Applying Deep Learning to Reduce Large Adaptation Spaces of Self-Adaptive Systems with Multiple Types of Goals

Jeroen Van Der Donckt
jeroen.vanderdonckt@student.kuleuven.be
Katholieke Universiteit Leuven
Leuven, Belgium

Danny Weyns
danny.weyns@kuleuven.be
KU Leuven & Linnaeus University
Leuven, Belgium

Federico Quin
federico.quin@kuleuven.be
Katholieke Universiteit Leuven
Leuven, Belgium

Jonas Van Der Donckt
jonvdrdo.vanderdonckt@ugent.be
Ghent University
Zwijnaarde, Belgium

Sam Michiels
sam.michiels@cs.kuleuven.be
Katholieke Universiteit Leuven
Leuven, Belgium

ABSTRACT

When a self-adaptive system needs to adapt, it has to analyze the possible options for adaptation, i.e., the adaptation space. For systems with large adaptation spaces, this analysis process can be resource- and time-consuming. One approach to tackle this problem is using machine learning techniques to reduce the adaptation space to only the relevant adaptation options. However, existing approaches only handle threshold goals, while practical systems often need to address also optimization goals. To tackle this limitation, we propose a two-stage learning approach called Deep Learning for Adaptation Space Reduction (DLASeR). DLASeR applies a deep learner first to reduce the adaptation space for the threshold goals and then ranks these options for the optimization goal. A benefit of deep learning is that it does not require feature engineering. Results on two instances of the DeltaIoT artifact (with different sizes of adaptation space) show that DLASeR outperforms a state-of-the-art approach for settings with only threshold goals. The results for settings with both threshold goals and an optimization goal show that DLASeR is effective with a negligible effect on the realization of the adaptation goals. Finally, we observe no noteworthy effect on the effectiveness of DLASeR for larger sizes of adaptation spaces.

CCS CONCEPTS

• **Software engineering → Extra-functional properties**.

KEYWORDS

Self-adaptation, adaptation space, deep learning

ACM Reference Format:
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. In *SEAMS '20: 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 25–26, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Many systems today operate in uncertain environments. For such systems, employing a fixed configuration may result in sub-optimal performance. For example, a web-based system that runs a fixed number of servers will waste a lot of resources when the load is very low, but be insufficient when there is a peak load [25].

Self-adaptation is one prominent approach to tackle such problems [6, 40]. A self-adaptive system uses a feedback loop that monitors the system and its environment to determine whether the adaptation goals are satisfied under uncertain conditions [12, 30, 36]. In case the goals are not satisfied, the feedback loop determines the best option for adapting the system to meet the adaptation goals. In the above example, enhancing the system with self-adaptation capabilities will enable it to increase or decrease the number of servers based on the actual load. This results in higher user satisfaction as well as improved economical and ecological use of resources.

In this paper we apply architecture-based adaptation [14][26][45] using the MAPE-K reference model (Monitor - Analyzer - Planner - Executor - Knowledge) [23, 44]. Our focus is on the analysis of the adaptation options, i.e., the adaptation space (a task of the analyzer), and ranking the adaptation options enabling to select the best option (a task of the planner). The adaptation space consists in general of all the possible configurations that can be reached by applying adaptation actions to the current configuration of the system. Finding the best adaptation option in a large adaptation space is often computationally expensive [5, 6, 9, 41], in particular when rigorous analysis techniques are used, see for example [3, 32]. One approach to tackle this problem is improving the run-time performance of model checking in two steps: a pre-computation performed at design time results in a set of symbolic expressions that are evaluated at runtime by replacing variables with monitored values [13]. Another more generally applicable approach that has been proposed recently is reducing the adaptation space. Adaptation space reduction aims at retrieving a subset of adaptation options from an adaptation space that contains only relevant system configurations that are then considered for analysis.

Existing work has showed that different techniques can be used to reduce adaptation spaces, including feature selection [4, 31], search-based techniques, e.g., [5, 24], and machine learning, see e.g., [11, 37]. Among the learning techniques that have been studied are decision trees, classification, regression, and online learning. However, current approaches only handle threshold goals, i.e., goals

---

SEAMS '20, May 25–26, 2020, Seoul, South Korea

ACM ISBN AAA-B-CCC-XXXX-X/YY/ZZ. . . $XX.YY
https://doi.org/10.1145/1122445.1122456

where a system parameter needs to stay below or above a given value. For instance, the throughput of a web-based system should not drop below a required level. For many practical systems today this is not sufficient as they also need to address optimization goals. For instance, the operational cost of a web-based system should be minimized. This leads to the following research question:

How to reduce large adaptation spaces of self-adaptive systems with multiple threshold goals and an optimization goal effectively?

With effectively, we mean the solution should ensure: 1) the reduced adaptation space is significantly smaller, 2) high performance, i.e., the reduced adaptation space covers the relevant adaptation options well, 3) the effect of the state space reduction on the realization of the adaptation goals is negligible, 4) there is no notable effect on 1, 2 and 3 for larger sizes of the adaptation space.

To answer the research question, we propose a novel approach for adaptation space reduction: "Deep Learning for Adaptation Space Reduction" (DLASeR). DLASeR reduces the adaptation space in two learning stages. In the first stage, a classification deep neural network is applied to reduce the adaptation space for the threshold goals. In the second stage, a regression deep neural network is applied to rank the options, further reducing the adaptation space for the optimization goal.

The motivation for applying deep learning is threefold. First, compared to classic machine learning that relies on linear models as for instance in [37], we want to investigate the effectiveness of deep learning relying on non-linear models. Second, classic learning techniques usually require feature engineering, but deep learning can handle raw data, making feature engineering unnecessary. Third, given the success of deep learning in various other domains, e.g. computer vision, we want to explore the applicability of deep learning for an important problem in self-adaptive-systems.

The effectiveness of DLASeR is evaluated on two instances of the DeltaIoT artifact [19], with different adaptation space sizes. The Internet-of-Things is a particularly interesting domain to apply self-adaptation, given its complexity and high degrees of uncertainties [46]. We define different metrics to evaluate DLASeR and compare the results with alternative approaches.

The remainder of this paper is structured as follows. Section 2 illustrates the problem we tackle in this paper with a concrete example. Section 3 gives a high-level overview of the research methodology. In Section 4, we specify metrics to measure the performance of solutions. Section 5 presents DLASeR and its learning pipeline. In Section 6, we describe the MAPE-K runtime architecture enhanced with DLASeR. In Section 7, we evaluate DLASeR for different settings. Section 8 discusses related work on the use of learning in self-adaptation. Finally we draw conclusions in Section 9.

2 PROBLEM ILLUSTRATION BY EXAMPLE

In this section, we illustrate the problem we tackle in this paper with a concrete example case in the domain of Internet of Things. We use the same case for the evaluation of DLASeR in Section 7.

2.1 DeltaIoT in a Nutshell

The DeltaIoT artifact offers a reference Internet-of-Things (IoT) application that supports research on self-adaptation [19].

DeltaIoT comprises of a set of battery-powered motes that are equipped with different types of sensors to measure parameters in the environment. The network employs wireless multi-hop communication to relay the sensor data from the motes to a gateway that is connected with a user application. In this paper, we use two instances of DeltaIoT, one with 15 motes, and one with 37 motes. The larger network is more challenging in terms of the number of possible configurations to adapt the system. We use *DeltaIoTv1* and *DeltaIoTv2* to refer to the smaller and larger instance of the network respectively. The networks are time-synchronized, i.e., the communication is organized in cycles with a fixed number of slots. Neighbouring motes are assigned such slots during which they can exchange packets. The cycle time for DeltaIoTv1 is 8 minutes, while it is 9.5 minutes for DeltaIoTv2.

Crucial quality goals of DeltaIoT are the energy consumed by the motes (to be minimized), and packet loss and latency of packet transmission (to be kept below a given level). These goals are conflicting as transmitting packets with less energy will reduce packet loss but reduce the lifetime of batteries. Furthermore, the IoT network is subject to various types of uncertainty, which makes it very challenging to achieve the quality goals. The main uncertainties are interference along network links and changing load in the network, i.e., motes only send packets when there is useful data, which may depend on environment conditions that are difficult to predict.

To achieve the goals despite of the uncertainties, the network settings need to be optimized. We consider two possible settings here. On the one hand, the transmission power of the motes can be set in discrete steps form 1 to 15. On the other hand, the distribution of messages sent to parents, i.e., the distribution factor, can be set. If a mote only has one parent, it obviously will relay 100% of its messages to its parent. But, when a mote has multiple parents (in our case maximum two), the fractions of messages sent to the parents can be chosen. For DeltaIoTv1, we consider settings in steps of 20%, while for DeltaIoTv2 we consider settings in steps of 33%.

In practice, typically a conservative approach is used where the transmission power of the motes is set to maximum and all messages are duplicated to all parents. These settings are then manually adjusted using trial and error. In this paper, we automate these costly and error-prone management tasks using self-adaptation. The quality goals of the IoT network then become adaptation goals.

2.2 Adaptation goals

In this paper, we consider two types of adaptation goals: (i) a threshold goal that requires that some quality property should either remain below or above a given value, and (ii) an optimization goal that requires that some quality property should be minimized or maximized. Concretely, the adaptation goals for DeltaIoT that we consider are defined as follows:

T1: The average packet loss over 12 hours should not exceed 10 %.
T2: The average latency over 12 hours should not exceed 5 %.
O1: The energy consumption should be minimized.

In the approach we present in this paper, the threshold goals define constraints that determine which of the adaptation options of the adaptation space are relevant and which options are not relevant. The relevant adaptation options can then be ranked based on the optimization goal and the highest ranked optimizes the goal.

2.3 Adaptation Space

The adaptation space is the set of adaptation options. In DeltaIoT, the permutation of all possible adaptation settings of the system determines the adaptation space. For DeltaIoTv1, this results in an adaptation space of 216 adaption options. The adaptation space of DeltaIoTv1is significantly larger and consists of 4096 options.

Applying different adaptation options will produce different qualities for the system. For DeltaIoT these qualities are packet loss, latency and energy consumption. During the analysis of the adaptation options, these qualities can be predicted. Fig. 1 shows the adaptation space at a particular point in time (i.e., for a particular cycle) of the DeltaIoTv1 network. The adaptation options are presented in terms of their qualities predicted during analysis. Different techniques can be used to predict the qualities of adaptation options. For instance, the quality of the system can be represented as a parameterized Discrete Time Markov Model. One set of parameters represent the possible settings of the system, another set represent uncertainties. By configuring the model for a given setting using the current knowledge about uncertainties, and representing the quality property of interest as a logical expression, one can use probabilistic model checking to determine the expected quality of the adaptation option, see for instance [3].

Representing the adaptation options in 3D allows to visually grasp how the options are expected to achieve the adaptation goals. The threshold values for the packet loss and latency goals demarcate the relevant adaptation options, which are graphically represented by the grey box of the figure. The positioning of each adaptation option along the z-axis corresponds with the prediction of the quality property that needs to be optimized. The best adaptation option will then be the lowest in the box.

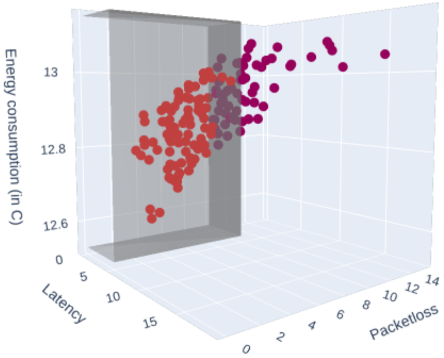


Figure 1: Adaptation space for DeltaIoTv1 at one point in time. Red and blue dots represent respectively adaption options that meet and not meet the threshold goals T1 and T2.

It is important to note that adaptation options have dynamic behavior, i.e., their predicted qualities change over time (i.e., in different adaptation cycles). This is direct consequence of the uncertainties that the system is subjected to. Suppose that there is suddenly substantial more interference (e.g. noise) in cycle $x + 1$. Then it is obvious that for most adaptation options the packet loss in cycle $x + 1$ will be higher than the previous cycle x. Fig. 2 shows how the predicted quality values for one of the adaptation options change over a sequence of cycles.

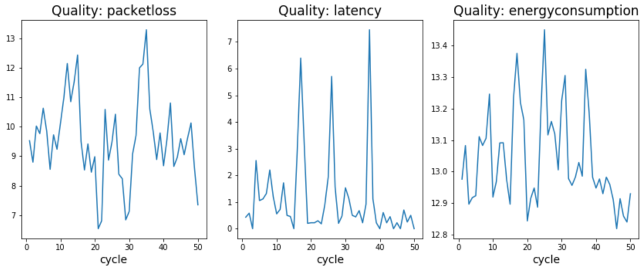


Figure 2: Evolution of the qualities of a certain adaptation option of in DeltaIoTv1 over a sequence of cycles.

2.4 Illustration of the Problem

Consider now the adaptation space of DeltaIoTv2 at some point in time, as shown in Figure 3

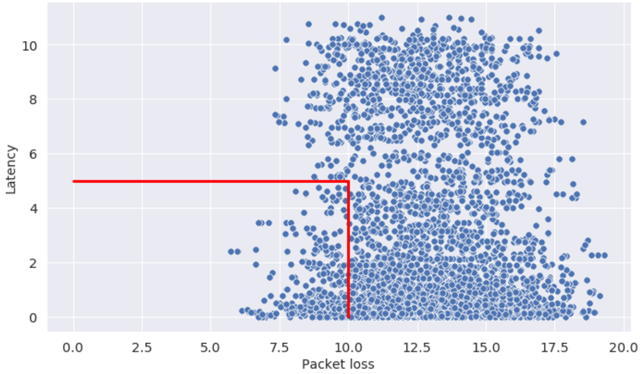


Figure 3: Example of a large adaptation space in DeltaIoTv2.

The adaptation space in this example consists of 4096 adaptation options. Applying rigorous methods to analyze this large adaptation space will be infeasible within the available cycle time. The problem is then how to reduce this space to only the valid adaptation options that can then be analyzed. The relevant adaptation options are those in the box defined by the threshold goals in Figure 3.

2.5 Learning Approach

In DLASeR, we apply deep learning to reduce large adaptation spaces. Deep learning uses neural networks as learning models. A neural network connects neurons that are organized in layers. The layer that receives external data is the input layer, while the layer that produces the result is the output layer. In DeltaIoT, the input of the neural network are configurations of adaptation options with uncertainties in the environment; the output are predictions of the quality properties of adaptation options.

The neurons between layers can be connected in different ways, e.g., fully connected or group based. A propagation function computes the input to a neuron from the outputs of its predecessor neurons and their connections as a weighted sum. The network learns by adjusting the weights of the connections by minimizing the observed errors, improving the accuracy of results over time.

3 RESEARCH METHODOLOGY

To tackle the research problem, we followed a systematic methodology as shown in Fig. 4.

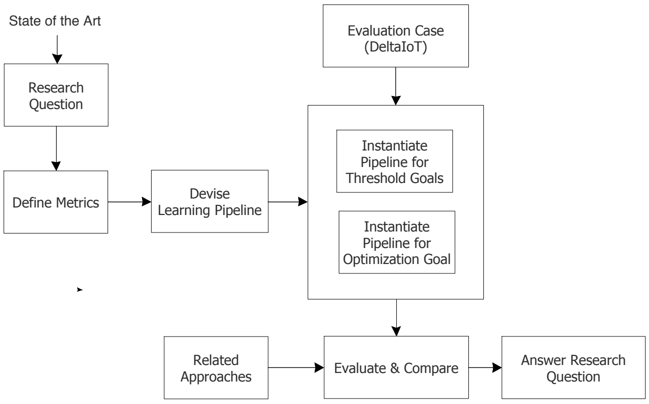


Figure 4: Overview of research methodology.

Based on the state-of-the-art and our own experiences with applying self-adaptation in IoT, we defined the research question (see the introduction). Earlier work that applied machine learning techniques to reduce large adaptation spaces focused on threshold goals only. In this paper, we focus on both threshold and optimization goals. Inspired by recent progress in the area of deep learning, we decided to investigate whether we can use deep learning to effectively reduce large adaptation spaces for both threshold or optimization goals, without compromising the system goals.

Once we determined the research question, we specified the metrics that allow us to measure the effectiveness of DLASeR and compare it with other approaches. Next, we devised DLASeR's learning pipeline that applies deep learning both for threshold and optimization goals. This pipeline was then instantiated for the two evaluation cases of DeltaIoT. We then evaluated the DLASeR and compared it with representative related approaches. Finally, we reflect and provide an answer to the research question.

4 METRICS

To evaluate DLASeR, we identified two metrics to evaluate the performance of the trained learning model, and a third metric to compare the effectiveness of DLASeR with other approaches.

4.1 Precision and recall

Precision and recall are two important metrics to evaluate an approach that deals with adaptation space reduction. Precision captures how many of the predicted adaptation options are relevant. Recall, on the other hand, indicates how many of the relevant adaptation options are predicted. A low recall will result in more feasible adaptation options that are not verified.

Both metrics are combined in a single score, called F1 score, that gives both metrics an equal importance. Concretely, the F1-score is defined as the harmonic mean of precision and recall: [15]:

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{1}$$

We will use the F1 score to evaluate DLASeR and compare solutions for the reduction of the adaptation space based on threshold goals.

4.2 Spearman correlation

Spearman correlation, or Spearman's rho, is a useful non-parametric measure of rank correlation. We use this metric for evaluating the ranking of adaptation options based on an optimization goal. Spearman correlation converts the values of the estimated quality property (for each adaptation option) to numeric ranks. Then, the linear relationship is computed with the true ranks. This allows assessing how well a monotonic function describes the relationship between the predicted and the true ranks. Large errors are penalized harder. For example, swapping the first and third rank in prediction results in a worse association than swapping the first and second rank. For more details about Spearman correlation, we refer to [47]

4.3 Average adaptation space reduction

In addition to F1 and Spearman correlation, we define an integrated metric to compare the effectiveness of different adaptation space reduction approaches. This metric captures the average (relative) adaptation space reduction (*AASR* in short). *AASR* measures percentage of adaptation options that are not considered relevant. The metric *AASR* is particularly suitable for the problem under consideration as it covers the high-end goal of adaptation space reduction.

The interpretation of the relative adaptation space reduction depends on the types of goals considered. When focusing on threshold goals (we refer to this as the threshold approach), the relative adaptation space reduction corresponds to the percentage of adaptation options that are predicted to conform with the given threshold goals. In the case of optimization goals (i.e., the optimization approach), the relative adaptation space reduction corresponds to 100% minus the percentage of adaptation options that have to be analyzed until an adaptation option is found that meets all the threshold goals.

It is important to note that the relative adaptation space reduction metric is influenced by the "hardness" of the threshold goals. Suppose that the threshold goals are not very restrictive, thus many adaptation options lay within the feasible box (see Fig. 3). Then, the threshold approach will result in a rather small reduction, whereas for the optimization approach a large reduction is expected. On the other hand, for very restrictive threshold goals the opposite is true. A larger reduction for the threshold approach can be expected, and a smaller reduction for the optimization approach.

5 DEEP LEARNING FOR ADAPTATION SPACE REDUCTION

We introduce now DLASeR. DLASeR spans two stages of a learning pipeline: an offline and online stage. We defined a generic learning pipeline that can be used both for threshold and optimization goals. For each goal type, we select a distinct deep learning model and a scaler that are maintained throughout the pipeline. We use the DeltaIoT example to illustrate DLASeR and its learning pipeline.

5.1 Offline Stage of DLASeR Learning Pipeline

In the offline stage of the pipeline, shown in Fig. 5, we aggregate data of a series of adaptation cycles (via observation or simulation of the system). In order to successfully train a (deep) neural network, it is important that all relevant data is collected. We refer to this data

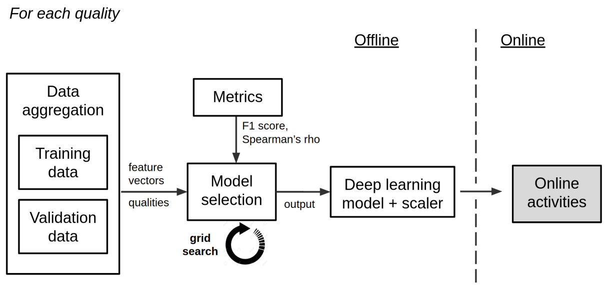


Figure 5: The offline stage of the learning pipeline.

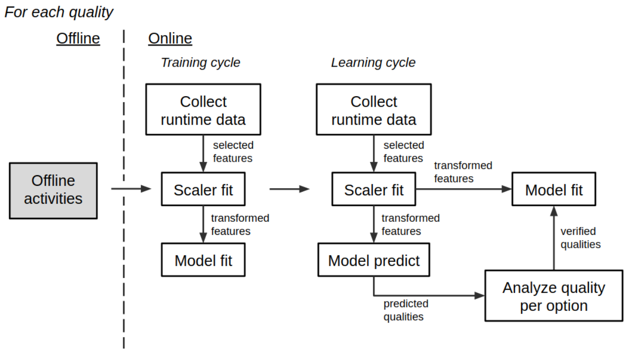


Figure 6: The online stage of the learning pipeline.

as *features*.[1]. A feature vector for DeltaIoT contains: the distribution factor per link that determines the percentage of messages sent over the link and the setting of the transmission power for each link (configuration parameters), the traffic load and the signal-to-noise ratio over the links (uncertainties), and the current qualities of the system (one for each adaptation goal).[2] The aggregated data allows us to perform back-testing of candidate learning solutions on the application and evaluate their performance. To that end, we have split the aggregated data in a train set and a validation set. It is important that both sets contain data of consecutive cycles without any overlap. For example, if the train data contains all data from cycle 1 up to cycle n, then the validation data should contain all the remaining data starting from cycle $n + 1$.

The main activity of the offline stage of the pipeline is model selection for which we use back-testing. During model selection, we use the data of the train set as input to learn the parameters of the deep learning models, one model for each quality goal. Then, we let the models make predictions for the corresponding quality property using the validation set. More specifically, model selection aims at finding optimal hyper-parameter settings. The hyper-parameters are the number of layers of the deep learning model, the number of neurons, the batch size, and the learning rate, the optimizer, as well as the scalar algorithm. We explain batch size, learning rate, the optimizer, and the scalar algorithm below. To determine the optimal values of the hyper-parameters for a given model we applied grid search [15], meaning that a model is trained and evaluated for every combination of hyper-parameters. The training process ends when the model is overfitting, i.e., the model learns the details and noise in the training data to the extent that it negatively impacts the performance of the model on new data.[3] Once the models are trained, they are evaluated on the validation data. The effectiveness of models for threshold goals is measured by the F1-score, whereas we use Spearman's rho for determine the effectiveness of models for optimization goals.

Model selection results in a deep learning model and a scaler algorithm. We explain these now more in detail.

5.1.1 Scaler. A scaler normalizes the data of the features, which is common pre-processing step in machine learning. Learning algorithms tend to perform better when scaling is applied to the data [21]. Shanker M. et al. demonstrated these effects also for neural networks [38]. During grid search we determined the optimal scaler for the different deep learning models. We considered the options standard scaling, min-max scaling, max-abs scaling, and no scaling.

5.1.2 Deep learning model. We started with studying classification deep neural networks. These networks are trained to predict whether or not an adaptation option meets a threshold goal. Then we studied regression deep neural networks. These networks are trained to infer a ranking for the adaptation options. This ranking is based on the regressed values for the optimization quality.

As we explained above, the hyper-parameters determined during grid search (besides the scalar algorithm) are: the number of layers of the deep learning model (i.e., network depth), the number of neurons in each layer, batch size, learning rate, and the optimizer.

The depth of the network and the number of neurons in each layer influence the complexity and also to total number of learnable parameters, which affects the learning time. The batch size defines how much of the data shall be looked at before doing an update of the model based on the output of the model (classification of adaptation options for the models of threshold goals and ranking of adaptation options for the models of optimization goals). A large batch size results in less updates, where a small one results in many updates. This affects the granularity of the learning. Batch size is highly correlated with the learning rate. The learning rate determines the degree of updates that are performed during the learning process. This will affect the learning speed. Finally, the model optimizer determines the algorithm that is used to perform the parameter updates utilizing the learning rate. Setting all these hyper-parameters properly enables the deep neural network to adapt not too slow nor too fast to changes.

When the deep learning models are fine-tuned and proper scalers are determined, these elements are integrated into a coherent solution. The solution can then be deployed which brings us to the second stage of the pipeline.

5.2 Online Stage of DLASeR Learning Pipeline

The online stage, shown in Fig. 6, consists of two cycles.

---

[1] A feature refers to measurable property or characteristic of the system that affects the effectiveness of the learning algorithms. These features should not be confused with features that define the adaptation space, such as for instance authentication or caching techniques as in [11].

[2] We decided not to apply feature selection, which is a common pre-processing step in machine learning. Feature selection typically requires domain knowledge. Furthermore, it is possible to miss important features due to the limited coverage of the aggregated data. Since deep learning can handle raw data, feature selection can be avoided.

[3] We applied an early stopping procedure that discards 10% of the train data to check whether or not there is a large discrepancy in the train error, and halt if necessary.

5.2.1 *Training Cycle.*

In the *Training cycle*, the learning models are trained. The goal of this cycle is to initialize the learnable parameters of the models properly to the current setting of the problem at hand. To that end, relevant runtime data is collected to construct feature vectors. This data is then used to update the scaler (e.g., updating min/max values of parameters, the means, etc.) and the feature vectors are adjusted accordingly. The deep learning model is then trained using the transformed features, meaning the parameters of the models are initialized, for instance the weights of the contributions of neurons. The training stage 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. Since in this cycle the models are only initialized, there is no reduction of the adaptation space yet.

5.2.2 *Learning Cycle(s).*

In the *Learning cycle*, the learning models are actively used to make predictions about the qualities of adaptation options to reduce the adaptation space. Furthermore, the analysis results are used to incrementally update the learning models. Concretely, the features of the feature vectors are transformed similar as for the training cycle. The deep learning model then makes predictions for the feature vector of each adaptation option. The adaptation options are verified based on these predictions. In our work, we use statistical model checking [2, 8, 42, 43] at runtime to verify the adaptation options during analysis, however other analysis techniques can be applied. Analyzing the adaptation options differs for the threshold and the optimization approach.

For the **threshold approach**, the adaptation options that are predicted as relevant, i.e., compliant with the threshold goals, are used for verification. In addition, we add a fixed percentage of the other adaptation options, i.e., the *exploration rate*. We explore these options that due to uncertainties, were not considered relevant so far, may now become relevant. As a final step, the verification results are used to further improve the learning models. Algorithm 1 shows how the adaptation space reduction is applied for threshold goals.

Algorithm 1 Adaptation space reduction for threshold goals

```
1:  $pred\_subspace \leftarrow K.adaptation\_options$
2:  for each $DL\_thresh\_model$ in $K.threshold\_models$ do
3:      $preds \leftarrow DL\_thresh\_model$.predict($features$)
4:      $pred\_subspace \leftarrow pred\_subspace \cap preds$
5:  end for
6:  $unselected \leftarrow K.adaptation\_options \setminus pred\_subspace$
7:  $explore \leftarrow unselected$.randomSelect($exploration\_rate$)
8:  $subset \leftarrow pred\_subspace \cup explore$
9:  $Analyzer$.verifyAdaptOpt($subset$)                ▷ Knowledge
10: $features, qualities \leftarrow K.verification\_results$
11: for each $DL\_thresh\_model$ in $K.threshold\_models$ do
12:     $DL\_thresh\_model$.update($features, qualities$)
13: end for
```

In lines 1 to 5 the relevant subspace is predicted based on threshold goals (*DL_thresh_model* refers to a model for any threshold goal). For each threshold goal, the corresponding deep learning model, stored in the Knowledge (K), predicts the relevant subspace. Note that the *predict()* function on line 3 first transforms the features before making predictions. The intersection of the predicted subspaces that are relevant for each goal represents the relevant subspace. This process represents the actual adaptation space reduction for all threshold goals. In lines 6 to 9 the adaptation options for verification are selected. We start from the relevant subspace and add a randomly selected set of adaptation options that are not in the predicted subspace based on the *exploration_rate*. Then, the combined set is sent to the verifier for analysis. The verification results, i.e., the predicted qualities of the verified adaptation options are stored in the knowledge. In lines 10 to 13 the verification results are exploited to update the learning models for the threshold goals. First, we retrieve the features and the analysis results (i.e., the qualities for each threshold goal) of the verified adaptation options from the knowledge. Then, for each threshold goal, the corresponding model is updated. To that end, the relevant quality for the given model is selected, e.g. packet loss for the packet loss threshold model. This information is then used to update the parameters of the deep learning model using the same learning mechanism as we used in the training cycle. In case there is no optimization goal, the selection of the adaptation option to adapt the system can be made based on the adaption options that meet all the threshold goals. However, if there is an optimization goal, we need to identify the adaptation option that optimizes this goal.

Algorithm 2 Adaptation space reduction for threshold and optimization goals.

```
1:  Predict relevant subspace (line 1 to 5 from Algorithm 1)
2:  $DL\_opt\_model \leftarrow K.optimization\_model$
3:  $ranking \leftarrow DL\_opt\_model$.predict($features\_pred\_subspace$)
4:  $valid\_found \leftarrow$ False, $idx \leftarrow 0$
5:  $verified\_subspace \leftarrow \emptyset$
6:  while not $valid\_found$ do
7:      $adapt\_opt \leftarrow rankeding[idx]$
8:      $Analyzer$.verifyAdaptOpt($adapt\_opt$)          ▷ Knowledge
9:      $\_, qualities \leftarrow K.verification\_results[idx]$
10:     if $qualities$ meet all threshold goals then
11:         $valid\_found \leftarrow$ True
12:     end if
13:     $verified\_subspace$.add($adapt\_opt$)
14:     $idx \leftarrow idx + 1$
15: end while
16: $unselected \leftarrow K.adaptation\_options \setminus verified\_subspace$
17: $explore \leftarrow unselected$.randomSelect($exploration\_rate$)
18: $Analyzer$.verifyAdaptOpt($explore$)               ▷ Knowledge
19: $features, qualities \leftarrow K.verification\_results$
20: Update threshold models (line 11 to 13 from Algorithm 1)
21: $DL\_opt\_model$.update($features, qualities$)
```

The **optimization approach** starts from the adaptation options selected by the threshold approach. Algorithm 2 shows the integrated approach that applies to the different types of goals.

Just as in algorithm 1, in line 1, we predict the relevant subspace for the threshold goals. Then, in line 2 to line 3, the optimization deep learning model predicts a ranking of the relevant adaptation options, i.e., the *predict()* method on line 3 outputs the adaptation options in ranked order based on the optimization goal.[4] In lines 4 to 18

---

[4] Our approach relies on goals defined as rules with only a single optimization goal. Multi-objective optimization is beyond the scope and subject of future research.

we iterate over the ranked adaptation options in descending order of the predicted value for the quality of the optimization goal. We verify an adaption option and check whether it complies with the threshold goals. If this is the case we select this option for adaptation. If not, we continue verifying adaptation options until one is found that satisfies the threshold goals. Then, we randomly select a sample of adaptation options from the options that were not verified based on the *exploration_rate*. These unexplored options are then also verified. All the verification results are stored in the knowledge. Next, in lines 19 to 21 the deep learning models are updated, exploiting the verification results.

Note that in the case where we only consider threshold goals (the threshold approach), the adaptation space reduction occurs prior to verification. If we consider both threshold goals and an optimization goal, the adaptation space reduction is conducted by cleverly and efficiently verifying adaptation options, i.e., in the order of their predicted ranking with respect to the optimization goal until an option is found that satisfies all the threshold goals.

5.3 DLASeR's Neural Network Architecture

Technically, we use a neural network with multiple nonlinear hidden layers [7, 16, 29]. Non-linearity refers to the functions that determine the flow in the network. Non-linear layers help capturing the complex uncertainties of the problem at hand. Concretely, for the activation function we apply the rectified linear unit (ReLU). This function has proven to greatly accelerate the convergence of stochastic gradient descent, compared to the sigmoid/tanh functions [27]. As activation function in the last layer we apply a sigmoid and linear activation in the case of classification and regression respectively. For the classification neural network, we want as output either 1 or 0, representing meeting and not meeting the threshold goal respectively. For the regression neural network, we want an output value that corresponds to the quality of the property we want to optimize (for DeltaIoT this is the value of the energy consumption). Sigmoid is desirable for binary classification, since it produces a value in the range 0 to 1. Linear activation can output any value and is the standard activation function for the output layer in the case of regression. To prevent overfitting, we apply regularization. On the first layer we apply L1 regularization, enforcing some feature selection by the neural network [34]. On the other layers we apply L2 regularization that incorporates the squared sum of the weights, ensuring that the weights will not become (very) large, reducing overfitting. Finally, before the final classification or regression head we employ a dropout with a fraction of 10% [39]. These regularization techniques make the models more robust.

6 MAPE ARCHITECTURE WITH DLASER

Fig. 7 shows how the deep neural network learners of DLASeR are integrated in the MAPE-K architecture.

We focus here on the Analyzer, Planner, and the relevant models of Knowledge. When the analyzer is triggered (1), it collects the possible adaptation options (2), i.e., all possible configurations of the managed system that are reachable through adaptation from the current configuration. Then the threshold deep learner determines the relevant set of adaptation options (3.1-2). These options are then written to the knowledge repository (4) and the planner is triggered (5). The planner reads the relevant options together with an exploration sample (6). The optimization learner is then invoked to rank the options (7.1-2). The options are then verified one by one for the all the goals in order of ranking (8.1-2). As soon as an option is found that complies with the threshold goals, the option will be selected for adaptation. Finally, the verification results are exploited to update the deep learning models (9.1-2). After that, a plan is generated for the selected adaptation option and the executor is invoked to apply the adaptation (10-11). The realization of these last two steps are outside the scope of this paper.

7 EVALUATION

We evaluate DLASeR using the DeltaIoT artifact. We used the DeltaIoT simulator, as experimentation with the real physical setup is time consuming. We start with the explaining the evaluation setup. Then, we report the results of the offline stages, the results for threshold goals only and then the results for both types of goals. The section concludes with a discussion of validity threats. All evaluation material is available at the project website [10].

7.1 Evaluation Setup

We evaluate DLASeR on two instances of DeltaIoT (Section 2.1). For both instances we use the same settings. The uncertainty profiles for the traffic loads ranged from 0 to 10 messages per mote per cycle. The network interference varied between -40 dB and +15 dB. The MAPE-K feedback loop was designed using a network of timed automata. These models were executed by using the ActivFORMS execution engine [18, 20]. We applied runtime statistical model checking using Uppaal-SMC for the verification of adaptation options [8]. The exploration rate was set to 5 %.

For both instances we consider 275 online adaptation cycles, corresponding with a wall clock time of 77 hours. We only used a single cycle to initialize the network parameters. The remaining 274 cycles are evaluated as learning cycles. We use as a benchmark for the threshold goals the approach proposed by Quin et al. [37]. To evaluate the quality of the adaptation decisions made by the learning approach, we benchmark the results with a reference approach that analyses the whole adaptation space without learning.

For the learning solutions, we used the implementations of scalers from *scikit-learn* [35] and neural networks from *Keras* and *Tensorflow* [1]. We run the simulated self-adaptive system on an i7-3770, 3.40GHz, 12GB RAM; the deep learning models are trained (and maintained) on a NVIDIA Tesla T4 GPU with 12 GB RAM.

7.2 Offline settings

As explained in Section 5, we used grid search for selecting models for the learners. Table 1 shows the best parameters of the pipeline obtained by evaluating the approaches on 30 sequential cycles. We note that the models can predict latency significantly better than packet loss. We speculate that this is due to the fact that the latency goal is less restrictive than the packet loss goal. On average the percentage of adaptation options that meet the latency goal lies around 90% for DeltaIoTv1 and 80% for DeltaIoTv2. Whereas the packet loss goal is satisfied respectively by ca. 40% and 20% of the adaptation options. On the other hand, when scaling up to the larger instance, we observe that the F1-score of the packet loss degrades with 10.37%, whereas the F1-score of the latency improves with 2.09%. For the optimization goal, we notice a large decay in Spearman

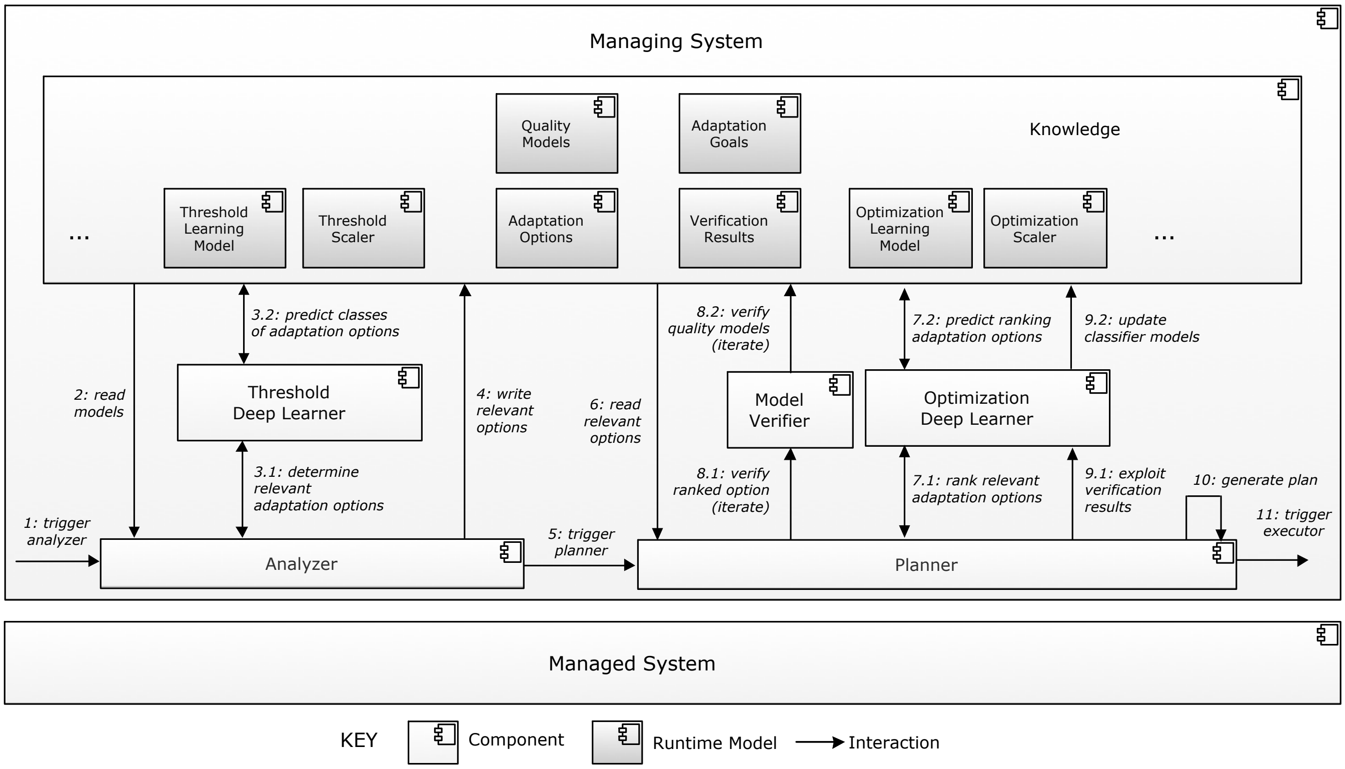

Managing System
Knowledge
Quality Models
Adaptation Goals
...
Threshold Learning Model
Threshold Scaler
Adaptation Options
Verification Results
Optimization Learning Model
Optimization Scaler
...
3.2: predict classes of adaptation options
8.2: verify quality models (iterate)
7.2: predict ranking adaptation options
9.2: update classifier models
2: read models
Threshold Deep Learner
4: write relevant options
6: read relevant options
Model Verifier
Optimization Deep Learner
3.1: determine relevant adaptation options
8.1: verify ranked option (iterate)
7.1: rank relevant adaptation options
9.1: exploit verification results
10: generate plan
1: trigger analyzer
Analyzer
5: trigger planner
Planner
11: trigger executor
Managed System
KEY
Component
Runtime Model
Interaction

Figure 7: Runtime architecture of the deep learning approach to reduce large adaptation spaces. The threshold deep learner and optimization deep learner components together with their models are determined during the offline activities as shown in Figure 5. The runtime activities within the two learners follows the workflow of online activities shown in Figure 6.

Table 1: Best grid search results for the threshold and optimization goals.

Problem	Goal	Hyper parameters					Objective
	Threshold	***Scaler***	***Batch size***	***Learning rate***	***Optimizer***	***Hidden layers***	***F1-score***
DeltaIoTv1	Packet loss	Standard	128	2e-3	Adam	[20, 10, 5]	83.18%
	Latency	MaxAbs	64	5e-4	Adam	[20, 10, 5]	93.96%
DeltaIoTv2	Packet loss	Standard	2056	5e-3	Adam	[50, 25, 10, 5]	75.81%
	Latency	Standard	1028	2e-3	Nadam	[50, 25, 10, 5]	96.05%
	Optimization	***Scaler***	***Batch size***	***Learning rate***	***Optimizer***	***Hidden layers***	***Spearman's rho***
DeltaIoTv1	Energy consumption	Standard	32	5e-3	RMSprop	[20, 10, 5]	87.63%
DeltaIoTv2	Energy consumption	MaxAbs	1028	5e-4	Nadam	[50, 25, 10, 5]	26.04%

correlation that is reduced from 87.63% for DeltaIoTv1 to 26.04% for DeltaIoTv2. At first sight, this might seem problematic. But, further analysis will show that this difference does not lead to significantly worse results. Based on the obtained configurations, we deploy the models in the simulator to perform adaptation space reduction.

7.3 Results

Table 2 presents the results obtained with DLASeR. For the threshold goals (T1 & T2) we obtain a reduction of the adaptation space of 64.42% for DeltaIoTv1 and 92.24% for DeltaIoTv2. The F1 scores are excellent, but, decrease from 85.78% to 73.06% when scaling up to the larger instance. DLASeR achieves a good reduction of the adaptation space for the small instance of DeltaIoT (64.42%) and an excellent result for the large instance (92.94%).

Table 2: Results for DLASeR & comparison with Quin et al.

Problem	Approach	Goals	F1 score	AASR
DeltaIoTv1	DLASeR	T1 & T2	85.78%	64.42%
	DLASeR	T1 & T2 & O1	/	**99.20%**
	Quin F. et al	T1 & T2	**85.85%**	50%
DeltaIoTv2	DLASeR	T1 & T2	**73.06%**	92.94%
	DLASeR	T1 & T2 & O1	/	**98.53%**
	Quin F. et al	T1 & T2	73.01%	75.5%

We compare these results with the work of Quin et al. [37], where classification was used for adaptation space reduction on identical cases. The results show that both approaches achieve similar results for F1 score. However, there is a significant difference for the adaptation space reduction (AASR): for DeltaIoTv1 we have 64.42% for DLASeR versus 50.00 % for Quin et al.; for DeltaIoTv2 we have 92.94% for DLASeR versus 75.50 % for Quin et al.

The results for settings with the three goals (threshold goals T1 & T2, and optimization goal O1) demonstrate the effectiveness of DLASeR. With 99.20% for DeltaIoTv1 and 98.53% for DeltaIoTv2, DLASeR comes close to the optimum of what can be achieved in terms of adaptation space reduction. Figure 8 visualizes the predicted adaptation space reduction for DeltaIoTv2 at one particular point in time. The red lines in the figure indicate the region with relevant adaptation options according to the threshold goals.

While the F1 score and AASR give us insight into the adaptation space reduction, these metrics do not tell us anything about the quality of the adaptation decisions made with reduced adaptation spaces. We measure this and benchmark DLASeR with a reference approach that applies exhaustive verification (which is the ideal case, but practically not always possible due to time constraints).

Figure 9 shows the qualities of the system for only the threshold goals. The results show no visible difference between the qualities for DLASeR and the reference approach. For DeltaIoTv1, the p-values for the different qualities are between 0.15 and 0.93 (Wilcoxon signed-rank test). For DeltaIoTv2, the tests indicate that there is a small difference for all qualities (all p-values lower than 0.014). For example, the energy consumption for DLASeR is 0.07% higher, but this is from a practical point of view negligible. The results are in line with the results of Quin et al. [37].

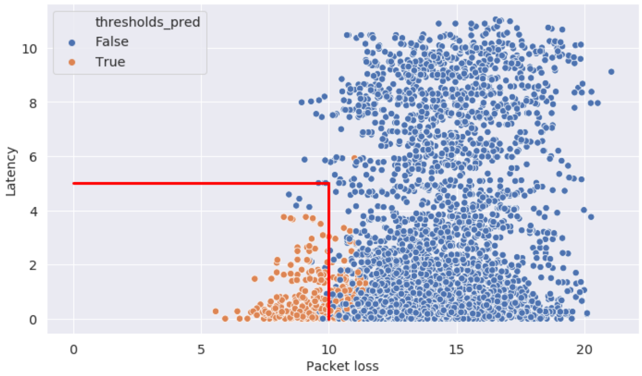
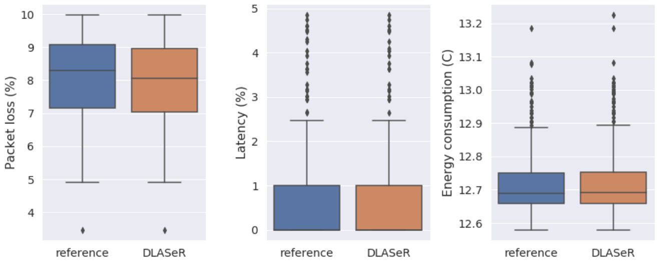
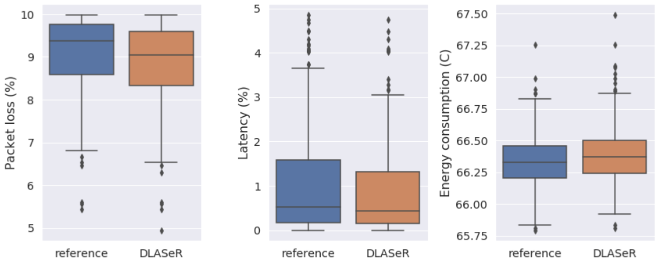


Figure 8: The predicted adaptation space reduction for some cycle in DeltaIoTv2. The orange and blue dots represent respectively the adaption options that are predicted to meet and not meet the threshold goals T1 and T2.

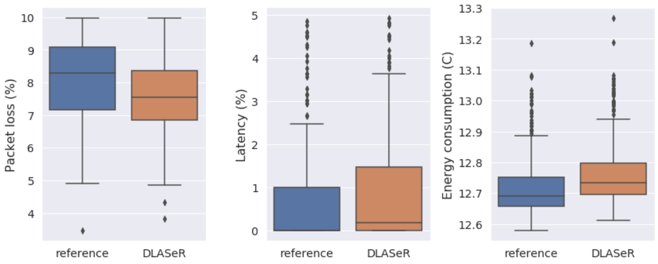


(a) DeltaIoTv1

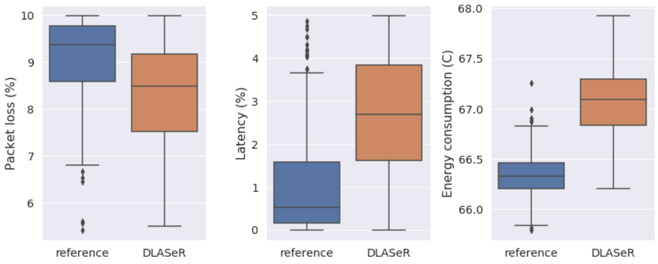


(b) DeltaIoTv2

Figure 9: Quality of adaptation decision for threshold goals.



(a) DeltaIoTv1



(b) DeltaIoTv2

Figure 10: Quality adaptation decisions for both goal types

Figure 10 shows the qualities of the system for two threshold goals and an optimization goal, both for DeltaIoTv1 and DeltaIoTv2. The reference approach again exhaustively verifies all adaptation options to make adaptation decisions. The results show that the threshold goals for packet loss and latency are always satisfied with DLASeR. Regarding the optimization goal, the total energy consumption of DLASeR is respectively 0.34% and 1.11% higher for DeltaIoTv1 and DeltaIoTv2. compared to the reference approach. This is a small cost for the dramatic improvement of adaptation time. For DeltaIoTv2, DLASeR requires on average 50.53 seconds for verification plus 0.92 seconds for online learning training, compared to 16.73 minutes with the reference approach (which is in practice infeasible since it exceeds the cycle time of 9.5 minutes).

Figure 11 visualizes the ranked adaptation space at a point in time for such a setting. The colors indicate the ranking based on predicted energy consumption.

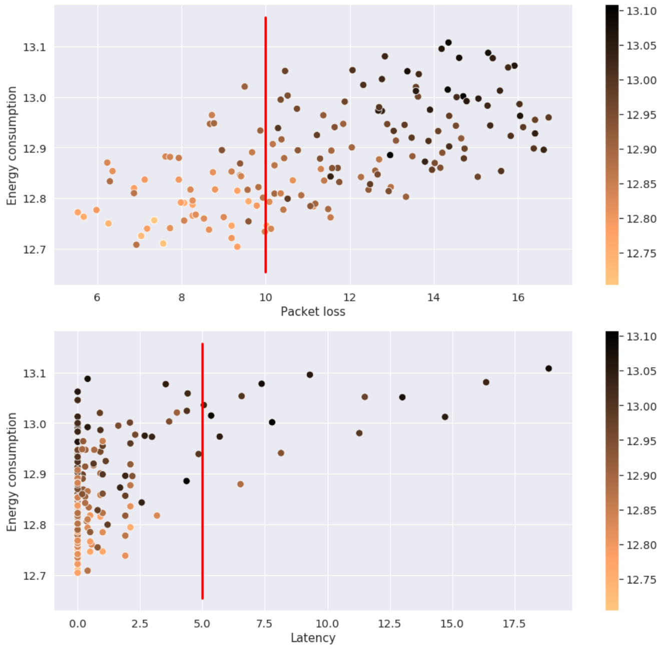


Figure 11: Example adaptation space with three goals.

7.4 Threats to Validity

The evaluation of DLASeR is subject to a number of validity threats. We evaluated the approach only in one domain, so we cannot generalize the conclusions (external validity). We mitigated this threat to some extent by using two different IoT networks (based on topology and adaptation space). However, more extensive evaluation in different domains is required to increase the validity of the results.

The presented approach considers only a single optimization goal. Additional research will be required to extend the proposed approach for multi-objective optimization goals.

We have evaluated DLASeR using different metrics. Nevertheless, the specifics of the setting of the applications we used, in particular the graph structure of the network, the uncertainties as well as the specific goals that we considered may have an effect on how difficult or easy the problem of adaptation space reduction may be (internal validity). To mitigate this threat to some extent we applied DLASeR to a setting of a real IoT deployment that was developed in collaboration with an industry partner in IoT.

For practical reasons, we evaluated DLASeR in simulation. The simulator we used applies particular forms of randomness to represent uncertainties in the problem settings. This may cause a threat that the results may be different if the study would be repeated (reliability). To minimize this threat, we evaluated DLASeR over a period of several days. Furthermore, the complete package we used for evaluation is available to replicate the study.

8 RELATED WORK

Over the past few years, we observe a growing interest in the use of machine learning and search-based techniques in self-adaptation. Here we discuss only a representative set of approaches most closely related to adaptation space reduction.

Elkhodary et al. [11] apply learning in their FUSION framework to grasp the impact of adaptation decisions on the system's goals. Concretely, they utilize M5 decision trees to learn the utility functions that are associated with the qualities of the system. Learning is enabled by a dynamic feature-oriented representation of the system. As our work, FUSION results in a significant improvement in analysis. Similarly to [31], the FUSION framework targets the feature selection space, whereas we target the configuration space.

Quin F. et al. [37] apply machine learning for the purpose of large adaptation space reduction. In their work, only threshold goals were considered and the evaluation was conducted with a focus on linear models. We research the effectiveness of non-linear deep learning models on both threshold and optimization goals.

Metzger et al. [31] combine feature models with online learning to explore the adaptation space of self-adaptive systems. The authors show that exploiting the hierarchical structure and the difference in feature models over different cycles leads to a speedup in convergence of the learning process. Our work complements this work by focusing on a configuration space.

Jamshidi et al. [22] identify a set of Pareto optimal configuration offline. During operation an adaption plan is generated using this set of configurations. Reducing the adaptation space ensures that they can still use PRISM [28] at runtime to quantitatively reason about adaptation decisions. Our approach is more versatile by reducing the adaptation space at runtime in a dynamic and learnable fashion.

Nair et al. [33] present FLASH that sequentially explores the configuration space by reflecting on the configurations evaluated so far to determine the next best configuration to explore. Other search-based approaches [17], such as Chen et al. [5] that considers multi-objective optimization, have been studied to deal with decision-making in self-adaptive systems with large adaptation spaces.

9 CONCLUSIONS

To tackle the research question: "How to reduce large adaptation spaces of self-adaptive systems with multiple threshold goals and an optimization goal effectively?" this paper contributes DLASeR, short for Deep Learning for Adaptation Space Reduction.

The core idea of DLASeR is to apply what can be called "lazy verification." In particular, the adaptation space is first reduced using a deep learner for the threshold goals. The selected adaptation options are then ranked using a deep learner for the optimization goal. Only then verification starts, that is, the adaptation options are verified one by one in the order of their predicted ranking until an option is found that complies with the threshold goals.

The results show that DLASeR effectively reduces the adaptation space with negligible effect on the the realization of the adaptation goals. If we only consider threshold goals, DLASeR improves also over a state of the art approach that uses classification.

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. We also plan to look into different types of goals, in particular set point goals and multi-objective optimization goals. In the mid term, we plan to look into the use of machine learning in support of self-adaptation in a decentralized setting. 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 with machine learning.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, 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] G. Agha and K. Palmskog. 2018. A Survey of Statistical Model Checking. *ACM Trans. Model. Comput. Simul.* 28, 1, Article 6 (Jan. 2018), 39 pages. https://doi.org/10.1145/3158668

[3] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. 2011. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering* 37, 3 (May 2011), 387–409. https://doi.org/10.1109/TSE.2010.92

[4] T. Chen and R. Bahsoon. 2017. Self-Adaptive and Online QoS Modeling for Cloud-Based Software Services. *IEEE Transactions on Software Engineering* 43, 5 (May 2017), 453–475. https://doi.org/10.1109/TSE.2016.2608826

[5] T. Chen, K. Li, R. Bahsoon, and X. Yao. 2018. FEMOSAA: Feature-Guided and Knee-Driven Multi-Objective Optimization for Self-Adaptive Software. *ACM Trans. Softw. Eng. Methodol.* 27, 2, Article Article 5 (June 2018), 50 pages. https://doi.org/10.1145/3204459

[6] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26. https://doi.org/10.1007/978-3-642-02161-9_1

[7] CMU. 2020. Neural Networks: What can a network represent. In *Deep Learning, Spring Course Carnegie Mellon University*. https://deeplearning.cs.cmu.edu/document/slides/lec2.universal.pdf

[8] A. David, K. Larsen, A. Legay, M. Mikučionis, and D. Poulsen. 2015. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* 17, 4 (2015), 397–415.

[9] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Camara, R. Calinescu, M. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J. Jezequel, S. Malek, R. Mirandola, M. Mori, H. Müller, R. Rouvoy, C. Rubira, E. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. Villegas, T. Vogel, and F. Zambonelli. 2017. Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese (Eds.). Springer International Publishing, Cham, 3–30.

[10] J. Van Der Donckt, D. Weyns, and F. Quin. 2020. DLASeR website. https://people.cs.kuleuven.be/danny.weyns/material/2020SEAMS/DLASeR/

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

[12] N. Esfahani, E. Kouroshfar, and S. Malek. 2011. Taming Uncertainty in Self-adaptive Software. In *19th Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 234–244. https://doi.org/10.1145/2025113.2025147

[13] A. Filieri, C. Ghezzi, and G. Tamburrelli. 2011. Run-time efficient probabilistic model checking. In *33rd International Conference on Software Engineering*. 341–350. https://doi.org/10.1145/1985793.1985840

[14] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* 37, 10 (Oct. 2004), 46–54. https://doi.org/10.1109/MC.2004.175

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

[16] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep learning.* MIT press.

[17] M. Harman, P. McMinn, J. de Souza, and S. Yoo. 2012. *Search Based Software Engineering: Techniques, Taxonomy, Tutorial.* Springer-Verlag, 1–59.

[18] U. Iftikhar, J. Lundberg, and D. Weyns. 2016. A model interpreter for timed automata. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 243–258.

[19] U. Iftikhar, G. S. Ramachandran, P. Bollansée, D. Weyns, and D. Hughes. 2017. DeltaIoT: A self-adaptive Internet of Things Exemplar. In *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* IEEE Press, 76–82.

[20] U. Iftikhar and D. Weyns. 2014. ActivFORMS: Active Formal Models for Self-Adaptation *(SEAMS 2014)*. Association for Computing Machinery, New York, NY, USA, 125–134. https://doi.org/10.1145/2593929.2593944

[21] S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[22] P. Jamshidi, J. Cámara, B. Schmerl, C. Käestner, and D. Garlan. 2019. Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 39–50. https://doi.org/10.1109/SEAMS.2019.00015

[23] J. Kephart and D. Chess. 2003. The vision of autonomic computing. *Computer* 1 (2003), 41–50.

[24] C. Kinneer, Z. Coker, J. Wang, D. Garlan, and C. Goues. 2018. Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search *(SEAMS '18)*. Association for Computing Machinery, New York, NY, USA, 40–50. https://doi.org/10.1145/3194133.3194145

[25] C. Klein, M. Maggio, K. Årzén, and F. Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *36th International Conference on Software Engineering.* ACM, 700–711.

[26] J. Kramer and J. Magee. 2007. Self-Managed Systems: An Architectural Challenge. In *Future of Software Engineering.* https://doi.org/10.1109/FOSE.2007.19

[27] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.* 1097–1105.

[28] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 585–591.

[29] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521 (2015), 436–444.

[30] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns. 2017. A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive Systems With Multiple Quality Requirements. In *Managing Trade-Offs in Adaptable Software Architectures*, I. Mistrik, N. Ali, R. Kazman, J. Grundy, and B. Schmerl (Eds.). Morgan Kaufmann, Boston, 45 – 77. https://doi.org/10.1016/B978-0-12-802855-1.00003-4

[31] A. Metzger, C. Quinton, Z. Mann, L. Baresi, and K. Pohl. 2019. Feature-Model-Guided Online Learning for Self-Adaptive Systems. *arXiv preprint arXiv:1907.09158* (2019).

[32] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. 2015. Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *Foundations of Software Engineering.* ACM, 1–12. https://doi.org/10.1145/2786805.2786853

[33] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2018. Finding Faster Configurations using FLASH. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2870895

[34] A. Ng. 2004. Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In *21th International Conference on Machine Learning.* ACM, 78.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12 (2011), 2825–2830.

[36] D. Perez-Palacin and R. Mirandola. 2014. Uncertainties in the Modeling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation *(ICPE '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2568088.2568095

[37] F. Quin, D. Weyns, T. Bamelis, S. B. Sarpreet, and S. Michiels. 2019. Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning. In *14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* IEEE, 1–12.

[38] M. Shanker, M. Hu, and M. Hung. 1996. Effect of data standardization on neural network training. *Omega* 24, 4 (1996), 385–397.

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[40] D. Weyns. 2019. Software Engineering of Self-adaptive Systems. In *Handbook of Software Engineering.* Springer, 399–443.

[41] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. 2017. Perpetual Assurances for Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems III. Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese (Eds.). Springer International Publishing, Cham, 31–63.

[42] D. Weyns and M. U. Iftikhar. 2016. Model-Based Simulation at Runtime for Self-Adaptive Systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC).* 364–373.

[43] D. Weyns and U. Iftikhar. 2019. ActivFORMS: A Model-Based Approach to Engineer Self-Adaptive Systems. arXiv:cs.SE/1908.11179

[44] D. Weyns, U. Iftikhar, and J. Söderlund. 2013. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In *International Symposium on Software Engineering of Self-Managing and Adaptive Systems (SEAMS '13).*

[45] D. Weyns, S. Malek, and J. Andersson. 2012. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems* 7, 1 (2012), 8:1–8:61. https://doi.org/10.1145/2168260.2168268

[46] D. Weyns, G.S. Ramachandran, and R.K. Singh. 2018. Self-managing Internet of Things. In *International Conference on Current Trends in Theory and Practice of Informatics.* Springer, 67–84.

[47] D. Zwillinger and S. Kokoska. 2000. CRC Standard Probability and Statistics Tables and Formulae, Section 14.7. In *Chapman & Hall.*