Master Thesis

Applying Artificial Neural Networks to Reduce the Adaptation Space in Self-Adaptive Systems
- an exploratory work

August 7, 2019

Author: Sarpreet Singh Buttar
Supervisor: Prof. Danny Weyns
Examiner: Rafael Martins
Semester: Spring 2019
Subject: Computer Science
Abstract

Self-adaptive systems have limited time to adjust their configurations whenever their adaptation goals, i.e., quality requirements, are violated due to some runtime uncertainties. Within the available time, they need to analyze their adaptation space, i.e., a set of configurations, to find the best adaptation option, i.e., configuration, that can achieve their adaptation goals. Existing formal analysis approaches find the best adaptation option by analyzing the entire adaptation space. However, exhaustive analysis requires time and resources and is therefore only efficient when the adaptation space is small. The size of the adaptation space is often in hundreds or thousands, which makes formal analysis approaches inefficient in large-scale self-adaptive systems. In this thesis, we tackle this problem by presenting an online learning approach that enables formal analysis approaches to analyze large adaptation spaces efficiently. The approach integrates with the standard feedback loop and reduces the adaptation space to a subset of adaptation options that are relevant to the current runtime uncertainties. The subset is then analyzed by the formal analysis approaches, which allows them to complete the analysis faster and efficiently within the available time. We evaluate our approach on two different instances of an Internet of Things application. The evaluation shows that our approach dramatically reduces the adaptation space and analysis time without compromising the adaptation goals.

Keywords: Self-Adaptive Systems, Self-Adaptation, Architecture-Based Adaptation, Autonomous Systems, Cyber-Physical Systems, CPS, DeltaIoT, IoT, ActivFORMS, MAPE-K Feedback Loop, Runtime Uncertainties, Adaptation Space, Analysis, Machine Learning, Artificial Neural Network, ANN, Online Learning, Deep Learning, Online Supervised Learning, Incremental Learning, Classification, Multi-Layer Perceptron, MLP.
Preface

First of all, I thank my supervisor, Prof. Danny Weyns, for allowing me to work on this interesting problem. I am grateful for the guidance that he provided me throughout this period. Then, I would like to thank Jesper Andersson and Morgan Ericsson for providing the computing resources. I am also very thankful to Francesco Flammini for providing necessary instructions regarding the thesis. I also thank to Muhammad Usman Iftikhar for helping me to understand the research problem. This thesis would not be possible without their support. I would also like to thank my teachers – Nadeem Abbas, Mauro Caporuscio, Ola Petersson, and Rüdiger Lincke, for giving me valuable lessons during my master’s studies.
### Contents

1 **Introduction**  
   1.1 Research Problem  
   1.2 Research Question  
   1.3 Motivation  
   1.4 Limitations  
   1.5 Contribution  
   1.6 Target Group  
   1.7 Outline

2 **Background**  
   2.1 Architecture-Based Adaptation  
   2.2 Active FORmal Models for Self-Adaptation  
   2.3 Machine Learning  
   2.3.1 Supervised Learning  
   2.3.2 Classification  
   2.3.3 Artificial Neural Networks  
   2.3.4 Multi-Layer Perceptron

3 **DeltaIoT: Internet of Things Application**  
   3.1 DeltaIoT.v1  
   3.2 DeltaIoT.v2  
   3.3 Self-Adaptive DeltaIoT

4 **Method**  
   4.1 Controlled Experiment  
   4.2 Reliability and Validity

5 **Our Learning Approach**  
   5.1 Architecture  
   5.2 Offline and Online Activities  
   5.2.1 Offline Activities  
   5.2.2 Online activities

6 **Implementation**  
   6.1 DeltaIoT.v1  
   6.2 DeltaIoT.v2

7 **Results and Analysis**  
   7.1 DeltaIoT.v1  
   7.2 DeltaIoT.v2  
   7.3 Conclusion of Results

8 **Related Work**

9 **Conclusion**  
   9.1 Future Work

References
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An overview of the adaptation space in DeltaIoT.v2 at a particular adaptation cycle</td>
</tr>
<tr>
<td>1.2</td>
<td>Impact of the runtime uncertainties on the qualities of one adaptation option over six particular adaptation cycles in DeltaIoT.v2</td>
</tr>
<tr>
<td>2.3</td>
<td>A self-adaptive system with architecture-based adaptation</td>
</tr>
<tr>
<td>2.4</td>
<td>Architecture of ActivFORMS</td>
</tr>
<tr>
<td>2.5</td>
<td>An overview of the sub-branches in ML [1]</td>
</tr>
<tr>
<td>2.6</td>
<td>An overview of the concept of the supervised learning compare to the traditional programming approach [2]</td>
</tr>
<tr>
<td>2.7</td>
<td>Multi-Layer Perceptron</td>
</tr>
<tr>
<td>2.8</td>
<td>An artificial neuron</td>
</tr>
<tr>
<td>2.9</td>
<td>Ranges of the activation functions</td>
</tr>
<tr>
<td>3.10</td>
<td>Physical deployment and structure of DeltaIoT.v1 at KU Leuven Campus, Belgium [23]</td>
</tr>
<tr>
<td>3.11</td>
<td>An overview of the profiles of runtime uncertainties over particular links</td>
</tr>
<tr>
<td>3.12</td>
<td>Physical deployment of DeltaIoT.v2 at KU Leuven Campus, Belgium</td>
</tr>
<tr>
<td>3.13</td>
<td>Physical structure of DeltaIoT.v2 at KU Leuven Campus, Belgium</td>
</tr>
<tr>
<td>3.14</td>
<td>Basic architecture of DeltaIoT with the no learning approach</td>
</tr>
<tr>
<td>5.15</td>
<td>Generic architecture of our approach</td>
</tr>
<tr>
<td>5.16</td>
<td>Generic working flow of the learning models to find relevant adaptation options</td>
</tr>
<tr>
<td>5.17</td>
<td>Offline and online activities in our approach</td>
</tr>
<tr>
<td>6.18</td>
<td>Performance of the learning models with the selected hyper-parameters and scalers for the selection of the training cycles in DeltaIoT.v1</td>
</tr>
<tr>
<td>6.19</td>
<td>An overview of the concrete instantiation of our learning approach on DeltaIoT.v1</td>
</tr>
<tr>
<td>6.20</td>
<td>Performance of the learning models with the selected hyper-parameters and scalers for the selection of the training cycles in DeltaIoT.v2</td>
</tr>
<tr>
<td>6.21</td>
<td>An overview of the concrete instantiation of our learning approach on DeltaIoT.v2</td>
</tr>
<tr>
<td>7.22</td>
<td>Adaptation results of DeltaIoT.v1 with the learning and no learning approaches</td>
</tr>
<tr>
<td>7.23</td>
<td>An overview of the selection of the relevant adaptation options in DeltaIoT.v1 with the learning approach in one particular adaptation cycle</td>
</tr>
<tr>
<td>7.24</td>
<td>Adaptation results of DeltaIoT.v2 with the learning and no learning approaches</td>
</tr>
<tr>
<td>7.25</td>
<td>An overview of the selection of the relevant adaptation options in DeltaIoT.v2 with the learning approach in one particular adaptation cycle</td>
</tr>
<tr>
<td>1.26</td>
<td>Time measurements of the learning models used in DeltaIoT.v1 (left) and DeltaIoT.v2 (right)</td>
</tr>
<tr>
<td>1.27</td>
<td>Time measurements of the offline activities in DeltaIoT.v1 (left) and DeltaIoT.v2 (right)</td>
</tr>
<tr>
<td>1.28</td>
<td>An example of the feature scaling in one particular adaptation cycle in DeltaIoT.v1</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Independent and dependent variables in our controlled experiments 19
6.2 An overview of the dataset made for the packet loss learning model 25
6.3 An overview of the dataset made for the latency learning model . . 26
6.4 Grid search results for the selection of the activation function and neurons in the first hidden layer for the packet loss learning model in DeltaIoT.v1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
6.5 Grid search results for the selection of the activation function and neurons in the first hidden layer for the latency learning model in DeltaIoT.v1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
6.6 Grid search results for the selection of the hidden layers for the packet loss learning model in DeltaIoT.v1 . . . . . . . . . . . 27
6.7 Grid search results for the selection of the hidden layers in the latency learning model for DeltaIoT.v1 . . . . . . . . . . . . . . 27
6.8 Summary of the selected hyper-parameters and scalers in DeltaIoT.v1 28
6.9 Summary of the selected hyper-parameters and scalers in DeltaIoT.v2 31
Listings

1 An overview of the runtime data in one particular adaptation cycle in DeltaIoT.v1 ........................................... 24
2 Training the learning models at runtime ........................................... 29
3 Selection of the relevant adaptation options by applying the learning models at runtime ........................................... 30
1 Introduction

We live in a world where the involvement of modern software systems in human activities is rapidly increasing. The software systems have to deal with various complex issues while managing human activities. The most challenging issue for the software systems is to handle uncertainties that appear during runtime. For instance, dynamic user goals, changing the availability of external components, fluctuating traffic load in the Internet of Things (IoT) or unstable conditions of the operating environment. Often, these uncertainties are unpredictable, which may affect the behavior of the software systems. Therefore, to tackle the runtime uncertainties, software engineers have proposed one prominent approach called self-adaptation [3, 4, 5].

Self-adaptation enables a software system to autonomously handle the runtime uncertainties to achieve specific adaptation goals, i.e., the quality requirements that are subject to adaptation. Software systems with such behavior are known as self-adaptive systems [6]. There are a number of approaches to realize self-adaptation such as architecture-based adaptation [7, 8, 9, 10, 11, 12], control-based software adaptation [13], and self-aware computing [14]. In this thesis, we focus on architecture-based adaptation that realizes self-adaptation by adding an external feedback loop to the software system. The feedback loop continuously monitors the software system and adapt its configurations whenever the runtime uncertainties violate the adaptation goals.

Runtime Quantitative Verification (RQV) [15, 16] and Active FORmal Models for Self-adaptation (ActivFORMS) [17] are two of the existing formal analysis approaches for architecture-based adaptation. Whenever the runtime uncertainties violate the adaptation goals, these approaches analyze all the possible adaptation options to determine the best one from them. Here, adaptation options refer to a set of possible configurations, which is also known as adaptation space. The best adaptation option is then applied to the software system to accomplish the adaptation goals.

Due to exhaustive analysis, the formal analysis approaches provide guarantees to achieve the adaptation goals. However, they are effective to only those self-adaptive systems that have smaller adaptation space. Self-adaptive systems often have large adaptation spaces, i.e., thousands of adaptation options, and limited time to adapt themselves. Therefore, exhaustively analyzing the large adaptation space may be infeasible, since it requires time and resources [5, 18, 19, 20]. Hence, the formal analysis approaches are inefficient to analyze large adaptation space.

In this thesis, we present an approach that enables the formal analysis approaches to efficiently analyze large adaptation spaces while having the same quality guarantees. The approach is integrated into the standard feedback loop and uses Artificial Neural Networks (ANN) [21, 22] to learn the adaptation space on-the-fly, i.e., online learning. Learning mechanism helps to reduce the adaptation space to a subset of adaptation options that are relevant to the current runtime uncertainties. The subset is then further analyzed by the formal analysis approaches. We evaluate our approach on two different instances of an IoT application named as DeltaIoT [23] and compare the results with an existing formal analysis approach called ActivFORMS. The results show that our approach dramatically reduces the adaptation space and analysis time without compromising
the adaptation goals.

1.1 Research Problem

Every self-adaptive system has an adaptation space that contains all the possible adaptation options. Whenever the runtime uncertainties violate the adaptation goals, it has limited time to adapt itself. Within the available time, it needs to analyze the adaptation space to find the best adaptation option that can achieve the adaptation goals. The adaptation space has a dynamic behavior, i.e., the qualities of each adaptation option change according to the current runtime uncertainties. Therefore, the adaptation goals can only be achieved if the best adaptation option is selected and applied within the available time. Existing formal analysis approaches such as ActivFORMS determine the best adaptation option by exhaustively analyzing the entire adaptation space. In particular, it uses statistical model checking techniques [24, 25] to predict the expected qualities of each adaptation option. However, exhaustive analysis on large adaptation space, i.e., thousands adaptation options, within limited time, may not be feasible. Thus, the formal analysis approaches are only efficient in small-scale self-adaptive systems [5, 18, 19, 20]. In this thesis, we aim to apply ANN to reduce the adaptation space to a subset of adaptation options that are relevant to the current runtime uncertainties. The reduced adaptation space would enable formal analysis approaches to deal with large-scale self-adaptive systems efficiently.

1.2 Research Question

We formulate the research problem into the following research question:

Can we use ANN to reduce the adaptation space in self-adaptive systems without compromising the adaptation goals?

Concretely, we aim to apply ANN in such a way that they can reduce the adaptation space to only those adaptation options that are relevant to the current runtime uncertainties. It would allow formal analysis approaches to efficiently deal with large adaptation spaces by only analyzing the relevant adaptation options.

1.3 Motivation

DeltaIoT.v2, an instance of an IoT application that we use for evaluation, (see Section 3.2), has a large adaptation space, i.e., 4096 adaptation options, and 9.5 minutes to analyze it. In this case, applying formal analysis approaches to analyze the entire adaptation space is not feasible. For instance, ActivFORMS is not able to analyze the entire adaptation space in 9.5 minutes. For detailed results, see Section 7. Therefore, to perform efficient analysis, only the adaptation options, which are relevant to the current runtime uncertainties should be analyzed. For example, Figure 1.1 shows an overview of the adaptation space in DeltaIoT.v2 at a particular adaptation cycle. The orange dots represent the adaptation options, which are irrelevant to the current runtime uncertainties. Whereas, the green dots represent the relevant adaptation options. It is clear from the Figure that analyzing the entire adaptation space is overhead because there is no need for it. Instead, it is more efficient to achieve the adaptation goals of DeltaIoT.v2, i.e., average packet
loss < 10% and latency < 5%, by only analyzing the relevant adaptation options. It would reduce the analysis time significantly, which would eventually enable formal analysis approaches to deal with large adaptation spaces efficiently. Note that the adaptation space has a dynamic behavior, i.e., the qualities of each adaptation option change according to the current runtime uncertainties as shown in Figure 1.2. Therefore, it might be inefficient to use caching or searching techniques.

### 1.4 Limitations

This thesis has three limitations:

1. In the field of self-adaptation, we consider only those self-adaptive systems that follow the principles of architecture-based adaptation, (see Section 2.1).

2. In the field of Machine Learning (ML), we only focus on supervised learning with a classification approach, (see Section 2.3). In addition, we only use one type of ANN, i.e., Multi-Layer Perceptron (MLP), (see Section 2.3.4).

3. We use simulation to evaluate our approach on two instances of an IoT application, (see Section 3), and against one formal analysis approach known as ActivFORMS, (see Section 2.2).

4. We do not apply ML to achieve the optimization goal, i.e., goal with no concrete threshold, of the IoT application.
Figure 1.2: Impact of the runtime uncertainties on the qualities of one adaptation option over six particular adaptation cycles in DeltaIoT.v2

1.5 Contribution

This thesis has the following contributions to the state-of-the-art:

1. A generic online learning approach that enables the formal analysis approaches to efficiently analyze large adaptation spaces without compromising quality guarantees.

2. A complete implementation of the learning approach.

3. Evaluation of the learning approach by applying it on two different instances of an IoT application.

1.6 Target Group

We aim to target both academia and industry readers. In academia, our approach could be interesting for the researchers that focus on speeding up the analysis process in self-adaptive systems. In particular, authors of formal analysis approach, since we enable their approaches to analyze large adaptation spaces efficiently. On the other hand, we evaluate our approach on an IoT application made by VersaSense\textsuperscript{1}. We believe that it could bring the attention of some IoT companies.

\textsuperscript{1}https://www.versasense.com/
1.7 Outline

The remaining of this thesis is structured as follows. In Section 2, we present the background of our research problem. Then, in Section 3, we describe an IoT application that we use to evaluate our approach. Furthermore, in Section 4, we illustrate the scientific methodology that we follow to conduct the experimentation. Later, in Section 5, we present our suggested learning approach. After that, in Section 6, we provide the implementation of our approach on the IoT application that we present in Section 3. Then, in Section 7, we discuss and analyze the results. In Section 8, we describe the related work. Finally, we conclude this thesis and provide future directions in Section 9.
2 Background

In this Section, we provide a brief explanation of the concepts that are related to our research problem. We begin this Section with the theory of architecture-based adaptation. Then, we provide an overview of an existing formal analysis approach called ActivFORMS. In the end, we describe the fundamentals of Machine Learning (ML).

2.1 Architecture-Based Adaptation

Architecture-based adaptation [7, 8, 9, 10, 11, 12] is one important approach to realize self-adaptation. It splits a self-adaptive system into two sub-systems: a managed system and a managing system, (see Figure 2.3). The managed system refers to the software system that holds the domain logic and is subject to adaptation. The managing system holds the adaptation concerns of the managed system. The managed system provides a sensor and actuator, which are used by the managing system to communicate with it. The managing system is equipped with an external feedback loop that provides a generic mechanism to realize self-adaptation. The feedback loop, also known as Monitor, Analyzer, Planner, Executor, Knowledge (MAPE-K) [8, 26, 27, 28, 29, 30, 31], was proposed by International Business Machines (IBM) in 2003 and is highly used in autonomic and self-adaptive systems. The managing system uses the MAPE-K feedback loop to accomplish the adaptation goals of the managed system. It works as follows:

1: The monitor uses the sensor to collect runtime data from the software system and its operating environment, which is otherwise impossible or hard
to achieve before deployment. The collected runtime data contains uncertain properties of the managed system and its environment. The properties are subject to change.

2: The monitor uses the collected runtime data to update the runtime models [32, 33, 34], i.e., managed system model and environment model, which are placed in the knowledge repository. These models have parameters to capture the uncertain properties of the managed system and its environment.

3: The monitor provokes the analyzer.

4.1: The analyzer collects the adaptation options from the knowledge repository. The adaptations options have parameters to capture the uncertain properties of the managed system, managed system’s environment, and adaptation goals. The analyzer sets the parameters of the adaptation options with the collected uncertain properties.

4.2: The analyzer analyzes the adaptation options by using a specific analysis approach that estimates the qualities of the adaptation options. Here, qualities refer to the properties of the adaptation goals. The analyzer then updates the qualities parameters of the adaptation options with the qualities properties obtained from the analysis.

5: The analyzer provokes the planner.

6.1: The planner collects the adaptation options from the knowledge repository and determines the best adaptation option based on the adaptation goals.

6.2: The planner generates the adaptation plan for the best adaptation option and saves it in the knowledge repository. The adaptation plan consists of a set of adaptation actions that are required to adapt the managed system.

7: The planner provokes the executor.

8: The executor collects the adaptation plan from the knowledge repository.

9: The executor adapts the managed system by executing the adaptation plan via the actuator.

Hence, in this way, architecture-based adaptation realizes self-adaptation.

2.2 Active FORmal Models for Self-Adaptation

Active FORmal Models for Self-adaptation (ActivFORMS) [17] is a model based formal analysis approach for architecture-based adaptation. It integrates a new component called statistical model checker [24, 25] with the MAPE-K feedback loop that enables it to provide guarantees to achieve the adaptation goals, (see Figure 2.4). The monitor, planner, and executor components of the MAPE-K work same as mentioned in Section 2.1. Therefore, we start with the analyzer that works as follows:

3: The monitor provokes the analyzer.

4.1: The analyzer collects the adaptation options from the knowledge repository.
4.2: The analyzer sends the adaptation options to the statistical model checker to determine their qualities, i.e., properties of the adaptation goals.

4.3: The statistical model checker uses simulation and statistical techniques to estimate the qualities of the adaptation options. In particular, it simulates the quality models with the adaptation options to estimate their qualities. The accuracy of the estimation is based on the number of simulations, i.e., a high number leads to better accuracy but requires more time and resources. A quality model is a runtime model that represents the managed system and its environment for a specific adaptation goal, which is subject to adaptation. It has parameters to capture the uncertain properties of the managed system and its environment.

4.4: The analyzer updates the qualities parameters of the adaptation options with the qualities properties that are estimated by the statistical model checker.

5: The analyzer provokes the planner.

Hence, in this way, ActivFORMS analyzes the adaptation space to achieve the adaptation goals of self-adaptive systems.

2.3 Machine Learning

Machine Learning (ML) [35, 1] is a study of algorithms that enable the computer/machine to learn from data. It uses statistical techniques to interpret and make predictions about the data. ML has two sub-branches, (see Figure 2.5), and in this thesis, we focus on supervised learning.

2.3.1 Supervised Learning

In supervised learning, the computer learns under human supervision. In particular, it learns to predict the output, i.e., label, of unseen data by constructing
Figure 2.5: An overview of the sub-branches in ML [1]

a model of the input and output relation in the training data given by the human. The training data consist of a set of samples and their corresponding labels. Each sample is represented with one or more attributes, also known as features. Whereas, unseen data, also known as testing data, contain a set of samples without the labels. This mechanism is fundamentally different from the traditional programming approach where rules are explicitly programmed, (see Figure 2.6). In supervised learning, we can train the computer either offline, i.e., before deployment, or online, i.e., after deployment. In this thesis, we do online learning, also known as incremental learning or continuous learning, since we deal with runtime uncertainties, which are unpredictable before the deployment. The online learning enables the computer to learn in an uncertain environment by continuously adapting itself according to the flow of input data. It is considered to be the best practice in autonomous systems [36].

2.3.2 Classification

Within supervised learning, we focus on the classification approach in which the computer learns to map each sample of the input data to one of the predefined classes. The mapping is based on the sample’s features, which can be expressed as:

$$x \rightarrow y$$

where $x = [x_1, x_2, x_3, \ldots, x_n]$ represents $n$ features of a sample, and $y \in \{0, 1, 2, 3, \ldots, K\}$ is the mapped class from a predefined set with $K$ classes. Classification is used in various applications – for instance, spam filtering where the computer identifies whether an email is a spam or not [21]. There are several learning algorithms
Figure 2.6: An overview of the concept of the supervised learning compare to the traditional programming approach [2]

as shown in Figure 2.5, which can be used in the classification approach. In this thesis, we focus on Artificial Neural Networks (ANN).

2.3.3 Artificial Neural Networks

Artificial Neural Network (ANN) [21, 22] were first introduced by the mathematician Walter Pitts and the neurophysiologist Warren McCulloch in 1943. They presented a simple yet powerful computational model, also known as an artificial neuron, that uses propositional logic to mimic the behavior of a biological neuron. It takes one or more binary inputs and produces a binary output. Pitts and McCulloch showed that it is possible to calculate any logical proposition by building a network of artificial neurons [21]. Since then, many new ANN were introduced, and in this thesis, we focus on Multi-Layer Perceptron (MLP).

2.3.4 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) [21, 22] is one of the most commonly used ANN to learn a function to map a given input to an output. It consists of one input layer, one or more hidden layers, and one output layer. ANN with two or more hidden layers is called a deep neural network. MLP is also known as a feed-forward neural network, since the input data travel in one direction, i.e., from the input to output layer, (see Figure 2.7). The input layer consists of simple neurons that capture the features X of a given sample. The number of neurons in the input layer is equivalent to the number of features X. Every layer except the output layer contains a bias neuron. The input layer is connected to the hidden layers, which are further connected to the output layer that predicts the output of a given sample. Note that there is no limit to how many neurons the hidden layers can have. Similarly, the output layer can also have one or multiple neurons.
MLP uses forward propagation to predict the output of a given sample. In forward propagation, the output of the simple neurons is the same as the input. Similarly, the output of the bias neurons is always 1. However, the remaining neurons compute their output as:

\[ y = \lambda \left( \sum_{i=1}^{n} x_i \cdot w_i + b \right) \]

where \( y \) represents the output, \( \lambda \) is an activation function, \( x \) is the input feature from \( n \) features, \( w \) is a weight of corresponding feature, and \( b \) is the bias, (see Figure 2.8). Each input \( X \) is associated with a weight \( W \) and a bias of value 1.

The neuron uses activation function to produce the output. The output of one neuron serves as input to the neurons present in the next layers. Tanh, logistic and rectified linear unit (ReLU) are most commonly used activation functions. The output of these functions lie within a specific range, (see Figure 2.9). The range of the tanh function is from -1 to 1. Whereas the logistic has from 0 to 1,
Figure 2.9: Ranges of the activation functions

and the ReLU has from 0 to ∞, i.e., $max(0, \infty)$. Here, $\infty$ refers to the weighted sum of $n$ input features. Note that the activation function has high impact on the accuracy of MLP, therefore it must be selected carefully.
3 DeltaIoT: Internet of Things Application

In this Section, we briefly introduce two instances, i.e., DeltaIoT.v1 and DeltaIoT.v2, of an IoT application. Both of the instances are deployed at the Computer Science Department of KU Leuven\(^2\) by VersaSense\(^3\) for smart environment monitoring. Due to pragmatic reasons, in this thesis, we use a simulator of these instances, since thorough experimentation on the deployed system can be time-consuming. We start the Section by describing DeltaIoT.v1. Similarly, then, we present the other instance, i.e., DeltaIoT.v2. In the end, we describe how these instances realize self-adaptation.

3.1 DeltaIoT.v1

DeltaIoT.v1 is a research artifact presented in 2017 by Weyns et al \cite{weyns2017deltaiot}. It simulates a smart mesh network that consists of 15 Long-Range (LoRa) IoT motes. For instance, heat sensors for sensing the temperature, passive infrared sensors for monitoring the building, and RFID sensors for providing the access control. The motes are connected via a wireless link, (see Figure 3.10). The motes deliver their sensing data, i.e., packets, to the monitoring facility, known as a gateway, where necessary actions, e.g., changing motes configurations, can be taken if needed. The communication in the network is time-synchronized and formulated in cycles that contain a predefined number of slots. A slot enables communication among two motes, i.e., sender and receiver, that consume their battery power for communication. The duration of an adaptation cycle is 9.5 minutes. The first 8 minutes are allocated to send the sensing data to the gateway. The remaining 1.5 minutes are available to adjust the motes configurations.

\(^2\)https://www.kuleuven.be/english/
\(^3\)https://www.versasense.com/
In this thesis, we consider two types of runtime uncertainties that may affect the quality requirements of DeltaIoT.v1. First, interference in the network due to other WiFi signals or bad weather conditions. Second, dynamic load of packets, e.g., passive infrared sensors only generate packets when human presence is detected. Due to offline experimentation, we include profiles of runtime uncertainties in the simulator to produce the runtime uncertainties in particular links, (see Figure 3.11). The profiles are provided by the IoT application and contain real-time data for 12 hours, i.e., 8:00 to 20:00. The above two profiles represent the number of packets generated by a particular mote. It shows that during the working hours, the mote produces a high number of packets, whereas, after 16:00, the number of packets decreases dramatically. The bottom two profiles represent Signal to Noise Ratio (SNR), i.e., interference, in the network. Lower SNR corresponds to high interference that can lead to high packet loss, whereas higher SNR is vice-versa.

In this thesis, we focus on the following quality requirements of DeltaIoT.v1:

1. The average packet loss in the network should be less than 10%.
2. The average latency in the network should be less than 5%.
3. The average energy consumption of the motes should be minimum.

The motes require optimal configuration to achieve the quality requirements. For instance, each mote has two types of configurations, i.e., power range and distribution of the packets. The power range can be configured between 0 to 15. High power range increases the signal strength but consumes more battery. Whereas, low power range decreases the signal strength that may increase the packet loss. Therefore, the optimal power range is required to improve the lifetime of the motes.
Similarly, a mote with two parents-motes can distribute its packets among them. The distribution range in DeltaIoT.v1 is 20%, which allows a mote to send its packets to the parent-motes in six different ways. For instance, 0% to one parent and 100% to the other parent, 20/80, 40/60, 60/40, 80/20, and 100/0. Hence, the optimal distribution range is needed to reduce packet duplication. Based on this combination, the sum of possible configuration in DeltaIoT.v1 is 216, i.e.,

\[ d^m \]

where \( d \) is the number of packets distribution ways, which is 6, and \( m \) is the number of motes with two parents-motes, which is 3 (see mote 12, mote 7 and mote 10 in Figure 3.10).

### 3.2 DeltaIoT.v2

DeltaIoT.v2 is another artifact that also simulates a smart mesh network that consists of 37 Long-Range (LoRa) IoT motes. For instance, heat sensors for sensing the temperature, passive infrared sensors for monitoring the building, and RFID sensors for providing the access control. The motes are connected via a wireless link. Figures 3.12 and 3.13 show the physical deployment and structure of DeltaIoT.v2. DeltaIoT.v2 uses the same technology as DeltaIoT.v1. However, due to a high number of motes, the duration of an adaptation cycle is 12 minutes. The first 9.5 minutes are allocated to send the sensing data to the gateway. The remaining 2.5 minutes are available to adjust the motes configurations. The runtime uncertainties and quality requirements of DeltaIoT.v2 are same as DeltaIoT.v1.
In addition, the power ranges of the motes are also same as DeltaIoT.v1. However, the distribution range is 34%, which allows a mote to send its packets to the parent-motes in four different ways. For instance, 0% to one parent and 100% to another parent, 34/66, 66/34, and 100/0. Hence, the total possible configurations in DeltaIoT.v2 is $4^6 = 4096$, which is much higher than DeltaIoT.v1. Here, 4 is the number of packets distribution ways and 6 is the number of motes with two parents-motes (see mote 16, mote 35, mote 14, mote 22, mote 3 and mote 6 in Figure 3.13).

### 3.3 Self-Adaptive DeltaIoT

DeltaIoT.v1 and DeltaIoT.v2 realize architecture-based adaptation, (see Section 2.1), that enable them to autonomously configure the settings of the motes whenever the runtime uncertainties violate their adaptation goals. In architecture-based adaptation, the quality requirements become adaptation goals. Both of the instances use ActivFORMS, (see Section 2.2), to achieve the adaptation goals at runtime. In this thesis, we refer to this approach as no learning approach. Figure 3.14 shows the basic architecture of DeltaIoT with no learning approach. This architecture is similar for DeltaIoT.v1 and DeltaIoT.v2. The managed system contains a gateway and a set of motes that are connected to forward their sensing data to the gateway. The managed system provides a sensor for monitoring and an actuator to adjust the motes configurations. The managing system follows the principles of ActivFORMS, (see Section 2.2). Here, we briefly discuss how it
works in DeltaIoT.v1 and DeltaIoT.v2.

1: The monitor collects runtime data from the managed system and its operating environment via the sensor. The collected runtime data contains the uncertain properties of the managed system, i.e., power and distribution range of the motes, and its operating environment, i.e., interference in the network and load of packets.

2: The monitor updates the corresponding runtime models with the collected data.

3: The monitor provokes the analyzer.

4.1: The analyzer collects the adaptation options. The number of adaptation options is based on the managed system, i.e., 216 when the managed system is DeltaIoT.v1, and 4096 when it is DeltaIoT.v2.

4.2: The analyzer sends the adaptation options to the statistical model checker to determine their qualities.

4.3: The statistical model checker uses simulation and statistical techniques to estimate the qualities of the adaptation options.
4.4: The analyzer updates the qualities parameters of the adaptation options with the qualities properties estimated by the statistical model checker.

5: The analyzer provokes the planner.

6.1: The planner collects the adaptation options from the knowledge repository and determines the best adaptation option based on the adaptation goals. In particular, the planner checks the qualities of each adaptation option and selects the one that has packet loss < 10\%, latency < 5\%, and minimum energy consumption.

6.2: The planner generates the adaptation plan for the selected adaptation option. The adaptation plan contains a set of adaptation actions. For instance, set the power range of mote 4 from 10 to 7 and change the distribution range between mote 3 and 4 to 20/80. The distribute range is dependent on the managed system, i.e., 20\% in DeltaIoT.v1 and 34\% in DeltaIoT.v2.

7: The planner provokes the executor.

8: The executor collects the adaptation plan from the knowledge repository and adapts the managed system by executing the adaptation plan via the actuator.

We use this approach, i.e., no learning, for the evaluation of our learning approach.
4 Method

In this Section, we describe the methodology that we use to evaluate our approach. We begin this Section by explaining the experiments that we conducted to determine the effectiveness of our approach. In the end, we discuss the reliability and validity of our work.

4.1 Controlled Experiment

We use a controlled experiment methodology to answer the research question of this thesis. This methodology is often used to measure quantitative data, and therefore, we use it to demonstrate the effectiveness of our approach. In a controlled experiment, there are two types of variables named as independent and dependent. The independent variables can be seen as input, whereas, the dependent variables as output, which is directly affected by the input. To evaluate our approach, we conduct two controlled experiments on both instances of the IoT application that we explained in Section 3. Table 4.1 shows the independent and dependent variables of our controlled experiments. In the first experiment, the independent variable is a no learning approach that we explained in Section 3.3. In the second experiment, the independent variable is a learning approach that we present in Section 5. The dependent variables in both of the experiments are adaptation goals, adaptation space, analysis time and adaptation time. We apply learning and no learning approaches on DeltaIoT.v1 and DeltaIoT.v2 to measure their impact on the dependent variables.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Independent Variables</th>
<th>Dependent Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No learning approach</td>
<td>(i) Adaptation goals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(iii) Analysis time</td>
</tr>
<tr>
<td>2</td>
<td>Learning approach</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Independent and dependent variables in our controlled experiments

4.2 Reliability and Validity

Due to pragmatic reasons, we use simulation to evaluate our approach. The simulation produces runtime uncertainties in the IoT application by following specific types of randomness. Therefore, randomness may impose a reliability threat that might slightly change the evaluation results if the experiments are repeated. However, we minimize this threat by running the simulator for 47.5 hours during each experiment. Moreover, we provide a complete implementation to replicate our work. On the other hand, we evaluate our approach on a specific application of the IoT domain. It introduces an external validity threat because our conclusion cannot be generalized in other domains. Hence, to determine the generalization of our approach, we require more evaluations from different domains.
5 Our Learning Approach

In this Section, we present our approach that uses online learning to reduce the adaptation space in self-adaptive systems. We begin this Section by explaining the architecture of our approach. Then, we illustrate the offline activities in our approach. Similarly, in the end, we describe the online activities in our approach.

5.1 Architecture

Recall from Section 2.1 that architecture-based adaptation splits a self-adaptive system into a managing system and a managed system. We keep this separation of concern and embed a new component, called machine learner, into the managing system so that it can integrate with the MAPE-K feedback loop. As we mentioned earlier that the formal analysis approaches use model checking techniques to analyze the adaptation space. Therefore, to integrate with them, we also keep the model checker component in the managing system, (see Figure 5.15). The

![Figure 5.15: Generic architecture of our approach](image)

monitor, planner, and executor components of the MAPE-K feedback loop work same as discussed in Section 2.1. Therefore, we start with the analyzer that works as follows:

3: The monitor provokes the analyzer.

4.1: The analyzer collects the adaptation options from the knowledge repository.

4.2: The analyzer sends the adaptation options to the machine learner to determine the relevant adaptation options.

4.3: The machine learner collects the learning models from the knowledge repository and apply them on the adaptation options to find the relevant adap-
tation options. A learning model is a runtime model that captures the uncertain properties of the adaptation options and predicts their qualities for a specific adaptation goal.

4.4: The analyzer sends the relevant adaptation options to the model checker.

4.5: The model checker simulates the quality models with the relevant adaptation option to estimate their qualities.

4.6: The analyzer updates the qualities parameters of the relevant adaptation options with the qualities properties that are estimated by the model checker.

4.7: The analyzer sends the analyzed relevant adaptation options to the machine learner.

4.8: The machine learner trains the learning models with the analyzed relevant adaptation options. It is also known as online learning, which keeps the learning models up to date.

5: The analyzer provokes the planner.

The machine learner component is generic. Therefore any online learning approach e.g., classification and regression, can be integrated. In addition, any suitable model checking technique, e.g., statistical or probabilistic, can be used. Our approach is modular since we use one learning model for one adaptation goal. The learning models work in a chain to determine the relevant adaptation options, (see Figure 5.16). The input to the first learning model is all the adaptation options. It predicts the qualities of the adaptation options for one specific adaptation goal. The input to the second learning model is only those adaptation options that are valid to achieve the first adaptation goal. Similarly, the input to the next learning models is the adaptation options that are able to achieve the previous adaptation goals. In the end, we get the relevant adaptation options. For more details, see Listing 3.

5.2 Offline and Online Activities

Our approach requires three manual activities before the deployment. We refer to these activities as offline activities. Once our approach is deployed, the remaining activities work autonomously. We refer to these activities as online activities. Figure 5.17 shows an overview of these activities.
5.2.1 Offline Activities

In the first activity, i.e., data collection, we collect runtime data from the system over multiple adaptation cycles. Here, the system refers to a self-adaptive system on which our approach is going to be applied. The runtime data contains the uncertain properties and qualities of the adaptation options. We achieve these properties by running the system with a formal analysis approach, and save the adaptation options once the model checker estimated their qualities.

The second activity is called model selection in which we use the collected data to train the learning models to determine their best hyper-parameters. The learning models often have several hyper-parameters that we can fine-tune to get better accuracy. This activity ends with a selection of the best hyper-parameters which are then used in the next activities.

The third activity is called training cycles selection in which we train the learning models with the selected hyper-parameters to determine how many initial training cycles are required before the learning models can start selecting the relevant adaptation options.

The last activity is called system set up in which we place new instances of the learning models in the machine learner component and connects it with the analyzer component. The learning models are initialized with the selected hyper-parameters once the system is deployed.
5.2.2 Online activities

The online activities are part of the MAPE-K feedback loop. Therefore, no human interference is required. First, the monitor collects runtime data and updates the corresponding runtime models. Then, the analyzer collects the adaptation options from the knowledge repository. If the current adaptation cycle is a training cycle, the analyzer uses the model checker to estimate the qualities of the adaptation options. Then, the learning models are trained on the adaptation options. If the learning models are not initialized yet, they are first initialized and trained, and then placed in the knowledge repository, as runtime models, for future use. On the other hand, if the current adaptation cycle is not a training cycle, then the analyzer uses the learning models to find the relevant adaptation options, which are then further analyzed by the model checker. The learning models are then trained on the analyzed relevant adaptation options. Here, we do online learning, which keeps the learning models up to date. Later, the planner and executor perform their activities to adapt the system, and then this whole process repeats. Note that this is a brief explanation of the activities in the MAPE-K feedback loop, since the details are already discussed in Section 5.1 and 2.1.
6 Implementation

In this Section, we present the implementation of our learning approach. We implement our approach with the help of an open source machine learning library named as Scikit-Learn\textsuperscript{4} [37]. This library provides various learning models, including MLP that we aim to use. We start this Section by presenting the concrete instantiation of our approach on DeltaIoT.v1. Similarly, in the end, we show the concrete instantiation of our approach on DeltaIoT.v2.

6.1 DeltaIoT.v1

We began the implementation with the offline activities. The first activity is about collecting runtime data. We ran DeltaIoT.v1 with ActivFORMS, i.e., no learning approach, (see Section 3.3), for 300 adaptation cycles to collect the runtime data. Listing 1 shows an overview of the runtime data in one particular adaptation cycle.

```json
{
  "motes_snr": [
    3,
    0,
    ...
  ],
...
  "motes_power": [
    0,
    7,
    ...
  ],
...
  "motes_packets_distribution": [
    100,
    100,
    ...
  ],
...
  "motes_load": [
    50,
    100,
    ...
  ],
}
```

\textsuperscript{4}https://scikit-learn.org/stable/
The runtime data contains the uncertain values of the adaptation options. For instance, \texttt{motes\_snr} (SNR) and \texttt{motes\_load} (Load) represent the runtime uncertainties in the operating environment. Whereas, \texttt{motes\_power} (Power) and \texttt{motes\_packets\_distribution} (Distribution) represent the current configurations of the motes. The \texttt{packet\_loss}, \texttt{latency}, and \texttt{energy\_consumption} represent the qualities of the adaptation options.

The second activity is about fine-tuning the hyper-parameters of the learning models. We use one MLP model for packet loss and one for latency adaptation goals. Note that we do not use learning model for energy consumption adaptation goal because it is an optimization goal, which is beyond the scope of this thesis. We start this activity by converting the runtime data into classification datasets. Table 6.2 and 6.3 show an overview of the datasets made for packet loss and latency learning models. Each sample in the datasets is an adaptation option, which is represented with a number of features. The features are its uncertain values that we mentioned above. The datasets also contain labels, which show the qualities of an adaptation option for a particular adaptation goal. There are two types of labels: 1 represents that the corresponding adaptation options are able to achieve the specific adaptation goal, whereas 0 is vice-versa.

We use a \textit{grid search} \cite{21, 22, 38} approach to fine-tune the hyper-parameters of the learning models. This approach checks the accuracy of the learning models by applying all the possible combinations of the hyper-parameters available.

```json
...
"packet\_loss": [
  7.1815500000000005,
  ...
],
"latency": [
  2.81818,
  ...
],
"energy\_consumption": [
  12.7938,
  ...
]
}
```

Listing 1: An overview of the runtime data in one particular adaptation cycle in DeltaIoT.v1

<table>
<thead>
<tr>
<th>Sample</th>
<th>SNR</th>
<th>Power</th>
<th>Distribution</th>
<th>Load</th>
<th>Packet Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3, 0, ...</td>
<td>0, 7, ...</td>
<td>100, 100, ...</td>
<td>50, 100, ...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-4, -5, ...</td>
<td>0, 15, ...</td>
<td>100, 20, ...</td>
<td>100, 100, ...</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1, -3, ...</td>
<td>0, 9, ...</td>
<td>100, 80, ...</td>
<td>27, 0, ...</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.2: An overview of the dataset made for the packet loss learning model
Table 6.3: An overview of the dataset made for the latency learning model

<table>
<thead>
<tr>
<th>Sample</th>
<th>Features</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SNR</td>
<td>Power</td>
</tr>
<tr>
<td>1</td>
<td>3, 0, ...</td>
<td>0, 7, ...</td>
</tr>
<tr>
<td>2</td>
<td>-4, -5, ...</td>
<td>0, 15, ...</td>
</tr>
<tr>
<td>3</td>
<td>1, -3, ...</td>
<td>0, 9, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6.3: An overview of the dataset made for the latency learning model

in a given search space. In the Scikit-Learn library, MLP has several hyper-parameters, but we focus on two of the most important ones. The first is the activation function, and the second one is the number of hidden layers. Recall from Figure 2.9 in Section 2.3.4 that activation functions work within a specific range. For instance, the tanh function has a range from -1 to 1, logistic from 0 to 1, and ReLU from 0 to $\infty$. To maximize their effectiveness, we re-scale the features of the adaptation options within their ranges by applying scaling models provided by the Scikit-Learn library. For instance, with standard and max abs scalers models, the features lie from -1 to 1 and with min max scaler model 0 to 1. Note that the re-scale range of standard and max abs scalers is same. The only difference is with standard scaler the mean and variance is 0. We use all three scalers in the grid search to determine the best one.

The search space that we define has 4 activation functions. Here, the number is 4 because the tanh function is applied with both the standard and max abs scalers. In addition, the search space also contains 300 neurons in the first hidden layer. Therefore, the total number of the possible combination is $4 \times 300 = 1200$. There is no such hard rule on defining the search space. Instead, it is based on experimentation. The datasets shown above are used for this search space. Each dataset contains 1000 samples from which 100 samples are used to train the learning model, and 900 samples are used for validation. We intentionally train the learning models on such a small number of samples, since we aim to find those hyper-parameters that can enable the learning models to learn from as few samples as possible. Moreover, it would also speed up the initial training process in our approach. Table 6.4 and 6.5 show the grid search results. The results show

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Scaler</th>
<th>Hidden Layers</th>
<th>Neurons</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>Standard</td>
<td>1</td>
<td>47</td>
<td>0.944</td>
</tr>
<tr>
<td>ReLU</td>
<td>Min max</td>
<td>1</td>
<td>264</td>
<td>0.878</td>
</tr>
<tr>
<td>Tanh</td>
<td>Max abs</td>
<td>1</td>
<td>229</td>
<td>0.872</td>
</tr>
<tr>
<td>Logistic</td>
<td>Min max</td>
<td>1</td>
<td>46</td>
<td>0.715</td>
</tr>
</tbody>
</table>

Table 6.4: Grid search results for the selection of the activation function and neurons in the first hidden layer for the packet loss learning model in DeltaIoT.v1

that the tanh activation function with standard scaler is the best for packet loss learning model. Similarly, it is also the best for the latency learning model but with min max scaler. The results also show the corresponding best number of hidden neurons in the first hidden layer. We separately provide the results of the remaining 1196 combinations, (see GitHub link in Appendix A).

Since the best activation function and number of neurons in the first hidden layer is determined, now define a new search space to determine the best number
### Activation Function | Scaler | Hidden Layers | Neurons | Accuracy (%)
---|---|---|---|---
Tanh | Standard | 1 | 293 | 0.890
ReLU | Min max | 1 | 245 | 0.913
Tanh | Max abs | 1 | 245 | 0.916
Logistic | Min max | 1 | 112 | 0.898

Table 6.5: Grid search results for the selection of the activation function and neurons in the first hidden layer for the latency learning model in DeltaIoT.v1

of hidden layers. The search space contains 9 hidden layers with 300 neurons in each. The total number of the possible combination is $9 \times 300 = 2700$. The approach is the same as we mentioned above, i.e., find the number of neurons in one hidden layer, fix them and repeat this process on the next hidden layer. Note that now the learning models use the selected best activation function and number of neurons found in the previous hidden layer. Table 6.6 and 6.7 show the grid search results. The results show that 9 hidden layers are the best for the

<table>
<thead>
<tr>
<th>Hidden Layers</th>
<th>Neurons</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>47, 161</td>
<td>0.943</td>
</tr>
<tr>
<td>3</td>
<td>47, 161, 17</td>
<td>0.944</td>
</tr>
<tr>
<td>4</td>
<td>47, 161, 17, 105</td>
<td>0.953</td>
</tr>
<tr>
<td>5</td>
<td>47, 161, 17, 105, 194</td>
<td>0.955</td>
</tr>
<tr>
<td>6</td>
<td>47, 161, 17, 105, 194, 276</td>
<td>0.952</td>
</tr>
<tr>
<td>7</td>
<td>47, 161, 17, 105, 194, 276, 285</td>
<td>0.952</td>
</tr>
<tr>
<td>8</td>
<td>47, 161, 17, 105, 194, 276, 285, 12</td>
<td>0.950</td>
</tr>
<tr>
<td>9</td>
<td>47, 161, 17, 105, 194, 276, 285, 12, 225</td>
<td>0.957</td>
</tr>
<tr>
<td>10</td>
<td>47, 161, 17, 105, 194, 276, 285, 12, 225, 293</td>
<td>0.955</td>
</tr>
</tbody>
</table>

Table 6.6: Grid search results for the selection of the hidden layers for the packet loss learning model in DeltaIoT.v1

<table>
<thead>
<tr>
<th>Hidden Layers</th>
<th>Neurons</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>245, 2</td>
<td>0.897</td>
</tr>
<tr>
<td>3</td>
<td>245, 2, 6</td>
<td>0.897</td>
</tr>
<tr>
<td>4</td>
<td>245, 2, 6, 1</td>
<td>0.897</td>
</tr>
<tr>
<td>5</td>
<td>245, 2, 6, 1, 215</td>
<td>0.898</td>
</tr>
<tr>
<td>6</td>
<td>245, 2, 6, 1, 215, 1</td>
<td>0.897</td>
</tr>
<tr>
<td>7</td>
<td>245, 2, 6, 1, 215, 1, 1</td>
<td>0.897</td>
</tr>
<tr>
<td>8</td>
<td>245, 2, 6, 1, 215, 1, 1, 3</td>
<td>0.897</td>
</tr>
<tr>
<td>9</td>
<td>245, 2, 6, 1, 215, 1, 1, 3, 1</td>
<td>0.897</td>
</tr>
<tr>
<td>10</td>
<td>245, 2, 6, 1, 215, 1, 1, 3, 1, 2</td>
<td>0.897</td>
</tr>
</tbody>
</table>

Table 6.7: Grid search results for the selection of the hidden layers in the latency learning model for DeltaIoT.v1

packet loss learning model. Whereas, for the latency learning model, it is 1 that we found earlier with the accuracy of 0.916%. Again, the results of the remaining combinations are provided separately, (see GitHub link in Appendix A). We end
this activity with a selection of best found hyper-parameters and scalers. Table 6.8 summarizes it.

<table>
<thead>
<tr>
<th>Learning Model</th>
<th>Scaler</th>
<th>Activation Function</th>
<th>Hidden Layers</th>
<th>Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Loss</td>
<td>Standard</td>
<td>Tanh</td>
<td>9</td>
<td>47, 161, 17, 105, 194, 276, 285, 12, 225</td>
</tr>
<tr>
<td>Latency</td>
<td>Max abs</td>
<td>Tanh</td>
<td>1</td>
<td>245</td>
</tr>
</tbody>
</table>

Table 6.8: Summary of the selected hyper-parameters and scalers in DeltaIoT.v1

The third activity focuses on determining the required number of initial training cycles at runtime. We use 1000 samples to train and validate the learning models. Note that DeltaIoT.v1 has 216 adaptation options in one adaptation cycle, therefore, 1000 samples $\approx 5$ adaptation cycles. The samples are divided into 4 folds, i.e., 100 for training and 900 for validation, 300/700, 500/500, and 700/300, (see Figure 6.18). The accuracy of both learning models is quite stable, i.e., $\sim 0.93\%$, regardless of the number of training samples. Therefore, we select 1 training cycle, i.e., 216 samples, to initially train the learning models at runtime.

In the fourth and last offline activity, we set up the system. We place the new instances of the learning models in the machine learner component and connect it with the analyzer component of DeltaIoT.v1. Note that the learning models are be initialized by the machine learner at runtime. Figure 6.19 shows the concrete instantiation of our learning approach on DeltaIoT.v1. The machine learner component contains the implementation of MLP and selected scalers. Note that we
do not implement these classes. Instead, we import them from the Scikit-Learn library. The machine learner component is also connected with the knowledge repository to retrieve the corresponding runtime models. The remaining flow is the same as we discussed in Section 5. Listing 2 and 3 show how the learning models are trained and used to select the relevant adaptation options at runtime.

```python
def training(dataset):
    models = repository.get_models()
    features = dataset['features']
    for model, scaler, labels in [
        (models['packet_loss'], models['standard_scaler'], dataset['packet_loss']),
        (models['latency'], models['max_abs_scaler'], dataset['latency']),
    ]:
        scaled_features = scaler.scale(features)
        model.train(scaled_features, labels)
    repository.save_models(models)
```

Listing 2: Training the learning models at runtime

2: Get the learning models from the knowledge repository.

3: Get features from the given dataset. Note that the dataset contains the features, i.e., uncertain values of the adaptation options, and their corresponding packet loss and latency labels.

4-9: Iterate over the learning models together with their corresponding scaler models and labels. Use the scaler model to scale the features. Then, train the learning model with the features and labels.

10: Save the learning models in the knowledge repository.
def select_relevant_adaptation_options(dataset):
    models = repository.get_models()
    features = dataset['features']
    indexes = [i for i in range(0, len(features))]
    for model, scaler in [
        (models['packet_loss'], models['standard_scaler']),
        (models['latency'], models['max_abs_scaler']),
    ]:
        relevant_features = []
        relevant_indexes = []
        scaled_features = scaler.scale(features)
        predictions = model.predict(scaled_features)
        for index, prediction in enumerate(predictions):
            if prediction == 1:
                relevant_features.append(features[index])
                relevant_indexes.append(indexes[index])
        if len(relevant_features) > 0:
            features = relevant_features
            indexes = relevant_indexes
        repository.save_models(models)
    return indexes

Listing 3: Selection of the relevant adaptation options by applying the learning models at runtime

2: Get the learning models from the knowledge repository.

3-4: Get features from the given dataset. Note that the dataset only contains the features. Save the indexes of the features, since we return them at the end.

5-8: Iterate over the learning models together with their corresponding scaler models.

9-10: Initialize two arrays to store relevant features and their indexes for a specific adaptation goal.

11-12: Use the scaler model to scale the features. Then, use the learning model to predict the qualities of the adaptation options for a specific adaptation goal.

13-16: Iterate over predictions. Check if the prediction is valid. If yes, then save the corresponding features and index.

17-19: Check if the relevant features array is not empty. Note that it is possible that none of the prediction is valid. In that case, we forward all the features to the next learning models. Otherwise, we replace the features with the relevant features. It enables the next learning model to predict the qualities for only the relevant features. Hence, in this way we find the relevant adaptation options.

20: Save the learning models in the knowledge repository.

21: Return the indexes of the relevant features.
6.2 DeltaIoT.v2

In DeltaIoT.v2, we apply the same approach as DeltaIoT.v1. Table 6.9 shows a summary of the selected hyper-parameters and scalers. In the third activity, i.e., training cycles selection, we use 16000 samples to train and validate the learning models. The number is higher than DeltaIoT.v1 because DeltaIoT.v2 has 4096 adaptation options in one adaptation cycle. Therefore, 16000 samples \( \approx 4 \) adaptation cycles. The samples are divided into 4 folds, i.e., 1600 for training and 14400 for validation, 4800/11200, 8000/8000, and 11200/4800, (see Figure 6.20). The accuracy of both learning models is again quite stable, i.e., \( \sim 0.94\% \), regardless of the number of training samples. Therefore, we select 1 training cycle, i.e., 4096 samples, to initially train the learning models at runtime.

In the last activity, we set up the system in the same way as we did in DeltaIoT.v1, (see Figure 6.21). Note that there is only one scaler model, i.e., standard scaler, since both learning models selected the same scaler, (see Table 6.9). Therefore, the scaler model is shared between the learning models. We use the same approach,

<table>
<thead>
<tr>
<th>Learning Model</th>
<th>Scaler</th>
<th>Activation Function</th>
<th>Hyper-Parameters</th>
<th>Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Loss</td>
<td>Standard</td>
<td>Tanh</td>
<td>4</td>
<td>274, 177, 129, 94</td>
</tr>
</tbody>
</table>

Table 6.9: Summary of the selected hyper-parameters and scalers in DeltaIoT.v2 training cycles selection, we use 16000 samples to train and validate the learning models. The number is higher than DeltaIoT.v1 because DeltaIoT.v2 has 4096 adaptation options in one adaptation cycle. Therefore, 16000 samples \( \approx 4 \) adaptation cycles. The samples are divided into 4 folds, i.e., 1600 for training and 14400 for validation, 4800/11200, 8000/8000, and 11200/4800, (see Figure 6.20). The accuracy of both learning models is again quite stable, i.e., \( \sim 0.94\% \), regardless of the number of training samples. Therefore, we select 1 training cycle, i.e., 4096 samples, to initially train the learning models at runtime.

In the last activity, we set up the system in the same way as we did in DeltaIoT.v1, (see Figure 6.21). Note that there is only one scaler model, i.e., standard scaler, since both learning models selected the same scaler, (see Table 6.9). Therefore, the scaler model is shared between the learning models. We use the same approach,
as shown in Listing 2 and 3 to train and select the relevant adaptation options at runtime. The remaining workflow is the same as discussed in Section 5.

Figure 6.21: An overview of the concrete instantiation of our learning approach on DeltaIoT.v2
7 Results and Analysis

In this Section, we present the results and analysis of the controlled experiments, (see Section 4), that we conduct on the simulator of DeltaIoT. We ran the simulation on Intel Xeon Skylake 8 core 2GHz machine with 16GB RAM. We begin this Section by analyzing the adaptation results of DeltaIoT.v1. Similarly, we then analyze the adaptation results of DeltaIoT.v2. In the end, we conclude the results.

7.1 DeltaIoT.v1

We simulate DeltaIoT.v1 with learning and no learning approaches for 300 adaptation cycles, where the maximum duration of each adaptation cycle is 570 seconds. During the learning approach, we use the first adaptation cycle for training the learning models, since we empirically derive this number from the offline activities. Figure 7.22 shows the adaptation results of DeltaIoT.v1 with the learning and no learning approaches. It is clear from the adaptation results that both approaches achieve the adaptation goals, i.e., average packet loss <10%, average latency <5%, and minimum energy consumption. Note that minimum energy consumption is an optimization goal, which means that unlike other adaptation goals it has no concrete threshold. With the learning approach, the average packet loss is 7.6% compared to the no learning approach that has 8.4%. Similarly, the average latency with the learning approach is 0.4%, whereas, with the no learning approach, it is 0.7%. The average energy consumption with both approaches is 12.7 coulomb. The adaptation results also show that the no learning approach analyzes the entire adaptation space, i.e., 216 adaptation options, to achieve the adaptation goals. In contrast, the learning approach reduces the adaptation space by 76% on average, and thus achieves the adaptation goals by only analyzing 51
adaptation options. The reduced adaptation space enables the learning approach to complete the analysis within 5 seconds on average, compared to the no learning approach that takes 26 seconds. Furthermore, fast analysis consequently reduces the adaptation time in the learning approach, i.e., 5 seconds on average, whereas, with the no learning approach, it is 26 seconds. It is worth to mention that in the learning approach, the learning models have almost no overhead on the analysis and adaptation time since they take a tiny amount of time, i.e., <1 second on average, for training and predictions, (see Figure 1.26 in Appendix A).

Figure 7.23 shows an overview of the selected adaptation options in one particular adaptation cycle by the learning approach. Each orange dot represents an adaptation option in one particular adaptation cycle. Whereas, green dots represent those adaptation options that are selected by the learning approach. The Figure shows that learning approach has selected most of the relevant adaptation options. No doubt, few outliers are also selected. The outliers would be ignored by the planner, a component of MAPE-K feedback loop, during the selection of the best adaptation option. The results of the adaptation options selected by the learning approach in the remaining adaptation cycles are provided separately, (see GitHub link in Appendix A).

7.2 DeltaIoT.v2

We also simulate DeltaIoT.v2 with learning and no learning approaches for 300 adaptation cycles, where the maximum duration of each adaptation cycle is 570
During the learning approach, we again use the first adaptation cycle for training the learning models, since we empirically derive this number from the offline activities. Figure 7.24 shows the adaptation results of DeltaIoT.v2 with learning and no learning approaches. It is clear from the adaptation results that both approaches achieve the adaptation goals. With the learning approach, the average packet loss is 8.8% compared to the no learning approach that has 9.0%. Similarly, the average latency with the learning approach is 0.6%, whereas, with the no learning approach, it is 1.0%. The average energy consumption with both approaches is 66.5 coulomb. The adaptation results also show that the no learning approach uses all the available time to analyze as much adaptation space as possible. On average, it analyzes 2825 adaptation options out of 4096. In contrast, the learning approach reduces the adaptation space by 92% on average, and thus achieves the adaptation goals by only analyzing 293 adaptation options. The reduced adaptation space enables the learning approach to complete the analysis within 48 seconds on average, compared to the the no learning approach that takes all the available time, i.e., 570 seconds. Furthermore, fast analysis consequently reduce the adaptation time in the learning approach, i.e., 48 seconds on average, whereas, with the no learning approach, it is 570 seconds. It is again worth to mention that in the learning approach, the learning models have almost no overhead on analysis and adaptation time, since they take a tiny amount of time, i.e., <1 second on average, for training and predictions, (see Figure 1.26 in Appendix A). Figure 7.25 shows an overview of the selected adaptation options in one particular adaptation cycle by the learning approach. The Figure shows that the subset selected by the learning approach contains most of the relevant adaptation options and few irrelevant adaptation options. However, the irrelevant adaptation options would not impact on the adaptation goals, since the subset also has several relevant adaptation options that are able to achieve the
adaptation goals. The results of the adaptation options selected by the learning approach in the remaining adaptation cycles are provided separately, (see GitHub link in Appendix A).

7.3 Conclusion of Results

The research question that we aimed to answer in thesis was:

*Can we use ANN to reduce the adaptation space in self-adaptive systems without compromising the adaptation goals?*

The results demonstrate that we can use ANN to reduce the adaptation space in self-adaptive systems without compromising the adaptation goals. On average, the learning approach successfully reduced the adaptation space by 76% in DeltaIoT.v1 and 92% in DeltaIoT.v2. The reduced adaptation space further reduces the analysis time on average by 80% in DeltaIoT.v1 and 91% in DeltaIoT.v2. Moreover, with the learning approach, the achievement of adaptation goals is slightly better compared to the no learning approach. We also found that having one learning model for each adaptation goal has almost no impact on the analysis and adaptation time, since the average time is less than 1 second.
8 Related Work

In this Section, we discuss the related work. We mainly focus on those existing approaches that apply ML, especially supervised learning, in self-adaptive systems.

Elkhodary et al. [39] present a framework called FUSION that uses online learning to learn the impact of applied adaptation options on the quality requirements. Tree-based learning models are used to analyze the system’s runtime data to find its relationships with the features, i.e., system capabilities. The relationships are used to select the best adaptation option, which significantly improves the analysis time. However, FUSION focus on feature selection space, whereas, we target the adaptation space, i.e., configuration space.

Epifani et al. [40] combine Bayesian learning with probabilistic quality models that have parameters to capture the runtime uncertainties. During design time, domain knowledge is used to set the initial values of the parameters. Later, the system’s runtime data is provided to a Bayesian model that predicts the value of a quality requirement and updates the parameters of a corresponding quality model with it. A probabilistic model checker is used to verify the quality models. This novel approach is used by Calinescu et al. [19] to provide quantitative verification at runtime. In contrast, we use learning to reduce the adaptation space to relevant adaptation options. However, [19] can be used in our approach to update the quality models.

Ghahremani et al. [41] treats a self-adaptive system as a black-box and uses Ensemble learning models to predict changes in its utility. In particular, it predicts the values of the quality requirements and then compares them with previously predicted values to calculate the differences. Later, the differences are used to sort the adaptation options. Whereas, we use learning to find the relevant adaptation options and uses the model checker to analyze them further. The adaptation options are then sorted based on their qualities that are estimated by the model checker.

Durate et al. [42] apply learning to obtain human-readable models that explicitly capture the runtime uncertainties. More concretely, clustering techniques are applied to the runtime data to find the uncertainties that are not learned yet. Then, the probability of each uncertainty is calculated, and the human-readable model is generated for each adaptation cycle. The model contains future predictions with probabilistic estimation. In comparison, we apply ML to improve the efficiency of formal analysis approaches that do not require any human in the feedback loop.

Rodrigues et al. [43] propose a method that assures quality requirements by combining design time and runtime activities. During design time, the system’s behavior is verified using a model checker, and a prototype is implemented. The prototype is then simulated to learn the system. Later, learning knowledge is transferred to the real system. During runtime, data mining and classification techniques are applied to analyze the runtime data. Then, the analyzed results are compared with the properties that are verified during the design time. In contrast, our approach uses the model checker to learn the relevant adaptation options as well as analyze them at runtime.

Our previous work [44] uses online supervised learning to reduce the adaptation space in self-adaptive systems. However, it does not use ANN and is limited
to achieving one adaptation goal only.
9 Conclusion

In this thesis, we examined whether ANN can be used to reduce the adaptation space in self-adaptive systems without compromising the adaptation goals. Based on our results, we can confirm that it is indeed possible. Our contribution to the state-of-the-art is a learning approach that integrates with the MAPE-K feedback loop and continuously learns the adaptation space on-the-fly. The continuous learning enables the learning models to efficiently reduce the adaptation space to a subset of relevant adaptation options. It enables formal analysis approaches to efficiently analyze large adaptation spaces without compromising quality guarantees. We demonstrated the effectiveness of our approach on two different instances of an IoT application.

9.1 Future Work

In the future, we aim to work in four directions. First, to determine the generalization of our approach, we plan to apply it on different self-adaptive artifacts, e.g., Tele Assistance System [45] and Automated Traffic Routing Problem [46], developed by the community. Second, we plan to explore other types of ANN, such as Convolutional Neural Networks and Recurrent Neural Networks. Third, we plan to evaluate our approach against other existing formal analysis approaches such as RQV. Fourth, we plan to find a way to achieve optimization goals with the help of ML.
References


A Appendix

The implementation of our learning approach and in-depth results of the offline activities as well as the selection of the relevant adaptation options are available on GitHub – https://github.com/sarpreetsingh3131/4dv50e

Figure 1.26: Time measurements of the learning models used in DeltaIoT.v1 (left) and DeltaIoT.v2 (right)

Figure 1.27: Time measurements of the offline activities in DeltaIoT.v1 (left) and DeltaIoT.v2 (right)
Figure 1.28: An example of the feature scaling in one particular adaptation cycle in DeltaIoT.v1