realizes more precise adaptation decisions. The evaluation is done only for threshold goals, and the authors only provide some hints to scale up their solutions to large-sized systems. In our work, we explicitly evaluate the effect of scaling up the adaptation space.

Nguyen Ho et al. [33] rely on model-based reinforcement learning to improve system performance. By utilizing engineering knowledge, the system maintains a model of interaction with its environment and predicts the consequence of its action. DLASeR**+** relies on different learning techniques. Furthermore, we study the effect of scalability of the adaptation space, which is not done in that paper.

8.3 Search-based Techniques to Explore Large Adaptation Spaces

Cheng et al. [13] argue for three overarching techniques that are essential to address uncertainty in software systems: model-based development, assurance, and dynamic adaptation. In relation to the work presented in this article, the authors argue for the application of search-based software engineering techniques to model-based development, in particular, the use of evolutionary algorithms to support an adaptive system to self-reconfigure safely. In Reference [61], the authors propose Hermes, a genetic algorithmic approach that adapts the system efficiently in time.

In Reference [62], the authors propose Plato, an approach that maps data monitored from the system or the environment into genetic schemes and evolves the system by leveraging genetic algorithms. The main aim of these approaches is ensuring safety under uncertainty in an efficient manner. In contrast, DLASeR**+** is conceptually different, relying on deep learning to explicitly reduce large adaptation spaces, providing explicit support for different types of adaptation goals.

Le Goues et al. [47] propose GenProg, an automated method for repairing defects in legacy programs. GenProg relies on genetic programming to evolve a program variant that retains required functionality but is not susceptible to a given defect, using existing test suites to encode both the defect and required functionality. The focus of this work is on efficiently producing evolved programs that repair a defect without introducing substantial degradation in functionality. The focus of DLASeR**+**, however, is on reducing large adaptation spaces at architectural level, aiming to enhance the efficiency of the decision-making of self-adaptive systems that need to deal with different types of quality properties.

Nair et al. [53] present FLASH, which aims at efficiently finding good configurations of a software system. FLASH sequentially explores the configuration space by reflecting on the configurations evaluated so far to determine the next best configuration to explore. FLASH can solve both single-objective and multi-objective optimization problems. Whereas FLASH assumes that the system is stationary, DLASeR**+** uses incremental learning to stay up to date during operation; i.e., DLASeR**+** deals with dynamics in the environment at runtime.

Kinneer et al. [42] propose a planner based on genetic programming that reuses existing plans. Their approach uses stochastic search to deal with unexpected adaptation strategies, specifically by reusing or building upon prior knowledge. Their genetic programming planner is able to handle very large search spaces. Similar to DLASeR**+**, the evaluation of this work considers efficiency and effectiveness. However, the technique used is different, focusing on planning, and that work put particular emphasis on reuse.

Chen et al. [12] present FEMOSAA, a framework that leverages a feature model and a multi-objective evolutionary algorithm to optimize the decision-making of adaptation at runtime. The authors show that FEMOSAA produces statistically better and more balanced results for tradeoff with reasonable overhead compared to other search-based techniques. Compared to DLASeR**+**, the authors use a different analysis technique and rely on feature models. The latter implies that the approach relies on domain engineers to construct a feature model for the self-adaptive system.

In Reference [16], Coker et al. use genetic programming planning and combine this with probabilistic model checking to determine the fitness of plans for a set of quality properties. The proposed search-based approach provides an integrated solution for guiding the decision-making of a self-adaptive system. This approach requires a well-defined objective function. In contrast, DLASeR**+** focuses on the reduction of the adaptation space for different types of adaptation goals. With DLASeR**+**, different types of decision-making mechanisms can be combined.

Pascual et al. [55] apply a genetic algorithm to generate automatically at runtime configurations for adapting a system together with reconfiguration plans. The generated configurations are optimal in terms of functionality, taking into account the available resources (e.g., battery). Concretely, the configurations are defined as variations of the application's software architecture based on a so-called feature model. In contrast, DLASeR**+** targets the reduction of large adaptation spaces targeting quality properties of the system that are formulated as adaptation goals.

8.4 Approaches for Efficient Verification

Filieri et al. [23] present a mathematical framework for efficient runtime probabilistic model checking. Before deployment, a set of symbolic expressions that represent satisfaction of the requirements is pre-computed. At runtime, the verification step simply evaluates the formulae by

replacing the variables with the real values gathered by monitoring the system. By shifting the cost of model analysis partially to design time, the approach enables more efficient verification at runtime. In later work [24], the authors elaborate on this and explain how the mathematical framework supports reasoning about the effects of changes and can drive effective adaptation strategies. Whereas DLASeR**+** focuses on reducing the set of adaptation options during operation, their work focuses on efficient runtime verification by offloading work before system deployment.

Gerasimou et al. [26] propose three techniques to speed up runtime quantitative verification, namely, caching, lookahead, and nearly optimal reconfiguration. The authors evaluate several combinations of the techniques on various scenarios of self-adaptive systems. The focus of this work is different from DLASeR**+**, but the proposed techniques are complementary and can perfectly be integrated in our work.

Moreno et al. [51] present an approach for proactive latency-aware adaptation that relies on stochastic dynamic programming to enable more efficient decision-making. Experimental results show that this approach is close to an order of magnitude faster than runtime probabilistic model checking to make adaptation decisions, while preserving the same effectiveness. Whereas our approach focuses on reducing the set of adaptation options to improve analysis, their work focuses on fast verification; here, too, a system may benefit from a combination of both approaches.

Goldsby et al. [31] and Zhang et al. [83] present AMOEBA-RT, a runtime approach that provides assurance that dynamically adaptive systems satisfy their requirements. In AMOEBA-RT, an adaptive program is instrumented with aspects that non-invasively collect state of the system that can then be checked against a set of adaptation properties specified in A-LTL, an extended linear temporal logic. At runtime, the instrumented code sends the collected state information to a runtime model checking server that determines whether the state of the adaptive program satisfies the adaptation properties. The focus of this work is on assuring properties using runtime model checking. In contrast, DLASeR**+** focuses on adaptation space reduction. AMOEBA-RT can be used in tandem with DLASeR**+** to enhance the efficiency of the decision-making process.

Junges et al. [39] present a runtime monitoring approach for partially observable systems with non-deterministic and probabilistic dynamics. The approach is based on traces of observations on models that combine non-determinism and probabilities. The authors propose a technique called forward filtering to estimate the possible system states in partially observable settings along with a pruning strategy to enhance its efficiency. Based on empirical results, the authors propose a tractable algorithm based on model checking conditional reachability probabilities as a more tractable alternative. In contrast, DLASeR**+** focuses on the reduction of large adaptation spaces of self-adaptive systems that are subject to uncertainties that can be expressed as parameters of runtime models. Yet, DLASeR**+** can be combined with the proposed approach to enhance the performance of decision-making in self-adaptive systems.

9 CONCLUSIONS

In this article, we studied the research question: "How to reduce large adaptation spaces and rank adaptation options effectively and efficiently for self-adaptive systems with threshold, optimization, and set-point goals?" To answer this question, we presented DLASeR**+**. DLASeR**+** relies on an integrated deep neural network architecture that shares knowledge of different adaptation goals. The approach is flexible, as the core layers can be easily extended with goal-specific heads. The evaluation shows that DLASeR**+** is an effective and efficient approach to reduce large adaptation spaces, including for settings with large adaptation spaces. The approach realizes the threshold and set-point goals for all the configurations we tested on the DeltaIoT artifact. Compared to the theoretical optimal, we observe only a small tradeoff for the quality property of the optimization goal. Yet, this is a small cost for the dramatic improvement of adaptation time.

We are currently applying DLASeR**+** to service-based systems, which will provide us insights in the effectiveness of the approach beyond the domain of IoT. For these systems, we are studying the reduction of adaptation spaces with sizes far beyond the adaptation spaces used in the evaluation of this article, posing more challenging learning problems to DLASeR**+**. We also plan to extend DLASeR**+** for multi-objective optimization goals. Beyond DLASeR**+** and learning-based approaches, we also plan to compare the approach with conceptually different approaches for improving the analysis of large adaptation spaces (as discussed in related work) and perform a tradeoff analysis. In the mid term, we plan to look into support for dynamically adding and removing adaptation goals. We also plan to explore the use of machine learning in support of self-adaptation in decentralized settings [59]. In the long term, we aim at investigating how we can define bounds on the guarantees that can be achieved when combining formal analysis techniques, in particular, runtime statistical model checking, with machine learning; a starting point is Reference [30].

REFERENCES

[1] Martin Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[2] Gul Agha and Kalm Palmskog. 2018. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.* 28, 1 (Jan. 2018). DOI: https://doi.org/10.1145/3158668

[3] Ivan Dario Paez Anaya, Viliam Simko, Johann Bourcier, Noël Plouzeau, and Jean-Marc Jézéquel. 2014. A prediction-driven adaptation approach for self-adaptive sensor networks. In *9th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 145–154.

[4] Arjun P. Athreya, Bruce DeBruhl, and Patrick Tague. 2013. Designing for self-configuration and self-adaptation in the Internet of Things. In *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 585–592.

[5] Nelly Bencomo, Amel Belaggoun, and Valery Issarny. 2013. Bayesian artificial intelligence for tackling uncertainty in self-adaptive systems: The case of dynamic decision networks. In *2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. 7–13.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Patt. Anal. Mach. Intell.* 35, 8 (2013), 1798–1828.

[7] Jason Brownley. 2020. *Data Preparation for Machine Learning*. Retrieved from https://machinelearningmastery.com/data-preparation-for-machine-learning/.

[8] Radu Calinescu, Lars Grunske, Martha Kwiatkowska, Raffaela Mirandola, and Giordana Tamburrelli. 2011. Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Softw. Eng.* 37, 3 (May 2011), 387–409. DOI: https://doi.org/10.1109/TSE.2010.92

[9] Javier Cámara, Henry Muccini, and Karthik Vaidhyanathan. 2020. Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive IoT systems. In *International Conference on Software Architecture*. 11–22.

[10] Lorena Castañeda, Norha M. Villegas, and Hausi A. Müller. 2014. Self-adaptive applications: On the development of personalized web-tasking systems. In *9th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 49–54.

[11] Tao Chen and Rami Bahsoon. 2016. Self-adaptive and online QoS modeling for cloud-based software services. *IEEE Trans. Softw. Eng.* 43, 5 (2016), 453–475.

[12] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. 2018. FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software. *ACM Trans. Softw. Eng. Methodol.* 27, 2 (2018), 1–50.

[13] Betty Cheng, Andres Ramirez, and Philip K. McKinley. 2013. Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty. In *1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*. 1–6. DOI: https://doi.org/10.1109/CMSBSE.2013.6604427

[14] Betty H. C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic et al. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-adaptive Systems*. Springer, 1–26.

[15] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. 2009. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Model Driven Engineering Languages and Systems*, Andy Schürr and Bran Selic (Eds.). Springer, 468–483.

[16] Zack Coker, David Garlan, and Claire Le Goues. 2015. SASS: Self-adaptation using stochastic search. In *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 168–174. DOI: https://doi.org/10.1109/SEAMS.2015.16

[17] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. 2015. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* 17, 4 (2015), 397–415.

[18] Rogério De Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo et al. 2017. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-adaptive Systems III. Assurances*. Springer, 3–30.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[20] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, and Brad Petrus. 2009. Architecture-driven self-adaptation and self-management in robotics systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-managing Systems*. IEEE, 142–151.

[21] Ross Edwards and Nelly Bencomo. 2018. DeSiRE: Further understanding nuances of degrees of satisfaction of non-functional requirements trade-off. In *13th International Conference on Software Engineering for Adaptive and Self-managing Systems (SEAMS'18)*. Association for Computing Machinery, New York, NY, 12–18. DOI: https://doi.org/10.1145/3194133.3194142

[22] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. FUSION: A framework for engineering self-tuning self-adaptive software systems. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 7–16.

[23] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. Run-time efficient probabilistic model checking. In *33rd International Conference on Software Engineering (ICSE'11)*. Association for Computing Machinery, New York, NY, 341–350. DOI: https://doi.org/10.1145/1985793.1985840

[24] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. 2016. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Softw. Eng.* 42, 1 (2016), 75–99. DOI: https://doi.org/10.1109/TSE.2015.2421318

[25] David Garlan, Shang-Wen Cheng, An Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (Oct. 2004), 46–54. DOI: https://doi.org/10.1109/MC.2004.175

[26] Simos Gerasimou, Radu Calinescu, and Alec Banks. 2014. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *9th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS'14)*. Association for Computing Machinery, New York, NY, 115–124. DOI: https://doi.org/10.1145/2593929.2593932

[27] Aurélien Géron. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media.

[28] Sona Ghahremani, Holger Giese, and Thomas Vogel. 2020. Improving scalability and reward of utility-driven self-healing for large dynamic architectures. *ACM Trans. Auton. Adapt. Syst.* 14, 3 (Feb. 2020). DOI: https://doi.org/10.1145/3380965

[29] Omid Gheibi, Danny Weyns, and Quin Federico. 2021. Applying machine learning in self-adaptive systems: A systematic literature review. *Trans. Auton. Adapt. Syst.* 15, 3 (2021), 1–37. https://doi.org/10.1145/3469440

[30] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. On the impact of applying machine learning in the decision-making of self-adaptive systems. In *16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 104–110. DOI: https://doi.org/10.1109/SEAMS51251.2021.00023

[31] Heather J. Goldsby, Betty H. C. Cheng, and Ji Zhang. 2008. AMOEBA-RT: Run-time verification of adaptive software. In *Models in Software Engineering*, Holger Giese (Ed.). Springer Berlin, 212–224.

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

[33] Han Nguyen Ho and Eunseok Lee. 2015. Model-based reinforcement learning approach for planning in self-adaptive software system. In *9th International Conference on Ubiquitous Information Management and Communication*. 1–8.

[34] Ching-Lai Hwang and Abu Syed Md Masud. 2012. *Multiple Objective Decision Making-methods and Applications: A State-of-the-art Survey*. Vol. 164. Springer Science & Business Media.

[35] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. 2017. DeltaIoT: A self-adaptive internet of things exemplar. In *IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)*. IEEE, 76–82.

[36] M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: Active formal models for self-adaptation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, 125–134.

[37] Pooyan Jamshidi, Javier Cámara, Bradley Schmerl, Christian Käestner, and David Garlan. 2019. Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots. In *IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'19)*. IEEE, 39–50.

[38] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada. 2016. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA'16)*. 70–79.

[39] Sebastian Junges, Hazem Torfah, and Sanjit A. Seshia. 2021. Runtime monitors for markov decision processes. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 553–576.

[40] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[41] Dongsun Kim and Sooyong Park. 2009. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Workshop on Software Engineering for Adaptive and Self-managing Systems*. IEEE, 76–85.

[42] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. 2018. Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In *13th International Conference on Software Engineering for Adaptive and Self-managing Systems*. 40–50.

[43] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *36th International Conference on Software Engineering*. 700–711.

[44] Stephen Kokoska and Daniel Zwillinger. 2000. *CRC Standard Probability and Statistics Tables and Formulae*. CRC Press.

[45] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: An architectural challenge. In *Future of Software Engineering*. DOI: https://doi.org/10.1109/FOSE.2007.19

[46] Marta Kwiatkowska, Gethin Norman, and David Parker. 2002. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation: Modelling Techniques and Tools*, Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder (Eds.). Springer Berlin, 200–204.

[47] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72. DOI: https://doi.org/10.1109/TSE.2011.104

[48] Axel Legay, Benoît Delahaye, and Saddek Bensalem. 2010. Statistical model checking: An overview. In *International Conference on Runtime Verification*. Springer, 122–135.

[49] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi, and Klaus Pohl. 2019. Feature-model-guided online learning for self-adaptive systems. *arXiv preprint arXiv:1907.09158* (2019).

[50] Gabriel Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Foundations of Software Engineering*. ACM, 1–12. DOI: https://doi.org/10.1145/2786805.2786853

[51] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2018. Flexible and efficient decision-making for proactive latency-aware self-adaptation. *ACM Trans. Auton. Adapt. Syst.* 13, 1 (2018), 1–36.

[52] Henry Muccini, Mohammad Sharaf, and Danny Weyns. 2016. Self-adaptation for cyber-physical systems: A systematic literature review. In *11th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 75–81.

[53] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding faster configurations using FLASH. *IEEE Trans. Softw. Eng.* (2018). DOI: https://doi.org/10.1109/TSE.2018.2870895

[54] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (2009), 1345–1359.

[55] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. 2013. Run-time adaptation of mobile applications using genetic algorithms. In *8th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 73–82. DOI: https://doi.org/10.1109/SEAMS.2013.6595494

[56] Fabian Pedregosa et al. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[57] Wenyi Qian, Xin Peng, Bihuan Chen, John Mylopoulos, Huanhuan Wang, and Wenyun Zhao. 2015. Rationalism with a dose of empiricism: Combining goal reasoning and case-based reasoning for self-adaptive software systems. *Requirem. Eng.* 20, 3 (2015), 233–252.

[58] Federico Quin, Danny Weyns, Thomas Bamelis, Singh Buttar Sarpreet, and Sam Michiels. 2019. Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning. In *IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS'19)*. IEEE, 1–12.

[59] Federico Quin, Danny Weyns, and Omid Gheibi. 2021. Decentralized self-adaptive systems: A mapping study. In *16th International Symposium on Software Engineering for Adaptive and Self-managing Systems*. IEEE, 18–29. DOI: https://doi.org/10.1109/SEAMS51251.2021.00014

[60] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[61] Andres J. Ramirez, Betty H. C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. 2010. Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In *7th International Conference on Autonomic Computing (ICAC'10)*. Association for Computing Machinery, New York, NY, 225–234. DOI: https://doi.org/10.1145/1809049.1809080

[62] Andres J. Ramirez, David B. Knoester, Betty H. C. Cheng, and Philip K. McKinley. 2009. Applying genetic algorithms to decision making in autonomic computing systems. In *6th International Conference on Autonomic Computing (ICAC'09)*. Association for Computing Machinery, New York, NY, 97–106. DOI: https://doi.org/10.1145/1555228.1555258

[63] Willi Richert and Bernd Kleinjohann. 2008. Adaptivity at every layer: A modular approach for evolving societies of learning autonomous systems. In *International Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS'08)*. Association for Computing Machinery, New York, NY, 113–120. DOI: https://doi.org/10.1145/1370018.1370039

[64] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[65] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein et al. 2015. ImageNet large scale visual recognition challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252.

[66] Clay Stevens and Hamid Bagheri. 2020. Reducing run-time adaptation space via analysis of possible utility bounds. In *42nd International Conference on Software Engineering. ICSE*.

[67] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. 2013. Learning revised models for planning in adaptive systems. In *35th International Conference on Software Engineering*. IEEE, 63–71.

[68] Gerald Tesauro and Jeffrey O. Kephart. 2004. Utility functions in autonomic systems. In *1st International Conference on Autonomic Computing (ICAC'04)*. IEEE Computer Society, 70–77.

[69] Jeroen Van Der Donckt, Danny Weyns, M. Usman Iftikhar, and Ritesh Kumar Singh. 2018. Cost-benefit analysis at runtime for self-adaptive systems applied to an Internet of Things application. In *International Conference on Evaluation of Novel Approaches to Software Engineering*. 478–490. Retrieved from https://people.cs.kuleuven.be/danny.weyns/papers/2020SEAMSa.pdf.

[70] Jeroen Van Der Donckt, Danny Weyns, Federico Quin, Jonas Van Der Donckt, and Sam Michiels. 2020. Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. *International Symposium on Software Engineering for Adaptive and Self-managing Systems*. 20–30.

[71] Danny Weyns. 2019. Software Engineering of Self-adaptive Systems. In *Handbook of Software Engineering*, Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang (Eds.). Springer. 399–443. DOI: 10.1007/978-3-030-00262-6_11

[72] Danny Weyns. 2020. *Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. Wiley, IEEE Computer Society Press.

[73] Danny Weyns, Nelly Bencomo, Radu Calinescu, Javier Cámara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek et al. 2017. Perpetual assurances for self-adaptive systems. In *Software Engineering for Self-adaptive Systems III. Assurances*. Springer, 31–63.

[74] D. Weyns and M. U. Iftikhar. 2016. Model-based simulation at runtime for self-adaptive systems. In *IEEE International Conference on Autonomic Computing (ICAC'16)*. 364–373.

[75] Danny Weyns and M. Usman Iftikhar. 2019. ActivFORMS: A Model-based Approach to Engineer Self-adaptive Systems. arXiv:cs.SE/1908.11179

[76] Danny Weyns, M. Usman Iftikhar, Danny Hughes, and Nelson Matthys. 2018. Applying architecture-based adaptation to automate the management of Internet-of-Things. In *European Conference on Software Architecture*. Springer, 49–67.

[77] Danny Weyns and Usman Iftikhar. 2016. Model-based simulation at runtime for self-adaptive systems. *Proc. Models Runt., Würz. 2016* (2016), 1–9.

[78] Danny Weyns and Usman Iftikhar. 2022. ActivFORMS: A formally-founded model-based approach to engineer self-Adaptive systems. *ACM Trans. Softw. Eng. Methodol.* (2022). Accepted on February 2022. DOI: https://doi.org/10.1145/3522585

[79] D. Weyns, U. Iftikhar, and J. Soderland. 2013. Do external feedback loops improve the design of self-adaptive systems? A controlled experiment. In *Software Engineering for Adaptive and Self-managing Systems*. IEEE.

[80] Danny Weyns, Sam Malek, and Jesper Andersson. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* 7, 1 (2012), 8:1–8:61. DOI: https://doi.org/10.1145/2168260.2168268

[81] Danny Weyns, Gowri Sankar Ramachandran, and Ritesh Kumar Singh. 2018. Self-managing Internet of Things. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 67–84.

[82] Håkan L. S. Younes and Reid G. Simmons. 2006. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Computat.* 204, 9 (2006), 1368–1409.

[83] Ji Zhang, Heather J. Goldsby, and Betty H. C. Cheng. 2009. Modular verification of dynamically adaptive systems. In *8th ACM International Conference on Aspect-oriented Software Development*. 161–172.

[84] M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM. 125–134. DOI: 10.1145/2593929.2593944

Received July 2021; revised February 2022; accepted April 2022