**Managing uncertainty in self-adaptive systems through the lens of deep reinforcement learning**

Alireza Kavianifar

Computer Engineering Department, Tarbiat Modares University, Iran

Saeed Jalili[[1]](#footnote-1)\*

Computer Engineering Department, Tarbiat Modares University, Iran

**Abstract**

**Context:** Engineering software systems in an uncertain and ever-changing operating environment is a challenging task. Uncertainty is a cross-cutting phenomenon and system objectives may be jeopardized if it is not handled properly. Uncertainty in self-adaptive systems is any deviation of deterministic knowledge that may reduce the confidence of adaptation decisions made based on the knowledge. Self-adaptation is one prominent method to deal with uncertainty. When system objectives are violated, the self-adaptive system has to analyze all the available adaptation options, i.e., the adaptation space, and choose the best one. Yet analyzing the whole adaptation space using rigorous methods is time-consuming and computationally expensive. Recently machine learning based methods have been proposed to reduce the adaptation space. However, most of them require domain expertise to perform feature engineering and also labeled training data that are representative of the system environment, which may be challenging to obtain due to design-time uncertainty.

**Objectives:** This paper aims to propose a method that can manage uncertainty in self-adaptive systems with large adaptation spaces in an effective and efficient manner.

**Method**: To tackle this challenge, a method that integrates deep reinforcement learning with self-adaptive systems is developed, considering factors such as scalability, real-time decision-making, and resource constraints.

**Result:** Results show that the proposed method is effective with a negligible effect on the realization of the adaptation goals compared to a state-of-the-art method.

**Conclusion:** This article aims to investigate how deep reinforcement learning can be utilized to handle large adaptation spaces in self-adaptive systems efficiently and effectively. By developing a novel method integrating DRL algorithms with self-adaptive systems, this study seeks to improve exploration and exploitation capabilities of self-adaptive systems when dealing with uncertainty.

**Keywords:** self-adaptive systems, deep reinforcement learning, adaptation space reduction, adaptation option

1. Introduction

Modern software systems engineering is complex. An important factor underlying this complexity is the dynamic environment in which systems must operate, which requires systems to deal with uncertain conditions that are often difficult to predict before they become operational. These uncertainties may compromise the objectives of the system. Uncertainty in self-adaptive systems is any deviation of deterministic knowledge that may reduce the confidence of adaptation decisions made based on the knowledge [1]. For example, network interference can affect system availability if not handled properly.

Self-adaptation is one prominent approach to deal with uncertainty [2, 3]. A self-adaptive system can change its structure and behavior at runtime based on its understanding of the environment, the system and its requirements. The use of self-adaptive systems in various fields such as robotics [4], Internet of Things [5], cyber-physical systems [6], self-driving vehicles [7] and smart homes is increasing [8] .

In this paper, we deal with self-adaptive software systems with large discrete adaptation spaces. We use the term adaptation space as the set of all possible adaptation options at some point in time, i.e., all the possible configurations that can be reached from the current configuration of the system by applying a set of adaptation actions to the system. The size of the adaptation space may be constant over time, or it may change dynamically.

When faced with a large adaptation space, choosing an adaptation option that satisfies system objectives becomes computationally expensive and, in some cases, almost impossible [9, 10]. A common technique used to find the best adaptation option is to validate runtime models of the system and environment for one or more quality properties. These quality models have parameters that can be modeled for a specific adaptation option [5, 11, 12]. However, verifying these runtime models is a time-consuming process and in software systems where the system needs to react to the change in a limited amount of time it is not feasible. So, there must be better solutions to help us find the appropriate adaptation option in large adaptation spaces for time-constrained software systems.

Various techniques have been explored to find a suitable adaptation option in a large adaptation space. A specific technique to deal with the problem is adaptation space reduction, which aims to retrieve a subset of adaptation options that are capable of meeting the system's objectives. Techniques that use this method include search-based techniques [13], feature-based techniques [14] and machine learning techniques [15-19]. Among the aforementioned techniques, supervised machine learning techniques are more effective, but supervised learning requires labeled training data that are representative of the system environment, which may be challenging to obtain due to design-time uncertainty. In this article, we address the following research question:

“Can deep reinforcement learning (DRL) be used to determine an effective adaptation option from a large adaptation space and improve the decision-making process, allowing a self-adaptive system to conduct a more efficient analysis without compromising adaptation goals?”

Here, the meaning of effective is that the selected adaptation option meets the quality objectives of the system. By efficient we mean that the proposed method should ensure that: the time required for selecting an appropriate adaptation option should be small compared to the time required for analyzing the whole adaptation space.

In order for SASs to select a suitable adaptation option in an effective and efficient manner, we consider two steps, namely, the training step and the online step. For the first step, we use formal verification of runtime models together with a method based on DRL which through a series of interactions with the system learns a model that selects the appropriate adaptation option. For the second step, the learned model is integrated into the self-adaptive system and used as a decision-maker.

We used formal runtime models because they provide standard and precise descriptions of the system and environment. They have parameters that can be instantiated at runtime and used as a mechanism to approximate the values of quality properties. Then these values can be used as inputs for the DRL-based method to evaluate and select an appropriate adaptation option.

We used deep reinforcement learning for the following reasons: (1) Deep reinforcement learning is a type of machine learning that enables an agent to learn from its environment by taking actions and receiving rewards or penalties based on those actions. This method has been successful in various applications, including game playing, robotics, and natural language processing. (2) In self-adaptive systems, deep reinforcement learning can be used to select actions that optimize system performance and adapt to changing environments. This method allows the system to learn from its past experiences and make decisions based on the current state of the system. (3) Additionally, deep reinforcement learning can handle complex decision-making problems where traditional rule-based methods may not be effective. It can also handle situations where there is uncertainty or incomplete information about the environment. (4) Overall, using deep reinforcement learning for selecting actions in self-adaptive systems can lead to more efficient and effective decision-making, improved system performance, and better adaptation to changing environments.

Combining formal runtime models and DRL, we propose a new method to select an appropriate adaptation option called “Deep Reinforcement Learning for Selecting Suitable Adaptation Option”- DRL4SAO in short. We extend the MAPE-K loop by introducing a DRL-based module on top of the MAPE-K and an adaptation option predictor in the planner element that selects an adaptation option.

We evaluate the proposed method on two instances of DeltaIoT [20]. DeltaIoT is a product for evaluating autonomous systems in the field of Internet of Things. The Internet of Things (IoT) is a challenging field for autonomous applications due to its complexity and high degree of uncertainty. The difference between the two instances of DeltaIoT is in the size of their adaptation space, enabling us to evaluate different aspects of effectiveness and efficiency. For this purpose, we define appropriate criteria to evaluate the proposed method and compare it with a state-of-the-art method and a reference method that examines all adaptation options and selects the best option.

The contributions of this article are: (1) A DRL-based method to select appropriate adaptation options. (2) a thorough assessment of the effectiveness and efficiency of the method in the IoT domain, including a comparison with a state-of-the-art method and a reference method.

1. background

DRL4SAO uses a DRL-based method that leverages runtime models of the system and environment to receive feedback from the system and learns based on them. In this section, the necessary background for this article is introduced. First, we introduce MAPE-K loop and DRL and then we explain how the analysis of selected adaptation options are performed using statistical model checking.

* 1. MAPE-K Loop

An important milestone in the field of self-adaptive systems is the conceptual reference model proposed by IBM which at that time was called Autonomic manager [21]. Autonomic manager is placed on top of the managed system and its main functionality is to adapt the managed system autonomously. The different functions represented by this model are as follows: 1) Knowledge: The first function is the Knowledge which is shared among all the other parts and basically it keeps representation of the underlying system, its context and the goals that needs to be achieved. 2) Monitor collects data from managed system and context at runtime and keeps the Knowledge up-to-date with respect to the managed system. 3) Analyzer looks at the Knowledge and determines need for adaptation and if so, it analyses the adaptation options. 4) If adaptation is needed Plan will determine a plan, meaning what are the actions that it needs to take in order to restore system to a desired state with respect to objectives. 5) Finally, the Execute element enacts the plan adapting the managed element.

* 1. Deep Reinforcement Learning

DRL is a subfield of machine learning that combines deep learning techniques with reinforcement learning algorithms [22]. Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal of the agent is to learn a policy that maximizes the cumulative reward over time [23].

DRL extends traditional reinforcement learning by using deep neural networks to approximate the value function or policy function Q(s,a;). Here are the parameters (that is, weights) of the Q-network at iteration i. To perform experience replay the agent’s experiences are stored in a dataset et = (st, at, rt, st+1) at each timestep t in a dataset Dt = (e1, …, et). During learning, Q-Learning updates are applied on samples (or minibatches) of experience (s, a, r, s′) ~ U(D), drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function (1):

2

Li() = E(s,a,r,s′) ~ U(D) – Q (s, a; ) (1)

In which is the discount factor determining the agent’s horizon, θi the parameters of the Q-network at iteration i and are the network parameters used to compute the target at iteration i. The target network parameters are only updated with the Q-network parameters θi every C step and are held fixed between individual updates.

* 1. Analysis of adaptation options

The model of the system and environment are designed as networks of timed automata using UPPAAL-SMC [24]. UPPAAL-SMC allows for the efficient analysis of performance properties of networks of timed automata under a natural stochastic semantics. At runtime, these models are directly executed using ActivFORMS execution engine [25]. ActivFORMS guides the adaptation of the system at runtime by analyzing selected adaptation options to assess the realization of adaptation goals with a required level of accuracy and confidence in an efficient manner using statistical model verification techniques at runtime.

1. Related Work

Over the past few years, there has been an increasing interest in the use of reinforcement learning and search-based methods for self-adaptation in software systems. For the adaptation space reduction, we classify the methods used in three main categories: Classic and deep machine learning based methods [17, 18, 26, 27], Search-based methods [29-34] and Classic and deep reinforcement learning based methods [35-43]

* 1. Classic and deep machine learning-based methods

Quin et al. [17] present a machine learning method to reduce large adaptation spaces. Their method enhances the traditional MAPE-K feedback loop with a learning module that selects subsets of adaptation options from a large adaptation space to support the analyzer with performing efficient analysis. It is instantiated for two concrete learning techniques, classification and regression, and evaluated for two instances of an Internet of Things application for smart environment monitoring with different sizes of adaptation spaces. The evaluation shows that both learning methods reduce the adaptation space significantly without noticeable effect on realizing the adaptation goals.

In [18, 26, 27], a Deep Learning-based method is proposed. First a deep learner is applied to reduce the adaptation space for the threshold goals and then ranks these options for the optimization goal; the adaptation options are verified one by one with respect to their predicted ranking until an option that complies with the threshold goals is encountered.

Jamshidi et al [16] use an method where a set of Pareto optimal configurations are learned offline and then used at runtime to generate adaptation plans. Their method reduces adaptation spaces, while the system can still apply model checking with PRISM [28] at runtime to quantitatively reason about adaptation decisions. This method is explored in the context of robot missions that need to consider task timeliness and energy consumption. Their evaluation results shows promising results as it selects high-quality adaptation plans in uncertain and adversarial environments. However, it is important to note that the effectiveness of the method can be influenced by the quality and relevance of the machine learning model that is being used. If the model is not well-suited to the new conditions, it might not result in optimal adaptation. Furthermore, restricting the search space to Pareto-optimal configurations might exclude potentially beneficial configurations that are not Pareto-optimal.

In conclusion, Classic and deep machine learning-based methods require labeled training data representative of the system’s environment, which may be challenging to obtain due to design time uncertainty.

* 1. Search-based methods

Coker et al. [29] discussed the need for improved planning techniques in self-adaptive systems. Next-generation self-adaptive systems must be optimized for multiple relevant, difficult-to-measure and evolving quality attributes. To navigate this complex search space, the authors argue that the research community should more directly pursue the application of stochastic search techniques. These techniques, such as hill climbing or genetic algorithms, incorporate an element of randomness and are well-suited to handling multi-dimensional search spaces and complex problems. The authors believe that recent advances in both fields have made this line of research particularly promising. They show one way of applying some of these advances to a search-based planning prototype technique to demonstrate the feasibility and potential of the proposed research. This strategy suggests a number of potentially interesting research directions and problems. In the long term, this general technique can provide sophisticated techniques for generating designs that improve domain knowledge, reduce human effort, and increase the utility of self-adaptive systems.

Authors in [30-32] propose a planner based on genetic programming that reuses existing plans. Their method uses stochastic search to deal with unexpected adaptation strategies, specifically by reusing or building upon prior knowledge. Their genetic programming planner is able to handle very large search spaces. However, it is important to note that although the method of reusing prior planning knowledge to adapt to unexpected situations can be beneficial, it also has potential limitations. For example, the effectiveness of the method can be influenced by the quality and relevance of existing plans that are reused. If these plans do not adapt well to new conditions, their reuse may not lead to optimal adaptation. Furthermore, the paper points out that naively reusing existing plans for self-adaptation can actually lead to a loss of utility. This suggests that further investigation and potentially additional techniques are needed to ensure that plan reuse is beneficial.

Chen et al.[33] presented a framework called “Feature-guided and Knee-driven Multi-objective Optimization for Self-adaptive Software”. (FEMOSAA) automatically synergizes the feature model and Multi-Objective Evolutionary Algorithm (MOEA), to optimize Self-Adaptive Software (SAS) at runtime. FEMOSAA operates in two phases: At design time, FEMOSAA automatically transposes the engineers’ design of SAS, expressed as a feature model, to fit the MOEA, creating new chromosome representation and reproduction operators. At runtime, FEMOSAA utilizes the feature model as domain knowledge to guide the search and further extend the MOEA, providing a larger chance for finding better solutions. The effectiveness of FEMOSAA is evaluated on two running SAS: one is a highly complex SAS with various adaptable real-world software under realistic workload trace, another is a service-oriented SAS that can be dynamically composed from services. The results reveal the effectiveness of FEMOSAA and its superiority over other search-based frameworks for SAS under various scenarios. However, the effectiveness of the method can be influenced by the quality and relevance of the feature model and MOEA that are being used. If these are not well-suited to the new conditions, they might not result in optimal adaptation. Furthermore, restricting the search space to knee solutions might exclude potentially beneficial configurations that are not knee solutions.

Pascual et al. [34] use a genetic algorithm to automatically generate adaptation plans to adapt a system along with reconfiguration schemes. The generated plans are optimal in terms of performance considering the available resources (eg, battery). Specifically, plans are defined as modifications of the application software architecture based on the so-called feature model. They used a case study involving an application that assists participants in international congresses, keeps them up-to-date with the latest news, and provides several social amenities.

* 1. Classic and deep reinforcement learning-based methods

Kim et al. [35] proposed a reinforcement learning-based method for online planning in self-adaptive systems. This method enables a software system to improve its behavior by learning from the results of its behavior and dynamically change its plans based on this learning in the presence of environmental changes. This paper presents a case study based on a robot battle simulator to illustrate the method, and its results show that online planning based on reinforcement learning is effective for architecture-based self-adaptation.

Camara et al. [36] proposed a method called RLMC (Reinforcement Learning and Model Checker) for architecting Internet of Things (IoT) systems that can guarantee Quality of Service (QoS) levels. Their method combines RL and formal quantitative verification (probabilistic model checking). In this method, RL is tasked with selecting the best adaptation option, and quantitative verification checks the feasibility of the adaptation decision. This avoids the execution of infeasible adaptations and provides feedback to the ML engine that helps achieve faster convergence towards optimal decisions. Their evaluation results show that their method can produce better decisions than ML and quantitative verification used in isolation.

Metzger et al. [37] use an RL-based method to address the problem of exploration-exploitation problem in Self-adaptive software systems. In their exploration strategies they use feature models to organize the system’s adaptation space and thereby leverage additional information to guide exploration. In addition, their strategies detect added and removed adaptation actions by analyzing the differences between the feature models of the system before and after an evolution step. Adaptation actions removed as a result of evolution are no longer explored, while added adaptation actions are explored first.

Caporuscio et al. [38] proposed a self-organizing fully decentralized solution for the service assembly problem. The aim of ensuring good overall quality for the services provided is to ensure fairness among participating peers. The main features of this solution are: (1) The use of a gossip protocol to support decentralized information dissemination and decision making. (2) The use of a reinforcement learning method to make each peer able to learn from its experience the service selection rule to be followed, thus overcoming the lack of global knowledge. Additionally, they explicitly take into account load-dependent quality attributes, which lead to the definition of a service selection rule that drives the system away from overloading conditions that could adversely affect quality and fairness. Simulation experiments show that their solution adapts to changes by rapidly converging to viable sets while maintaining the specified quality and fairness objectives.

Barret et al. [39] discussed the challenges faced by cloud services, such as Amazon, which deliver computational resources through virtualization technologies. These technologies allow multiple independent virtual machines to reside in apparent isolation on the same physical host. However, dynamically scaling applications running on IaaS clouds can lead to varied and unpredictable results due to the performance interference effects associated with co-located virtual machines. Determining appropriate scaling policies in a dynamic non-stationary environment is non-trivial. The authors propose a solution to this problem by applying a temporal difference, reinforcement learning algorithm known as Q-learning, to determine optimal scaling policies. This method helps decide which resources should be added and removed when the underlying performance of the resource is in a constant state of flux. However, reinforcement learning techniques typically suffer from curse of dimensionality problems, where the state space grows exponentially with each additional state variable. To address this challenge, the authors present a novel parallel Q-learning method aimed at reducing the time taken to determine optimal policies while learning online.

Arabnejad et al. [40] compared two dynamic learning strategies based on a fuzzy logic system. The key problem addressed is how and when to add/remove resources in order to meet agreed service-level agreements. Reducing application cost and guaranteeing service-level agreements (SLAs) are two critical factors of dynamic controller design. The authors proposed a self-adaptive fuzzy logic controller combined with two reinforcement learning (RL) methods: (1) Fuzzy SARSA learning (FSL). (2) Fuzzy Q-learning (FQL). Both methods are implemented and compared in their advantages and disadvantages in the OpenStack cloud platform. The authors demonstrate that both auto-scaling methods can handle various load traffic situations, sudden and periodic, and deliver resources on demand while reducing operating costs and preventing SLA violations.

Mustafa et al. [41] proposed two novel algorithms that aim to achieve greater data efficiency by saving experience data and using it in aggregate to make updates to the learned policy. The first algorithm introduces an offline learning scheme for service composition where the online learning task is transformed into a series of supervised learning steps. This method is designed to enhance the efficiency of reinforcement learning algorithms, which are commonly used to compose and adapt Web services in open and dynamic environments. However, these algorithms are often relatively inefficient in their use of experience data, which may affect the stability of the learning process. By saving experience data and using it in aggregate to make updates to the learned policy, the proposed algorithms aim to overcome this limitation and improve the efficiency of the learning process.

Zhao et al. [42] discussed the challenges in self-adaptive systems and how to make adaptations at runtime in response to possible and even unexpected changes from the environment and/or user goals. A feasible solution to this challenge is rule-based adaptation, where adaptation decisions are made according to predefined rules that specify what particular actions should be performed to react to different changing events from the environment. However, rule-based adaptation has two limitations: (1) There's no guarantee that those predefined rules will lead to optimal or nearly-optimal adaptation results. (2) There's weak support to evolve these rules to cope with non-stationary environment and changeable user goals at runtime. To address these limitations, the authors proposed a reinforcement learning-based framework for the generation and evolution of software adaptation rules. This framework has two key capabilities for self-adaptation: (1) The capability of automatically learning adaptation rules from different goal settings at the offline phase. (2) The capability of automatically evolving adaptation rules from real-time information about the environment and user goals at the online phase. These capabilities are built on the combination of reinforcement learning and case-based reasoning techniques. The authors evaluate this framework through a case study of an E-commerce web application, which shows that this framework improves both the efficiency and effectiveness of self-adaptation.

Shaw et al. [43] discussed the application of reinforcement learning to automate the process of energy-efficient virtual machine consolidation in cloud data centers. The authors proposed a reinforcement learning-based method to automate this process. This method enables the system to learn from its experience and dynamically adjust its actions based on this learning.

In conclusion, none of the methods reviewed above address the problem of large adaptation spaces. The main problem with them is that all of them use classic reinforcement learning methods such as Q-Learning or Sarsa. These classic RL methods cannot handle large adaptation spaces and are suitable for problems where the size of adaptation space is small.

1. Proposed method

Figure 1 shows the basic architecture of the proposed self-adaptive system that uses DRL for predicting an appropriate adaptation option (DRL4SAO). In this research we apply architecture-based adaptation with a MAPE-K feedback loop.

The figure highlights the main elements of the architecture and high-level flow of interactions between the elements. The managed system takes input from the environment and produces output to realize the user goals. The managing system controls the managed system to achieve a set of adaptation goals. Central to the managed system are the MAPE elements that share knowledge and realize a feedback loop. The feedback loop senses the managed system and adapts it to achieve the adaptation goals.

The main responsibility of each element is described as follows: The Monitor tracks the uncertainties and properties of the underlying managed system and sends the collected data to the Analyzer (1). When the Analyzer detects that the adaptation goals are violated or may no longer be achievable, it notifies the Adaptation option predictor component embedded in the planner (2). The Adaptation option predictor then retrieves the model from the trained DRL model repository and predicts an appropriate adaptation option (3). The index of the predicted adaptation option is sent to the executor (4). Finally, the Executor retrieves the predicted adaptation option from the Knowledge repository (5) and adapts the managed system (6).

In the remainder of this section, we explain how the Adaptation option predictor component is trained and how the component predicts an adaptation option at runtime.

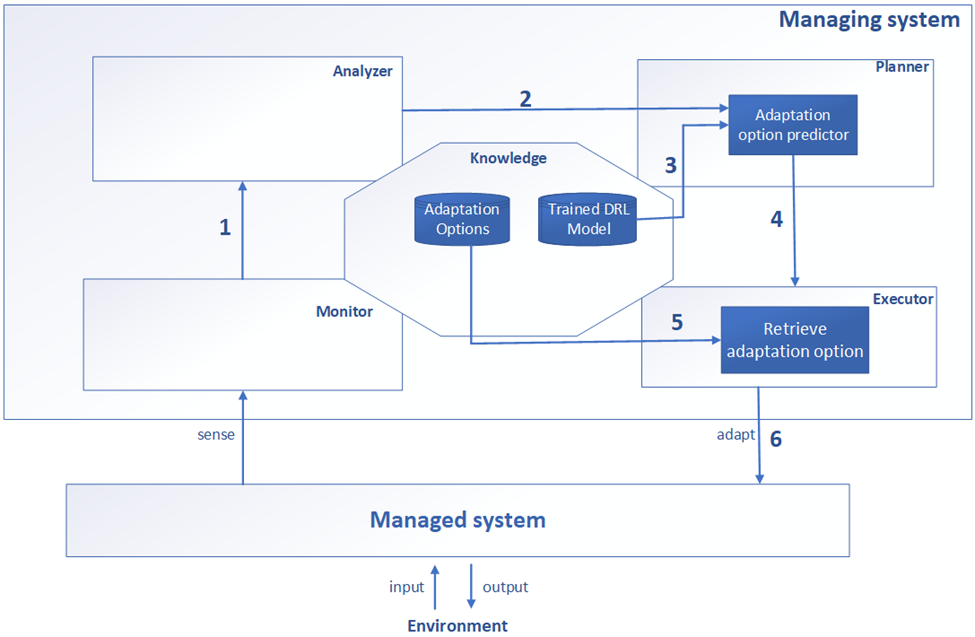


Figure : The Architecture of the DRL4SAO

* 1. The integration of DRL with MAPE-K

Figure 2 shows the integration of DRL with the mape-k loop. The training DRL module sits on top of the mape-k loop. The runtime stage works in cycles, each representing an opportunity for the system to perform adaptation. The training process is performed for a certain number of cycles. At the beginning of each training cycle, the monitor component of the mape-k collects required runtime data (1) and sends them to the deep RL module (2). Then for the duration of the training cycle the following steps are done. Based on received input vectors the deep RL module starts the training process. At the end of the training process the trained model is stored in the knowledge repository of the MAPE-K loop (3). The monitor component send the gathered data to the Analyzer (4). When the Analyzer detects that the adaptation goals are violated or may no longer be achievable, it notifies the Adaptation option predictor component embedded in the planner (5). The adaptation option predictor retrieves the model (6) and predicts an adaptation option (7). The Executor retrieves the predicted adaptation option from the adaptation options repository (8) and applies it to the managed system (9) and the whole process of training will be repeated for the next adaptation cycle.

* 1. The training process of the deep reinforcement learning module

The training stage starts with the collection of data from the managed system and its environment. This data captures information relevant to the adaptation of the system over a period of time. This includes properties in the environment that affect the behavior of the system, system configurations, and quality properties. The extracted features are then used by the Adaptation option selector for the identification of an adaptation option. The output of the Adaptation option selector is sent to the Adaptation option verifier where quality properties of the system are calculated and are forwarded to the Reward Calculator. Based on received quality properties the Reward Calculator calculates a reward and sends the reward along with features and selected adaptation option to the Replay memory. A mini batch of data is collected from the Replay memory and training the model is performed. If it is the end of the cycle then the trained model is stored in the Trained models repository, otherwise the whole training process is repeated.

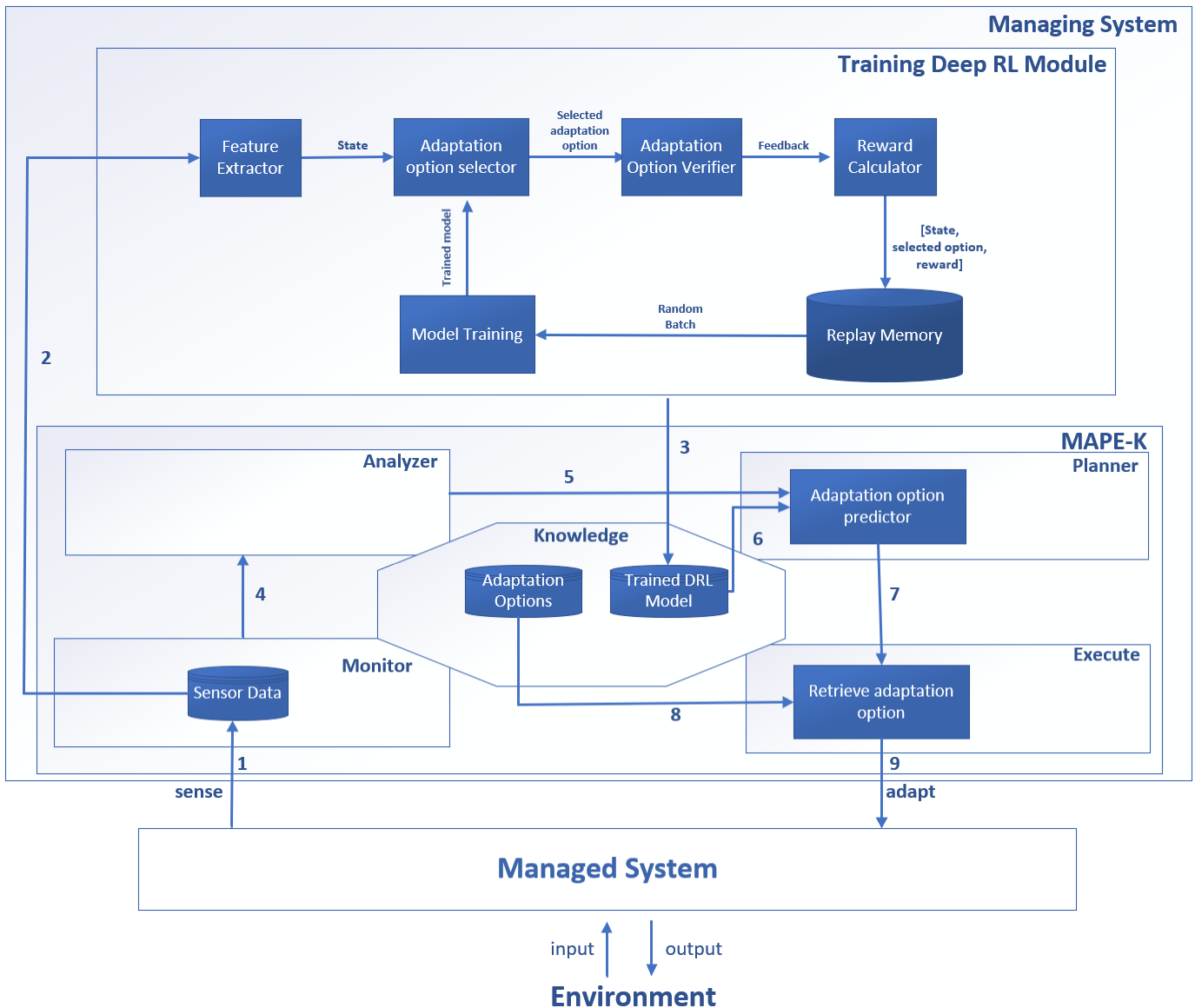


Figure : Runtime integration of DRL4SAO with MAPE-K

* + 1. Adaptation option selector

Adaptation option selector is based on an Epsilon-greedy policy. Epsilon-greedy policy is a common strategy used in reinforcement learning to balance exploration and exploitation. It is a way to decide whether an agent should take a random action (explore) or choose the action with the highest estimated value (exploit). The policy works by introducing a parameter called epsilon (ε), which represents the probability of exploration. At each decision point, the agent generates a random number between 0 and 1. If this number is less than epsilon, the agent chooses a random option from the available options, exploring the environment. However, if the generated number is greater than or equal to epsilon, the agent selects an adaptation option with the highest estimated value based on its current knowledge, exploiting what it has learned so far.

The purpose of using epsilon-greedy policy is to encourage exploration in order to discover potentially better adaptation options that may not have been tried before. By occasionally selecting random adaptation options, even when there are options with higher estimated values, the agent can gather more information about its environment and improve its overall performance.

In order to predict an adaptation option (i.e., exploitation), the adaptation option selector uses a deep neural network. The deep neural network used is typically a type of artificial neural network called a deep Q-network [22] (DQN).

A DQN is a multi-layered neural network that takes in the current state of the environment as input and outputs the expected value of each possible action. It learns to estimate the optimal action-value function, also known as Q-function, which represents the expected cumulative reward for taking a particular action in a given state.

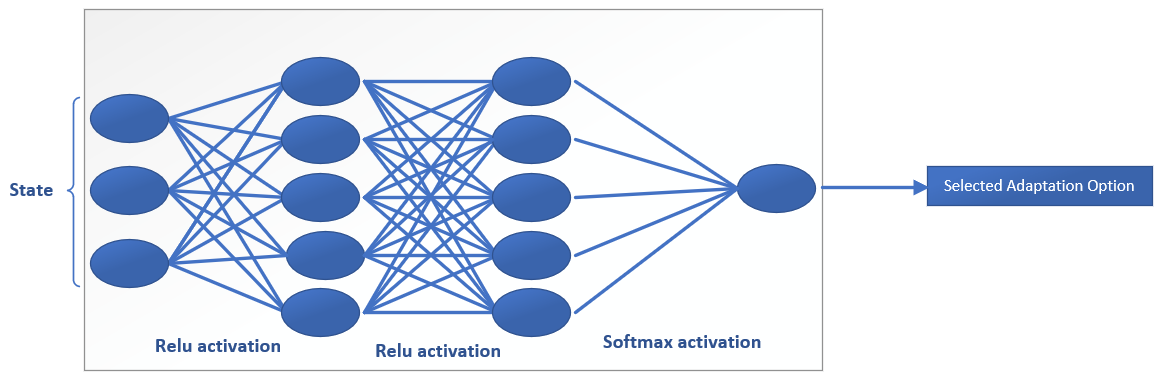


Figure : The architecture of DQN-network used in DRL4SAO

The architecture of a DQN consists of several layers of neurons, including input, hidden, and output layers. The input layer receives the state representation, which can be an image or a numerical vector. The hidden layers perform nonlinear transformations on the input data, extracting relevant features and representations. Finally, the output layer produces Q-values for each possible action. It is worth noting that in our proposed method the number of hidden layers, activation functions used in each layer, and the number of neurons in each layer are hyper-parameters which are determined through a randomized grid search.

During training, DRL employs an algorithm called Q-learning to update the weights and biases of the DQN. Q-learning uses a combination of exploration and exploitation to iteratively improve the network's estimates of Q-values. The agent interacts with the environment by taking actions based on its current policy (e.g., epsilon-greedy), observes rewards and new states, and updates its Q-values accordingly.

* + 1. Adaptation option verifier

Adaptation option verifier consists of a MAPE feedback loop and a set of quality properties designed as networks of timed automata models. These models are directly executed at runtime using the ActivFORMS execution engine [44]. The analysis of the selected adaptation option is performed using the runtime models by applying statistical model checking at runtime using runtime statistical model checking with Uppaal-SMC [24] .The resulting estimates of the quality properties of the selected adaptation option are then used by the Reward calculator to return a reward.

* + 1. Replay memory

The purpose of a replay memory in deep reinforcement learning is to store and manage the experiences or transitions (state, action, reward, next state) encountered by an agent during its interaction with the environment [22]. It acts as a memory that allows the agent to learn from past experiences and improve its decision-making process. The replay memory serves several important functions:

a) Experience Replay: By storing past experiences, the replay memory enables the agent to learn from a diverse set of transitions rather than just the most recent ones. This helps in breaking any temporal correlations present in consecutive experiences and reduces bias in learning.

b) Data Efficiency: The replay memory allows for more efficient use of data by reusing past experiences multiple times during learning. Instead of discarding each experience after it is used for a single update, it can be sampled multiple times, leading to better sample efficiency.

c) Batch Learning: The replay memory facilitates batch learning by allowing the agent to sample mini-batches of experiences randomly or using prioritized sampling. This helps in stabilizing and improving the learning process by reducing the variance of updates and avoiding overfitting on individual transitions.

d) Off-Policy Learning: The replay memory enables off-policy learning, where an agent learns from experiences generated by a different policy than the one currently being improved. This allows for more stable and safer learning as it decouples exploration from exploitation.

Overall, the purpose of a replay memory is to enhance the efficiency, stability, and effectiveness of deep reinforcement learning algorithms by providing a mechanism for experience storage, reuse, and random sampling during training.

* + 1. Model training

A batch of experiences is sampled from the replay memory. Using this batch, the network parameters are updated through backpropagation to minimize a loss function (see section ‎2.2) that reflects how well the current Q-values or policy approximations match with observed rewards.

1. evaluation

For the implementation of the DRL4SAO, we used the tensorflow library for the implementation of the deep reinforcement learning agent. We ran the simulated IoT network and the training of the deep reinforcement learning models on i7-10750H CPU @ 2.60GHz 2.59 GHz with 16.0 GB of Ram.

* 1. Case Study

DeltaIoT is an Internet-of-Things application developed by VersaSense[[2]](#footnote-2) [20] (Figure 4), which consists of a set of sensors[[3]](#footnote-3).

Each sensor is equipped by a battery-powered mote[[4]](#footnote-4) which takes care of the routing of sensor data to the gateway. Data from these sensors is transmitted through a wireless multi-hop communication[[5]](#footnote-5) network to a gateway connected to a user application. It is expected that motes can operate on a single battery for a long time while at the same time offering a reliable communication with minimum delay and packet loss.

The main sources of uncertainty in DeltaIoT are network interference and noise caused by external factors and message load fluctuations and are used to determine the QoS of the system together with system characteristics such as the transmission power of motes and packet distribution. Minimizing energy consumption, packet loss and latency are the most important objectives of the system. The main cause of network interference is the dynamicity of the environment such as changing weather conditions or construction work in the neighborhood which is difficult to predict upfront. As for message load fluctuations it depends on the frequency by which sensors take samples and transmit data. For example, RFID sensors collect and send data when there is data, but temperature sensors send data periodically. In DelatIoT, Network interference varies between −40 dB and +15 dB and traffic loads range from 0 to 10 messages per mote.

We use two instances of DeltaIoT with different number of adaptation options. Adaptation options are composed in each cycle following two steps. Firstly, the power setting is determined for each link of each mote. These settings are determined such that the current Signal to Noise ratio (SNR) over each link is at least 0 dB. The adaptation options are then determined based on the possible distribution settings for outgoing links of motes with two parents (0–100, 20–80, etc.). As such, the complete adaptation space for the DeltaIoTv1 is 63 = 216 and for DeltaIoTv2 it is 46 = 4096 adaptation options. Figure 5 shows a representation of the adaptation space for DeltaIoTv2 case study at some point in time.

The red lines denote two threshold goals for this particular scenario (latency and packet loss). Each blue dot in this graph represents one adaptation option. For this particular instance, the number of adaptation options is 4096. Analyzing the whole adaptation space using rigorous methods within a specified time slot may not be feasible. So, the problem becomes how to identify adaptation options that meet the quality requirements of the system and choose a suitable one between them.

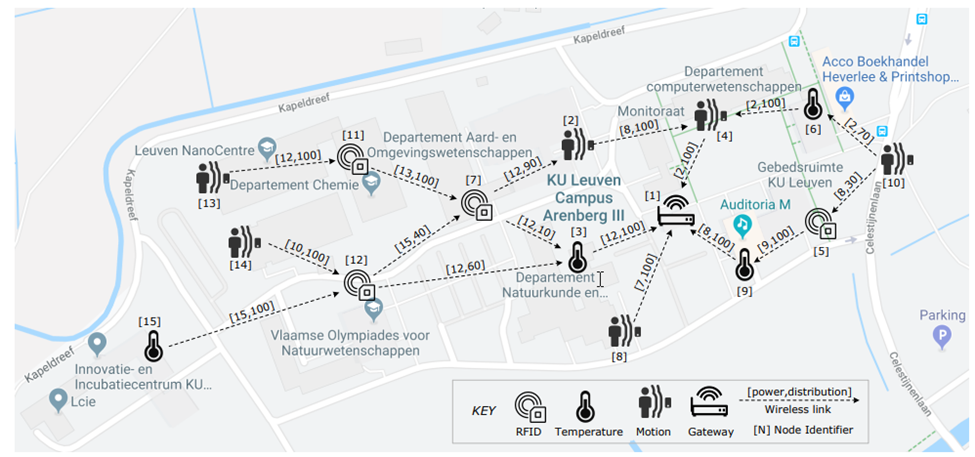


Figure : DeltaIoTv1 [20]

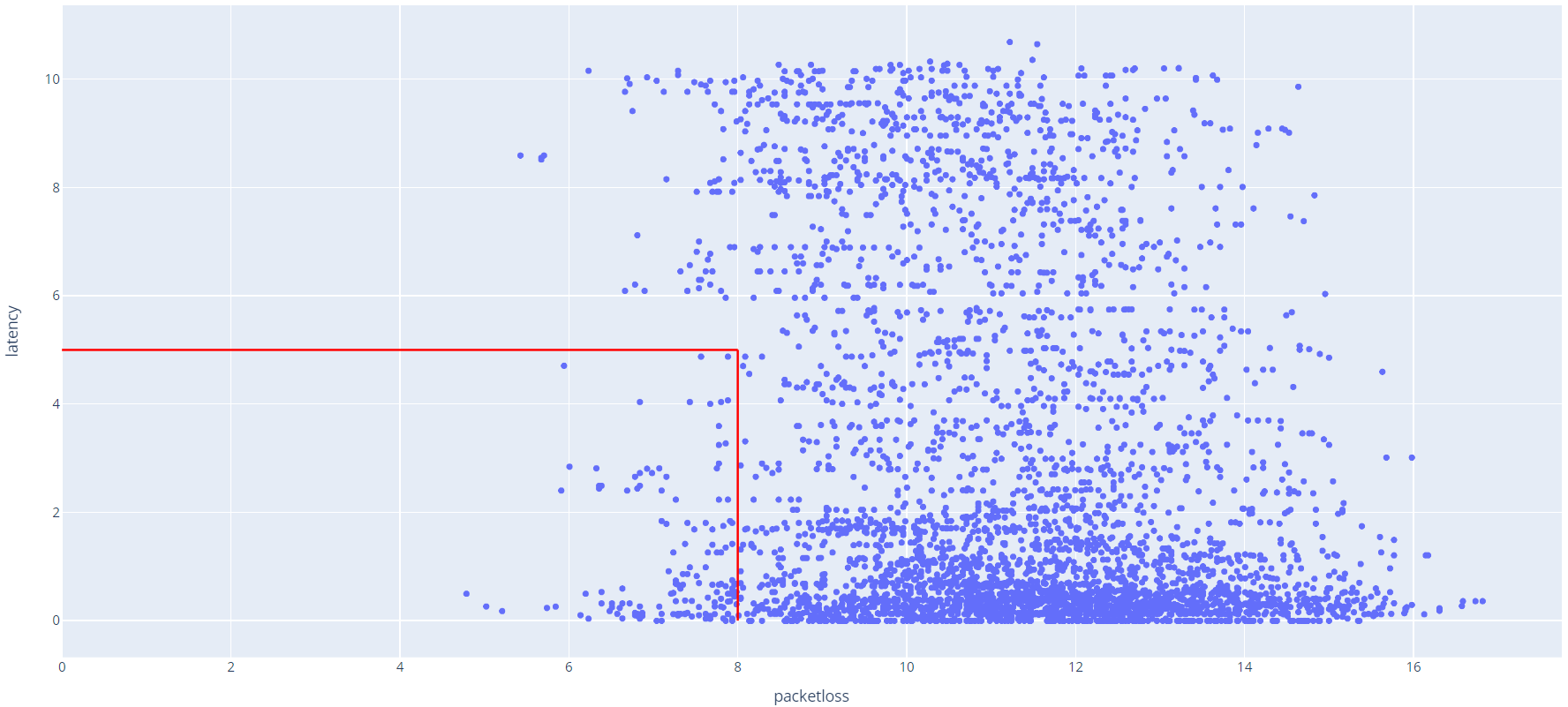


Figure . Adaptation space of DeltaIoTv2 at a point in time

* + 1. Goals

Our proposed method has the ability to handle different kind of goals. The goals supported in this paper are as follows:

(1) Threshold goals, A threshold goal requires that some quality property should either remain below or above a given value. The satisfaction of a threshold goal t ε T with a threshold value x for any value of the quality property q is defined by equations 2 and 3:

t < x (q) = (2)

t > x (q) = (3)

(2) Setpoint goals, A set-point goal states that some value of a quality property of the system should be kept at a certain value, i.e., the set-point, with a margin of at most 𝜖. the satisfaction of a set-point goal s ε S with target μ and error margin 𝜖 is defined by equation 4:

(q) = (4)

(3) Optimization goals, an optimization goal states that some value of a quality property of the system should be minimized (or maximized). the satisfaction of an optimization goal o ∈ O for any quality value q is defined by equations 5 and 6:

Omin (q) = (5)

Omax (q) = (6)

Table 1 shows the adaptation goals used for the evaluation of DeltaIoT: TTS (2 Threshold goals and 1 Set-point goal), TTO (2 Threshold goals and 1 optimization goal).

Table : Overview of adaptation goals used for the evaluation of DeltaIoTv1 and DeltaIoTv2 [20]. The adaptation goals are defined for packet loss (PL), latency (LA), and energy consumption (EC).

|  |  |  |
| --- | --- | --- |
| Type | Goals DeltaioTv1 | Goals DeltaioTv2 |
| TTS | T1: PL < 10%  T2: LA < 15%  S1: EC in | T1: PL < 10%  T2: LA < 15%  S1: EC in |
| TTO | T1: PL < 10%  T2: LA < 15%  O1: Minimize EC | T1: PL < 10%  T2: LA < 15%  O1: Minimize EC |

* + 1. metrics

In order to evaluate DRL4SAO in terms of efficiency and effectiveness, we use the following metrics [18]:

(1) Average adaptation space reduction (AASR) captures the efficiency of our proposed method. 𝐴𝐴𝑆𝑅 is defined by equation (6):

(6)

where 𝑠𝑒𝑙𝑒𝑐𝑡𝑒𝑑 is the number of adaptation options selected by learning (over multiple adaptation cycles) and 𝑡𝑜𝑡𝑎𝑙 is the total number of adaptation options (of multiple adaptation cycles). For instance, an average adaptation space reduction of 70% means that after learning only 30% of the original adaptation space is considered for analysis.

(2) *Learning time* overhead (LTO) is the proportion of additional time introduced by DRL4SAO at runtime. Here the objective is to minimize LTO and it is defined by equation (7):

(7)

with Tt time to verify total adaptation space, Tr time to verify reduced adaptation space and To time overhead introduced by DRL4SAO.

(3) *Overall time saved* (OTS) is the proportion of total time saved of DRL4SAO (taking into account overhead) compared to the reference method. Here the objective is to maximize OTS and it is defined by equation (8):

(8)

with Tr time to verify reduced adaptation space and To time overhead introduced by DRL4SAO.

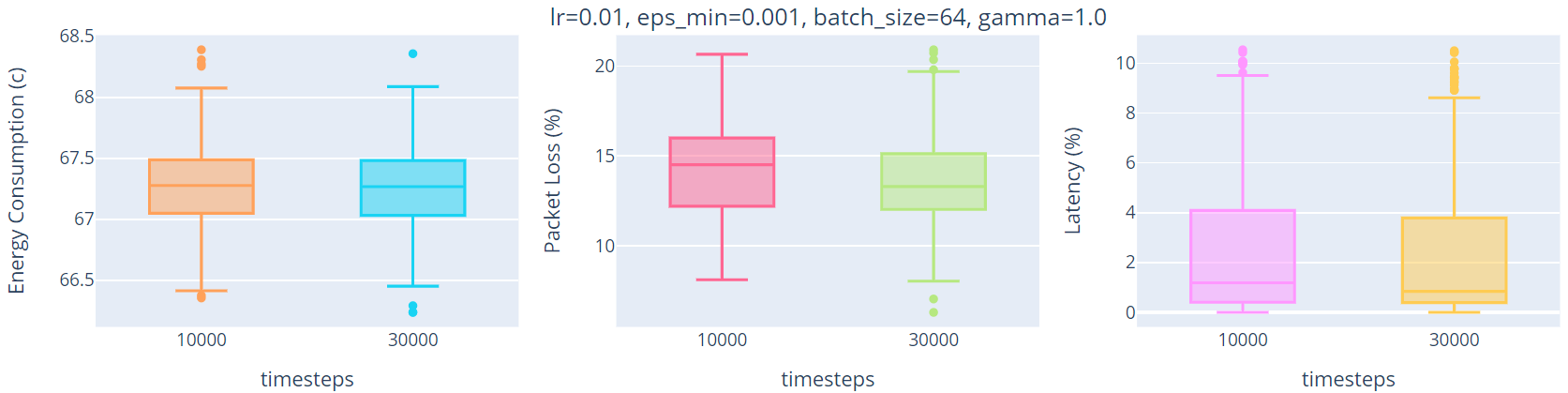
(4) *Penalty distance (PD)* refers to the tendency that the method chooses an adaptation option with little or no penalty on the quality properties that are the subject of adaptation, compared to an ideal solution where the whole adaptation space is analyzed. Here, distance refers to the difference in the mean values of the relevant quality properties over time with and without learning. Here the objective is to minimize UT and it is defined by equation (9):

(9)

Where n is the total number of cycles, is the value of the best adaptation option and is the value of adaptation option selected by our method.

* 1. Evaluation of the proposed method

To determine the best DRL model in terms of effectiveness and efficiency, we performed hyper-parameter tuning. The main hyper-parameters that apply for DRL architectures are listed in Table 2. In order to demonstrate how the best hyperparameters were chosen, we consider DeltaIoTv1 and DeltaIoTv2 and select the best hyperparameters obtained through experiments for the TTS goal. First, we select one of the hyperparameters, that is, the learning rate (LR) and change its value, leaving the rest of the hyperparameters unchanged. Then we train the model for 50000 time-steps for DeltaIoTv1 and 150,000 time-steps for DeltaIoTv2 and stop training at every 10,000 time-steps and evaluate the performance of the model trained so far according to the system goals. If the trained model performs better than the previous models, we save the model and otherwise continue with the rest of the training time steps. Table 3 shows the different values assigned to LR and Figure 6 and Figure 7 show the results with respect to quality properties.

In order to explain the process of selecting the best DRL model. we consider the case where the LR value is set to 1e-4. After the first 10,000 training steps, the evaluation step is performed and since it is the first evaluation model and there is no previous model to compare it with, it is saved as the best model obtained so far. Then a second evaluation is performed at 20,000 time-steps and the resulting model is compared with the first model, in this particular case the second model trained at 20,000 time-steps outperforms the first model and is therefore saved as the best model. At time step 30,000, the third round of evaluation is performed. and since the obtained model does not perform better than the current best model, it is ignored and we continue training. Finally, the training is stopped at time step 50000 and the best model, which in this case is the model obtained at time step 20,000, is reported back. A similar process is done with respect to other LRs and the impact that each model can have on the goals of the system can be seen in Figure 6. The box plots in Figure 6 show the impact that each different LR value can have on the values of quality properties. From Figure 6Figure 7 ,it can be observed that, in environments where multiple goals are considered, the selection of the best model with respect to different hyperparameters depends on the priorities of stakeholders of the system. They need to determine which quality properties are most important and then choose the best model accordingly. For example, in the case of DeltaioTv1, if energy consumption is of highest importance, then the model trained with the LR of 1e-4 should be chosen, if packet loss is more important, then the model with LR of 1e-2 should be chosen and if latency is more important, then the model with LR of 1e-1 should be selected. Similarly for the rest of hyper-parameters we performed a similar experiment. The experiment results for the rest of hyper-parameters are listed in 

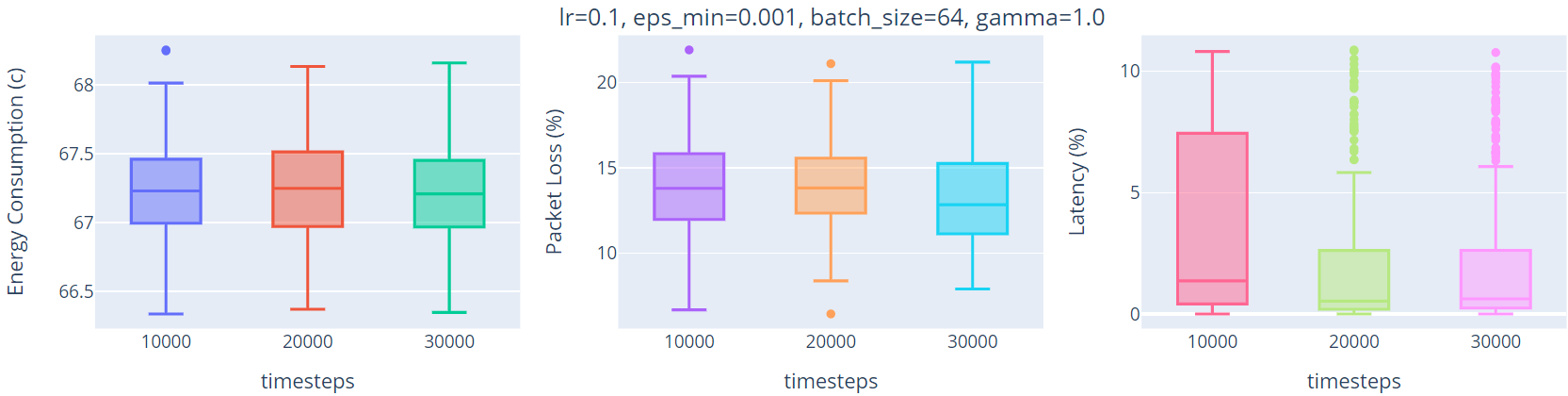


Figure 7: Values of different quality properties when experimenting with different LR values for DeltaIoTv2

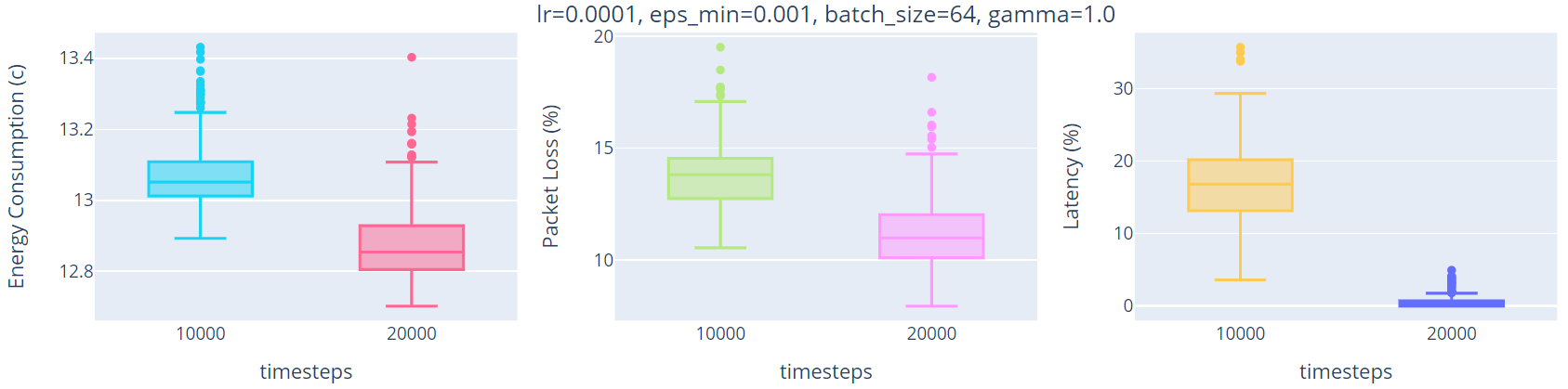
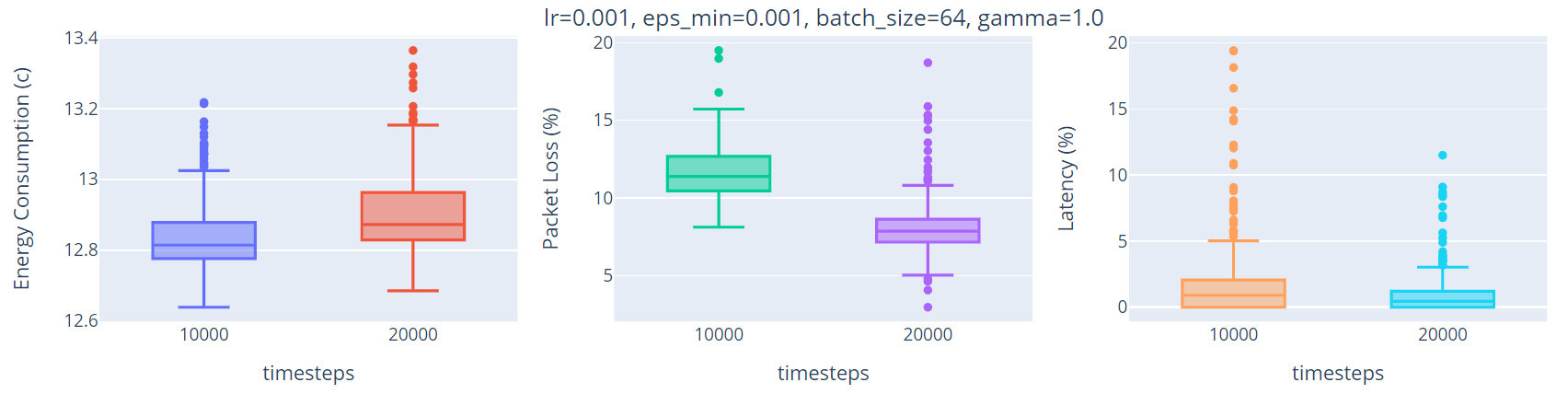
, Table 4 and Table 5.

Table : List of used hyper-parameters

|  |  |
| --- | --- |
| Name | Description |
| Learning rate (LR) | used to govern the pace at which an algorithm updates or learns the values of a parameter estimate, defined in the interval [0, 1]. |
| Discount factor (DF) | a constant to reflect the value of the reward signal over time, defined in the interval [0, 1]. |
| Epsilon decay rate (EDR) | used to determine how much the agent should explore and exploit when using epsilon-greedy policy, defined in the interval [0, 1]. |
| Batch size (BS) | defines how much of the data samples are considered before updating the model. Batch size considered is chosen from numbers [32, 64, 128, 256]. |

Table : Hyper-parameters used when experimenting with LR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem** | **Goals** | **Hyper-parameters** | | | | **Median values of different quality properties** | | |
|  | | BS | DF | EDR | LR | EC | PL | LA |
| DeltaIoTv1 | TTS | 64 | 1 | 1e-3 | 1e-4 | 12.85 | 10.98 | 0.33 |
| 1e-3 | 12.87 | 7.85 | 0.44 |
| 1e-2 | 12.90 | 10.35 | 0.34 |
| 1e-1 | 12.87 | 10.62 | 0.12 |
| DeltaIoTv2 | 1e-4 | 67.19 | 13.60 | 0.77 |
| 1e-3 | 67.27 | 13.49 | 0.70 |
| 1e-2 | 67.26 | 13.29 | 0.61 |
| 1e-1 | 67.20 | 12.84 | 0.84 |

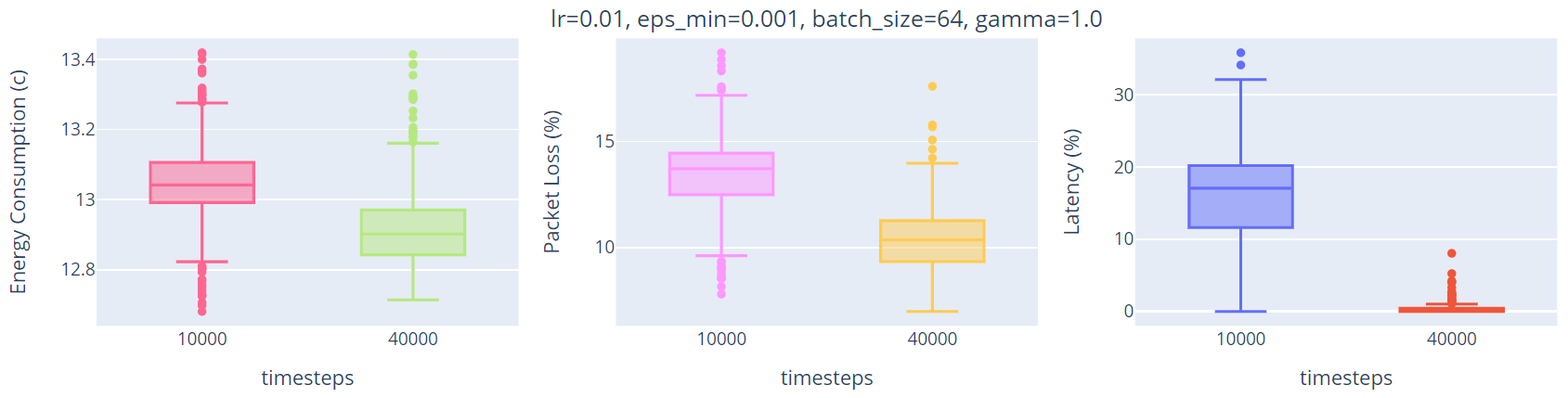
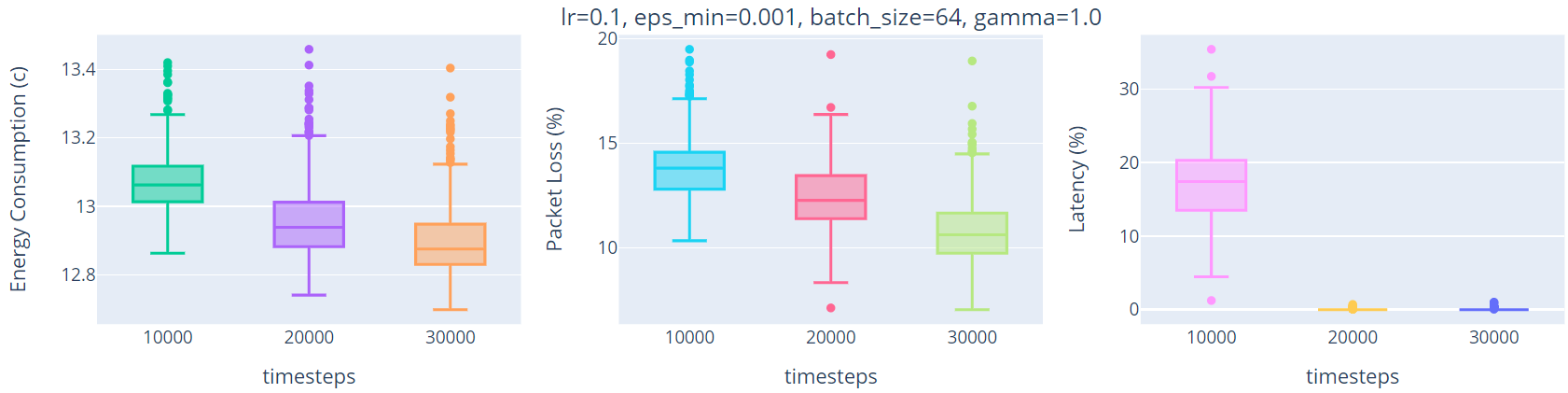
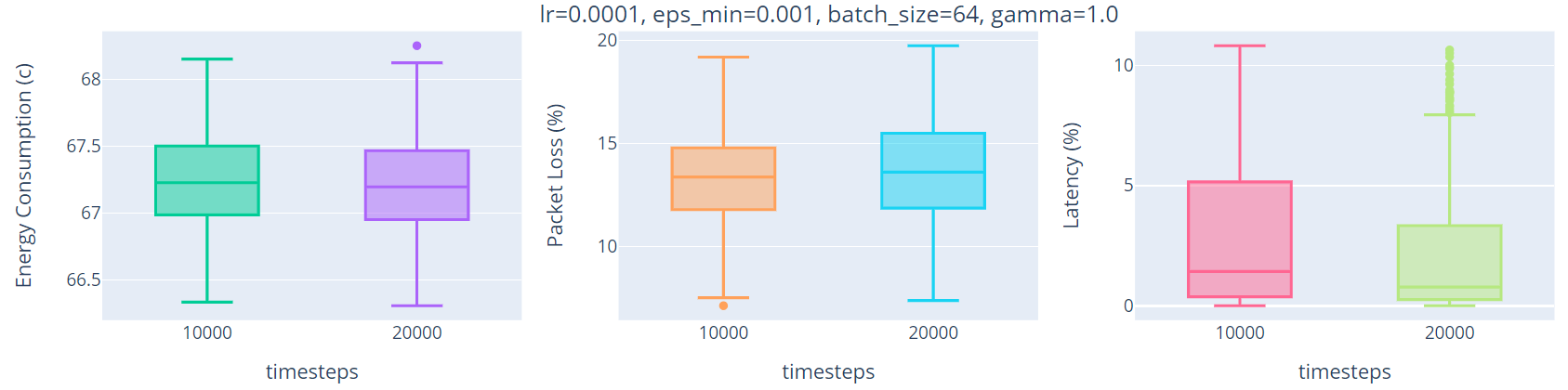
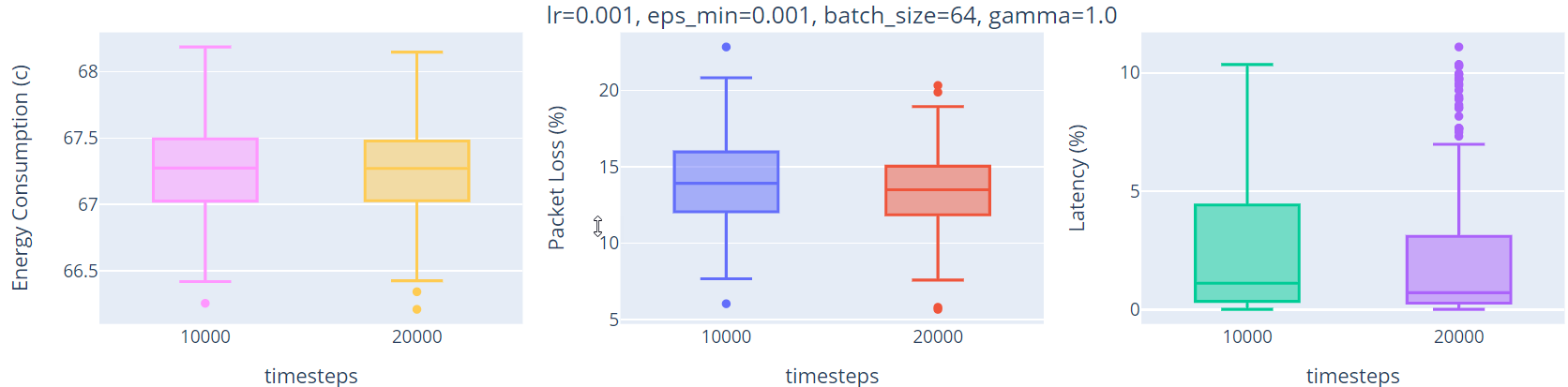
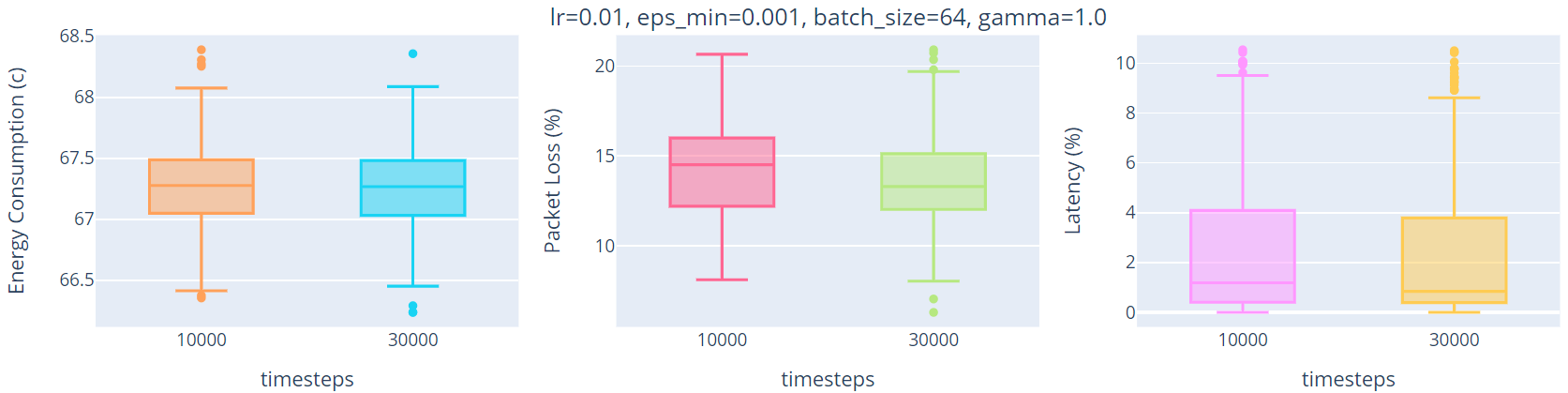
 

Figure 6: Values of different quality properties when experimenting with different LR values for DeltaIoTv1



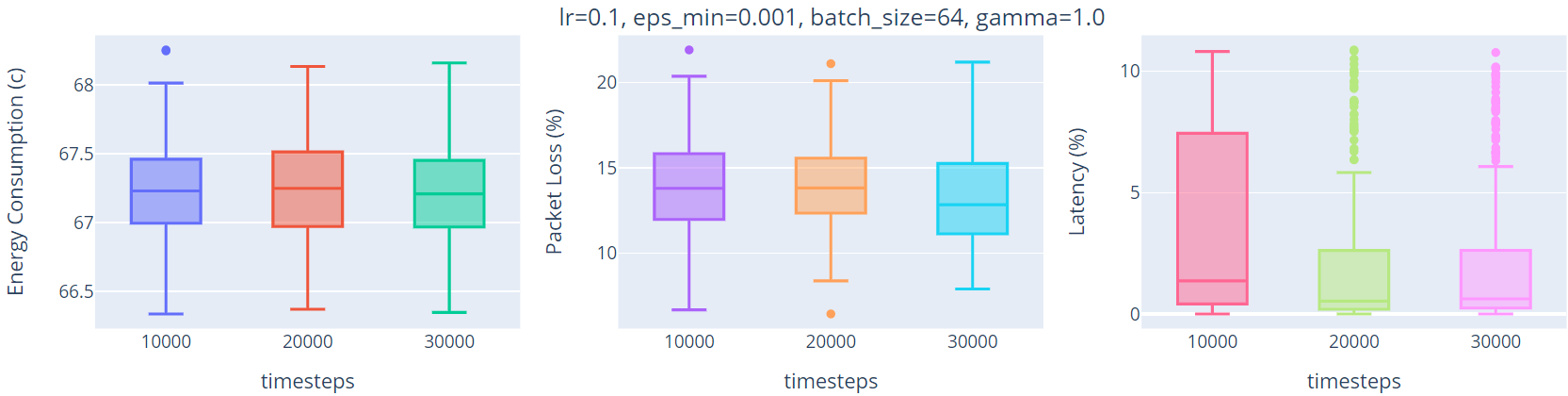


Figure : Values of different quality properties when experimenting with different LR values for DeltaIoTv2

Table 4: Hyper-parameters used when experimenting with DF (gamma)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem** | **Goals** | **Hyper-parameters** | | | | **Median values of different quality properties** | | |
|  | | BS | LR | EDR | DF | EC | PL | LA |
| DeltaIoTv1 | TTS | 64 | 1e-4 | 1e-3 | 0.99 | 12.85 | 10.98 | 0.33 |
| 0.98 | 12.87 | 7.85 | 0.44 |
| 0.95 | 12.90 | 10.35 | 0.34 |
| 0.90 | 12.87 | 10.62 | 0.12 |
| DeltaIoTv2 | 0.99 | 67.22 | 12.66 | 1.59 |
| 0.98 | 67.16 | 11.54 | 2.05 |
| 0.95 | 67.19 | 11.94 | 0.73 |
| 0.90 | 67.18 | 10.60 | 2.37 |

Table : Hyper-parameters used when experimenting with EDR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem** | **Goals** | **Hyper-parameters** | | | | **Median values of different quality properties** | | |
|  | | BS | LR | DF | BS | EC | PL | LA |
| DeltaIoTv1 | TTS | 64 | 1e-4 | 1 | 1e-4 | 12.88 | 7.88 | 0.6 |
| 1e-3 | 12.86 | 13.00 | 0.90 |
| 1e-2 | 12.86 | 7.98 | 0.41 |
| 1e-1 | 12.88 | 12.55 | 0.27 |
| DeltaIoTv2 | 1e-4 | 67.11 | 12.66 | 4.21 |
| 1e-3 | 67.20 | 13.36 | 0.56 |
| 1e-2 | 67.18 | 15.76 | 7.30 |
| 1e-1 | 67.19 | 13.55 | 2.34 |

Table : Hyper-parameters used when experimenting with BS

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem** | **Goals** | **Hyper-parameters** | | | | **Median values of different quality properties** | | |
|  | | EDR | LR | DF | BS | EC | PL | LA |
| DeltaIoTv1 | TTS | 1e-3 | 1e-4 | 1 | 64 | 12.88 | 7.88 | 0.6 |
| 128 | 12.86 | 13.00 | 0.90 |
| 256 | 12.86 | 7.98 | 0.41 |
| 512 | 12.88 | 12.55 | 0.27 |
| DeltaIoTv2 | 64 | 67.11 | 12.66 | 4.21 |
| 128 | 67.13 | 11.30 | 7.54 |
| 256 | 67.33 | 13.51 | 0.66 |
| 512 | 67.15 | 11.28 | 0.36 |

We measured the learning performance for 4 combinations of the aforementioned hyper-parameters for each of the adaptation goals and each of the instances of DeltaIoT. Table 6 shows the hyper-parameters chosen. It can be observed through our experiments that the choice of best hyper-parameters is really dependent on which quality property is more important by the stakeholders. Since in DelatIoT we have different objectives that need to be satisfied at the same time, we could select any of the hyper-parameters that satisfies all quality properties.

Table : Best hyper-parameter values for DRL4SAO on TTS and TTO goals.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Case study** | **Goals** | **Hyper-parameters** | | | |
|  | | LR | DF | EDR | BS |
| DeltaIoTv1 | TTS | 1e-4 | 1 | 1e-3 | 64 |
| DeltaIoTv2 | TTS | 1e-4 | 0.98 | 1e-3 | 64 |
| DeltaIoTv1 | TTO | 1e-4 | 1 | 1e-3 | 64 |
| DeltaIoTv2 | TTO | 1e-3 | 1 | 1e-3 | 64 |

* 1. Comparison with other methods

We compare DRL4SAO with three different methods. 1) we used a Reference method that analyzes the whole adaptation space without using machine learning. 2) we used a competing method DLASeR+ [27] that applies a deep neural network to reduce adaptation spaces. 3) Finally, as a sanity check, we used a method that selects an adaptation option randomly.

Effectiveness - Effect on realization of adaptation goals. To evaluate the effectiveness, we compare the median values of the quality properties that correspond to the adaptation goals over 300 learning cycles. We consider the goals of the system satisfied if the median of the values over 300 cycles meet the goal. It should be noted that in some cycles there is no adaptation option that satisfies the goal.

For the setting with TTS goals (threshold, threshold, set-point), Figure 8 and Figure 9 show the results for DeltaIoTv1 and DeltaIoTv2 respectively. For DeltaIoTv1, the results for packet loss and latency are better compared to other methods. We observe that DRL4SAO always satisfies the goals (i.e., all median values are below the thresholds). For energy consumption (set-point goal), the results for DRL4SAO are slightly higher compared to DLASeR+ [27] and Reference methods, i.e., an increase of 0.08 C (a utility penalty of 0.58%) compared to DLASeR+ [27] and an increase of 0.08 C (a utility penalty of 0.58%) compared to the Reference. For DeltaIoTv2, the results for all three quality properties are better compared to other methods.

For the setting with TTO goals (threshold, threshold, optimization), Figure 10 and Figure 11 show the results for DeltaIoTv1 and DeltaIoTv2 respectively. In case of DeltaIoTv1, for the optimization goal the median values of DRL4SAO are slightly higher compared to the other two methods. We observe an increase of 0.16 C (a utility penalty of 1.24%) compared to DLASeR+ [27] and an increase of 0.21 C (a utility penalty of 1.63%) compared to the Reference.

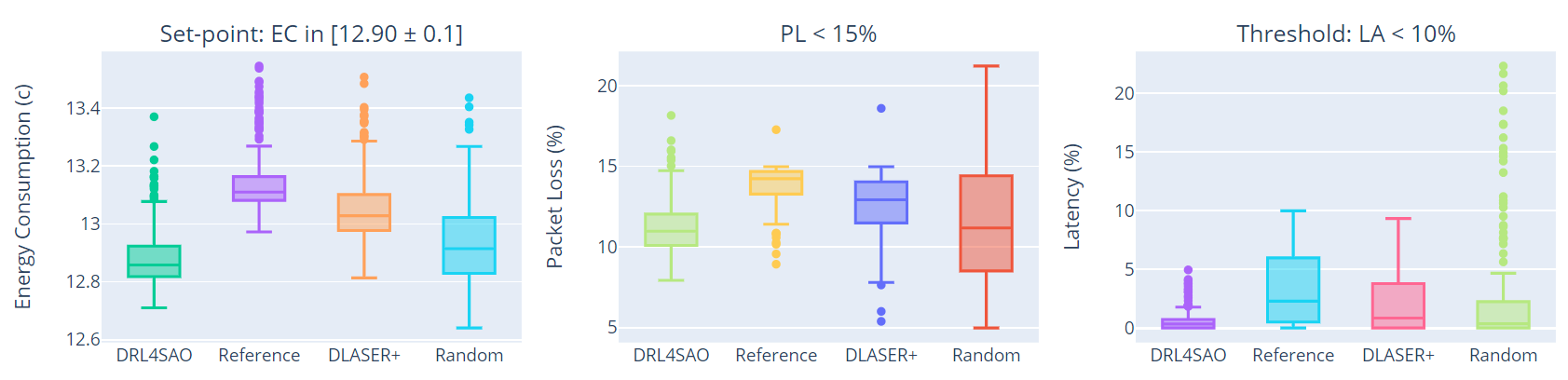


Figure 8: Effect on the realization of the adaptation goals for the TTS setting (DeltaIoTv1).

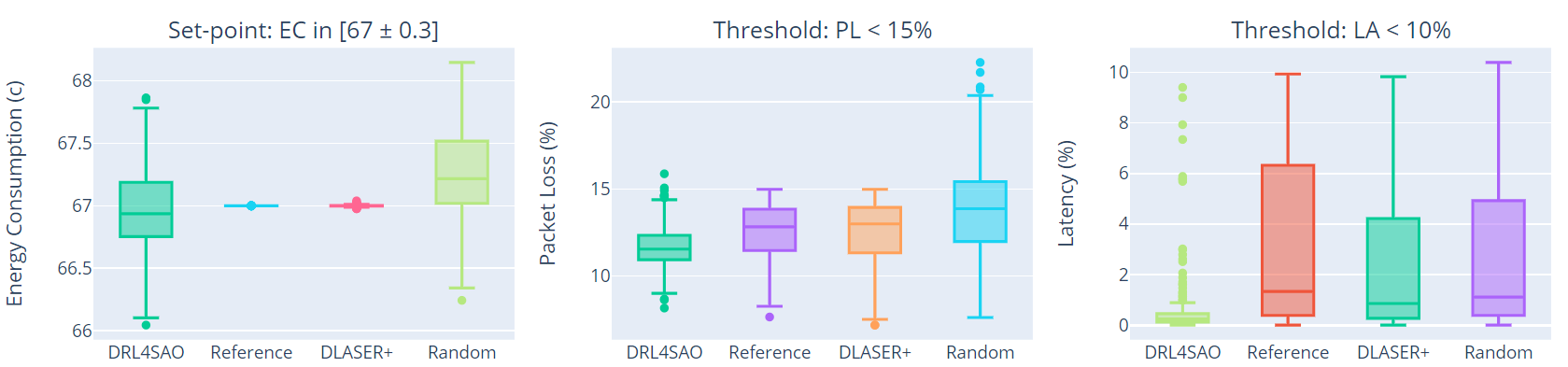


Figure 9: Effect on the realization of the adaptation goals for the TTS setting (DeltaIoTv2).

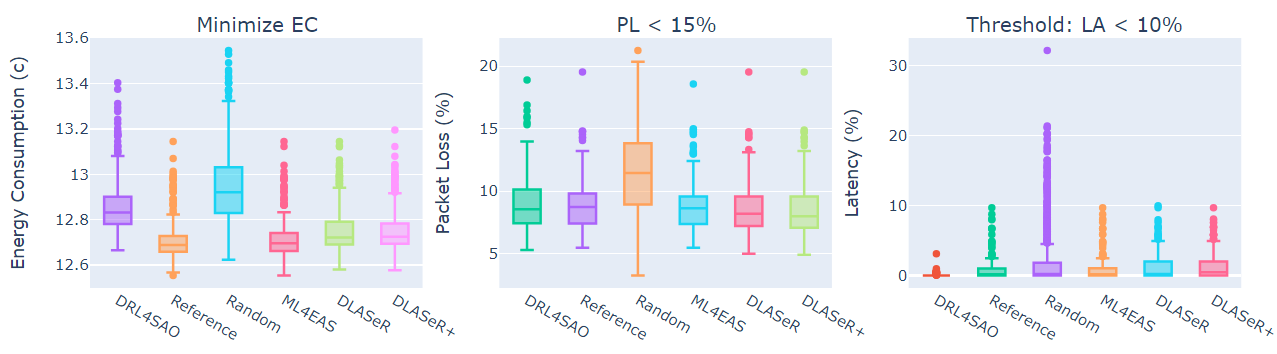


Figure 10: Effect on the realization of the adaptation goals for the TTO setting (DeltaIoTv1).

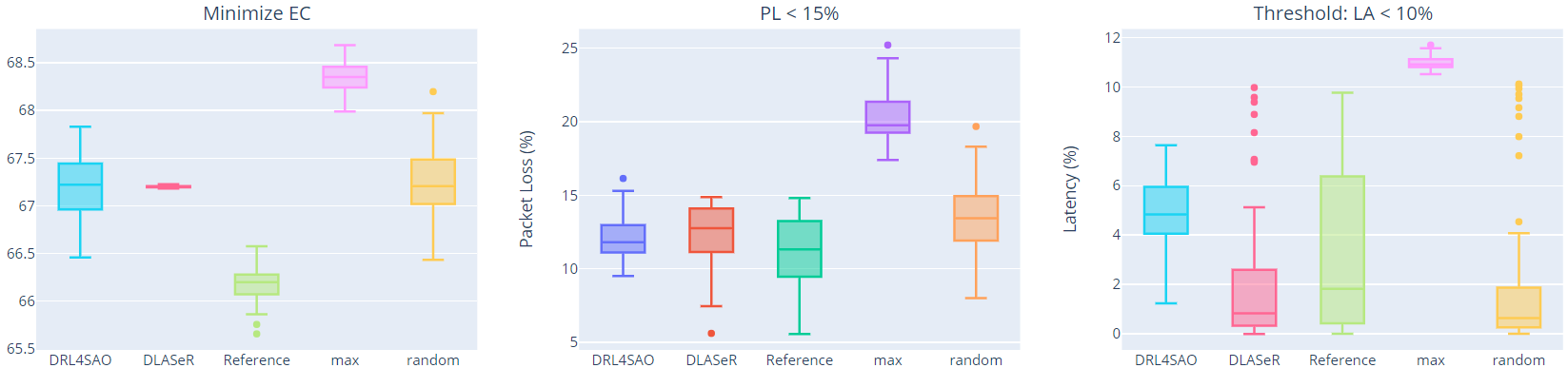


Figure 11: Effect on the realization of the adaptation goals for the TTO setting (DeltaIoTv2).

Efficiency - Table 7 presents the results for the Average adaptation space reduction (AASR), learning time overhead (LTO), and overall time saved (OTS) of DRL4SAO and DLASeR+ [27] compared to the reference method. Results show that the AASR and OTS obtained by our method is higher compared to DLASeR [27]. The reason for this is that in our method a deep reinforcement learning agent is deployed where it selects the best option based on its perception, whereas in DLASeR [27] a deep learning method is used where it classifies the adaptation options and selects a subset of suitable ones.

Table : Adaptation space reductions on two adaptation goals of Table 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Problem** | **goal** | **method** | **AASR** | **LTO** | **OTS** |
| DeltaIoTv1 | TTS | DRL4SAO | 99.5% | 0.004% | 98.34% |
| DLASeR+ [27] | 51.2% | 0.005% | 94.73% |
| TTO | DRL4SAO | 99.5% | 0.007% | 98.12% |
| DLASeR+ [27] | 84.55% | 0.008% | 90.82% |
| DeltaIoTv2 | TTS | DRL4SAO | 99.5% | 0.006% | 96.65% |
| DLASeR+ [27] | 54.84% | 0.009% | 90.23% |
| TTO | DRL4SAO | 99.5% | 0.008% | 97.11% |
| DLASeR+ [27] | 57.76% | 0.009% | 91.02% |

* 1. Analysis and discussion

The main objective of this paper is to select an adaptation option that satisfies the goals of the system, considering that the adaptation space is large and verifying all adaptation options is infeasible. That is because the time to adapt the system is limited and in this limited time the system must find a feasible solution. We compared our proposed method with a number of approaches that have tried to reduce the adaptation space with the same objective as stated by our method, namely DLASER, DLASER+, and ML4EAS. DLASER and DLASER+ both use a method that uses trained deep neural network to predict the quality properties of the system, then select a subset of those for analysis, and then choose the best one among them. As for ML4EAS, it employs the same technique, but the authors use a method based on traditional machine learning based methods such as decision tree. The problem with all of these competing methods is that they need labelled training data provided offline in order to learn a good model, which is difficult to obtain due to the dynamic and uncertain nature of environment in which these systems operate.

In DRL4SAO, as we employ a method based on DRL, there is no need for online training data compared to the aforementioned competing methods. The system can use online data to train itself and instead of choosing a subset of adaptation options and performing analysis to choose the best one, which adds an extra layer of overhead, it selects one adaptation option and self-adaptive system is trained accordingly.

1. Threats to validity

Although our evaluation results show that DRL4SAO is effective and efficient in reducing the adaptation space and finding an appropriate adaptation option, it is still subject to a number of validity threats.

External validity. Since the evaluation was done only in one domain, the conclusions cannot generalized and further evaluation is needed on different domains with different adaptation spaces. Furthermore, in our experiments we only considered threshold and set-point goals and one optimization, so our method may not be directly applicable for settings with optimization goals.

Internal validity. We measured the impact on the quality properties of two concrete instances of DeltaIoT to assess DRL4SAL’s capability to select an appropriate adaptation option and improve the decision-making. The characteristics of these applications, such as the network structure, uncertainty forms, and the specific aims that we considered, may influence how difficult it is to demonstrate the benefits of our method. We mitigated this threat to some extent by taking into account real-world application settings available in the DeltaIoT package.

1. conclusions

In the face of uncertainty, ensuring the continuous satisfaction of quality properties in self-adaptive software systems is a challenging task. To tackle this challenge, we studied the research question “Can deep reinforcement learning (DRL) be used to determine an effective adaptation option from a large adaptation space and improve the decision-making process, allowing a self-adaptive system to conduct a more efficient analysis without compromising adaptation goals?”. To answer this question, we presented DRL4SAO. DRL4SAO relies on a DRL-based learning module which sits on top of the mape-k feedback loop and learns what the appropriate adaptation option is based on received feedback from the environment. We evaluated our proposed method on two instances of DeltaIoT with varying adaptation space sizes. The evaluation results show that DRL4SAO can effectively and efficiently select an appropriate adaptation option.

We are currently investigating the effect of sudden, abrupt changes of the distribution of data received from the environment and the effect that this can have on the effectiveness of our proposed method. We also plan to look into support for dynamically adding and removing adaptation options.

REFERENCES

1. Weyns, D., *An introduction to self-adaptive systems: A contemporary software engineering perspective*. 2020: John Wiley & Sons.

2. Krupitzer, C., et al., *A survey on engineering approaches for self-adaptive systems.* Pervasive and Mobile Computing, 2015. **17**: p. 184-206.

3. Weyns, D., *Software engineering of self-adaptive systems: an organised tour and future challenges.* Chapter in Handbook of Software Engineering, 2017: p. 2.

4. Edwards, G., et al. *Architecture-driven self-adaptation and self-management in robotics systems*. in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2009. IEEE.

5. Weyns, D., et al. *Applying architecture-based adaptation to automate the management of internet-of-things*. in *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*. 2018. Springer.

6. Muccini, H., M. Sharaf, and D. Weyns. *Self-adaptation for cyber-physical systems: a systematic literature review*. in *Proceedings of the 11th international symposium on software engineering for adaptive and self-managing systems*. 2016.

7. Jamshidi, P., et al. *Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures*. in *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 2016. IEEE.

8. Castañeda, L., N.M. Villegas, and H.A. Müller. *Self-adaptive applications: On the development of personalized web-tasking systems*. in *Proceedings of the 9th international symposium on software engineering for adaptive and self-managing systems*. 2014.

9. Weyns, D., et al. *Perpetual assurances for self-adaptive systems*. in *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. 2017. Springer.

10. De Lemos, R., et al. *Software engineering for self-adaptive systems: Research challenges in the provision of assurances*. in *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. 2017. Springer.

11. Calinescu, R., et al., *Dynamic QoS management and optimization in service-based systems.* IEEE Transactions on Software Engineering, 2010. **37**(3): p. 387-409.

12. Moreno, G.A., et al. *Proactive self-adaptation under uncertainty: a probabilistic model checking approach*. in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015.

13. Vivek, N., et al., *Finding faster configurations using FLASH.* IEEE Transactions on Software Engineering, 2018.

14. Metzger, A., et al., *Feature-model-guided online learning for self-adaptive systems, vol. 12571.* 2020.

15. Elkhodary, A., N. Esfahani, and S. Malek. *FUSION: a framework for engineering self-tuning self-adaptive software systems*. in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 2010.

16. Jamshidi, P., et al. *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)*. 2019. IEEE.

17. Quin, F., et al. *Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning*. in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2019. IEEE.

18. Weyns, D., et al., *Deep Learning for Effective and Efficient Reduction of Large Adaptation Spaces in Self-Adaptive Systems.* arXiv preprint arXiv:2204.06254, 2022.

19. Quin, F., D. Weyns, and O. Gheibi, *Reducing large adaptation spaces in self-adaptive systems using machine learning.* Journal of Systems and Software, 2022: p. 111341.

20. Iftikhar, M.U., et al. *Deltaiot: A self-adaptive internet of things exemplar*. in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017. IEEE.

21. Kephart, J.O. and D.M. Chess, *The vision of autonomic computing.* Computer, 2003. **36**(1): p. 41-50.

22. Mnih, V., et al., *Human-level control through deep reinforcement learning.* nature, 2015. **518**(7540): p. 529-533.

23. Montague, P.R., *Reinforcement learning: an introduction, by Sutton, RS and Barto, AG.* Trends in cognitive sciences, 1999. **3**(9): p. 360.

24. David, A., et al., *Uppaal SMC tutorial.* International journal on software tools for technology transfer, 2015. **17**: p. 397-415.

25. Weyns, D. and U.M. Iftikhar, *ActivFORMS: A formally founded model-based approach to engineer self-adaptive systems.* ACM Transactions on Software Engineering and Methodology, 2023. **32**(1): p. 1-48.

26. Kachi, F. and C. Bouanaka, *A hybrid model for efficient decision-making in self-adaptive systems.* Information and Software Technology, 2023. **153**: p. 107063.

27. Van Der Donckt, J., et al. *Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals*. in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2020.

28. Kwiatkowska, M., G. Norman, and D. Parker. *PRISM: Probabilistic symbolic model checker*. in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. 2002. Springer.

29. Coker, Z., D. Garlan, and C.L. Goues. *SASS: Self-Adaptation Using Stochastic Search*. in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*. 2015. Institute of Electrical and Electronics Engineers Inc.

30. Kinneer, C., et al. *Managing uncertainty in self-adaptive systems with plan reuse and stochastic search*. in *ACM/IEEE 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2018, , co-located with International Conference on Software Engineering, ICSE 2018*. 2018. IEEE Computer Society.

31. Kinneer, C., et al. *Building reusable repertoires for stochastic self-\* planners*. in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2020. IEEE.

32. Kinneer, C., D. Garlan, and C.L. Goues, *Information reuse and stochastic search: Managing uncertainty in self-\* systems.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2021. **15**(1): p. 1-36.

33. Chen, T., et al., *FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software.* ACM Transactions on Software Engineering and Methodology, 2018. **27**(2).

34. Pascual, G.G., M. Pinto, and L. Fuentes. *Run-time adaptation of mobile applications using genetic algorithms*. in *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2013. IEEE.

35. Kim, D. and S. Park. *Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software*. in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2009. IEEE.

36. Cámara, J., H. Muccini, and K. Vaidhyanathan. *Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive iot systems*. in *2020 IEEE International Conference on Software Architecture (ICSA)*. 2020. IEEE.

37. Metzger, A., et al., *Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration.* Computing, 2022: p. 1-22.

38. Caporuscio, M., et al. *Reinforcement learning techniques for decentralized self-adaptive service assembly*. in *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*. 2016. Springer.

39. Barrett, E., E. Howley, and J. Duggan, *Applying reinforcement learning towards automating resource allocation and application scalability in the cloud.* Concurrency and computation: practice and experience, 2013. **25**(12): p. 1656-1674.

40. Arabnejad, H., et al. *A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling*. in *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*. 2017. IEEE.

41. Moustafa, A. and M. Zhang. *Learning efficient compositions for QoS-aware service provisioning*. in *2014 IEEE International Conference on Web Services*. 2014. IEEE.

42. Zhao, T., et al. *A reinforcement learning-based framework for the generation and evolution of adaptation rules*. in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 2017. IEEE.

43. Shaw, R., E. Howley, and E. Barrett, *Applying reinforcement learning towards automating energy efficient virtual machine consolidation in cloud data centers.* Information Systems, 2022. **107**: p. 101722.

44. Iftikhar, M.U. and D. Weyns. *Activforms: Active formal models for self-adaptation*. in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2014.

1. Email Address: k.alireza@modares.ac.ir, sjalili@modares.ac.ir

   [↑](#footnote-ref-1)
2. VersaSense website: www.versasense.com [↑](#footnote-ref-2)
3. Sensors used in DeltaIoT include RFID sensors which are used to provide access control to labs, passive infrared sensors monitor the occupancy of several buildings, and heat sensors are employed to sense the temperature. [↑](#footnote-ref-3)
4. ­­­ A networked tiny embedded computer. [↑](#footnote-ref-4)
5. The communication in DeltaIoT is time-synchronized and organized in cycles with a fixed number of slots. Neighboring motes are assigned such slots during which they can exchange packets. Motes collect data (locally generated or received from other motes) in a buffer. When a mote gets a turn to communicate with another mote, it forwards the packets to the other mote. Packets that cannot be sent remain in the buffer until the mote is assigned a next slot. [↑](#footnote-ref-5)