Online Reinforcement Learning for Self-adaptive Information Systems

Alexander Palm(✉), Andreas Metzger, and Klaus Pohl

paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany
{alexander.palm,andreas.metzger,klaus.pohl}@paluno.uni-due.de

Abstract. A self-adaptive information system is capable of maintaining its quality requirements in the presence of dynamic environment changes. To develop a self-adaptive information system, information system engineers have to create self-adaptation logic that encodes when and how the system should adapt itself. However, developing self-adaptation logic may be difficult due to design time uncertainty; e.g., anticipating all potential environment changes at design time is in most cases infeasible. Online reinforcement learning (RL) addresses design time uncertainty by learning the effectiveness of adaptation actions through interactions with the system's environment at run time, thereby automating the development of self-adaptation logic. Existing online RL approaches for self-adaptive information systems exhibit two shortcomings that limit the degree of automation: they require manually fine-tuning the exploration rate and may require manually quantizing environment states to foster scalability. We introduce an approach to automate the aforementioned manual activities by employing policy-based RL as a fundamentally different type of RL. We demonstrate the feasibility and applicability of our approach using two self-adaptive information system exemplars.

Keywords: Self-adaptation · Reinforcement learning · Information system engineering

1 Introduction

The concept of *self-adaptation* facilitates developing information systems that are capable of maintaining their quality requirements even if the systems' environment changes dynamically [3,25]. Self-adaptation thereby helps developing systems that can operate in a resilient way at run time. To this end, a self-adaptive information system can modify its own structure, parameters and behavior at run time based on its perception of the environment, of itself and of its requirements. An example is a self-adaptive online store that must maintain its performance requirements under changing workloads. Faced with a sudden increase in workload, the online store may adapt itself by deactivating optional system features to use less resources; e.g., it may deactivate its resource-intensive

S. Dustdar et al. (Eds.): CAiSE 2020, LNCS 12127, pp. 169–184, 2020.
https://doi.org/10.1007/978-3-030-49435-3_11

recommendation engine [17]. Another example is a predictive business process monitoring system that generates alarms for triggering proactive process adaptations [12,21]. If a violation of a process performance objective is predicted, the system may trigger an adaptation of the running process to use more or different resources to speed up the execution of the remaining process steps.

To develop a self-adaptive information system, information system engineers have to develop *self-adaptation logic* that encodes when and how the system should adapt itself. Information system engineers, for instance, may specify event-condition-action rules that define which adaptation action is executed in response to a given environment change. Developing self-adaptation logic requires an intricate understanding of the information system and its environment, and how adaptations impact on system quality [7,8]. Among other concerns, it requires anticipating the potential environment changes the system may encounter at run time to define how the system should adapt itself in response to these environment changes. However, anticipating all potential environment changes at design time is in most cases infeasible due to *design time uncertainty* [8,24]. In addition, while the principle effects of an adaptation on the system may be known, accurately anticipating the effect of a concrete adaptation is difficult; e.g., due to simplifying assumptions made during design time [8,15].

One emerging way to address design time uncertainty is to employ *online* reinforcement learning (RL) [1,2,4,10,19,22,31,32,35]. RL can learn the effectiveness of adaptation actions through interactions with the system's environment. This means that instead of information system engineers having to manually develop the self-adaptation logic, the system automatically learns the self-adaptation logic via machine learning at run time. The information system engineer expresses the learning problem in a declarative fashion, in terms of the learning goals the system should achieve. In the online store example, they may express maintaining system performance as a learning goal. Online RL thereby automates the manual engineering task of developing the self-adaptation logic.

Existing online RL approaches for self-adaptive information systems exhibit two shortcomings that limit the degree of automation that may be achieved. First, to facilitate convergence of learning, information system engineers have to manually fine-tune the rate of exploration versus exploitation, i.e., how often adaptation actions are selected that were not selected before. Second, most existing approaches use a lookup table to represent the learned knowledge, which requires information system engineers to manually quantize environment states to facilitate scalability if the environment has a high number of states. These two manual activities may be expensive and potentially unreliable [15] and may require information not available at design time due to design time uncertainty.

Our main idea is to automate the aforementioned manual activities by employing *policy-based RL* as a fundamentally different type of RL [23,29]. In simple terms, policy-based RL represents the learned knowledge as a an artificial neural network [29]. Our approach conceptually, formally and technically integrates policy-based RL into a well-known self-adaptive system reference model. Our approach thereby facilitates online RL for self-adaptive information systems

without having to (1) manually quantize environment states and (2) manually fine-tune the exploration rate. We demonstrate the feasibility and applicability of our policy-based approach using two information system exemplars: a self-adaptive web application and a predictive process monitoring system.

In the remainder of the paper, we provide a more detailed problem description in Sect. 2, introduce our approach in Sect. 3, present our experimental evaluation in Sect. 4, discuss limitations in Sect. 5, and analyze related work in Sect. 6.

2 Problem Statement

As motivated above, RL helps to effectively automate the engineering of an information system's self-adaptation logic. In general, RL learns the effectiveness of an agent's actions through the agent's interactions with its environment [28]. At time step t the agent executes an action a_t in environment state s_t. As a result, the environment transitions to s_{t+1} at time step $t+1$ and the agent receives a reward r_{t+1} for executing the action. The goal of RL is to optimize cumulative rewards. When RL is used for self-adaptive information systems, "action" means the concrete adaptation action (such modifying the structure, parameters or behavior of the system), "agent" takes the role of the self-adaptation logic, and "environment" includes the information system to be adapted at run time.

Existing approaches that use RL for building self-adaptive information systems utilize some variant of *value-based* RL (see Sect. 6). Value-based RL is a model-free RL technique that employs a so called value function for representing the learned knowledge. The value function gives the expected cumulative reward when performing a particular action in a given state [28]. A concrete action may be selected by choosing the action that has the highest value in a given state. Typical variants of value-based RL are Q-Learning and SARSA [28], which differ in how they update the value function. Existing RL approaches for self-adaptive information systems use two different ways to represent the value function:

Value Function as Lookup Table. Most existing approaches store the value function in a lookup table (see Sect. 6). Even though such tabular solution techniques are straightforward to implement and well understood [31], they exhibit two key shortcomings. First, due to the discrete nature of the lookup table, tabular solution techniques are limited to discrete state and action spaces and cannot cope with continuous state and action spaces. Specifically, this means that environment states have to be enumerable and cannot be represented by continuous (e.g., real-valued) variables [22,31]. Second, the size of the lookup table directly depends on the number of environment states that have to be stored, and thus the size increases exponentially with the number of state variables. As a result, tabular solution techniques suffer from poor scalability, because the learning process has to collect data for all entries of the table to learn effectively [22,31].

A common way to address these limitations is to quantize continuous environment states by defining a sufficiently small number of discrete environment states [28]. Such quantization is a manual activity performed by information system engineers, and thus may be expensive and potentially unreliable. The

environment states may be quantized too coarse-grained to reflect reality, or they may be quantized too fine-grained, which may lead to poor scalability due to a too large size of the lookup table. Also, the extent of the state space (i.e., set of all states) may be unknown due to design time uncertainty, and thus defining lower and upper bounds of discrete states may not be possible. To know the extent of the state space would mean that developers at design time can anticipate all potential environment changes the system may encounter at run time.

Value Function Approximation. An alternative approach to avoid quantization of the state space is to approximate the value function; e.g., using linear or non-linear techniques (such as artificial neural networks). This allows coping with large state spaces by generalizing over unseen states [28]. Despite such function approximation, value-based RL in general faces the exploration-exploitation dilemma [28]. To optimize rewards, actions should be selected that have shown to be effective (aka. exploitation). However, to discover such actions in the first place, actions that were not selected before should be selected (aka. exploration). One typical solution to the exploration-exploitation dilemma is the ϵ-greedy mechanism. During learning, the ϵ-greedy mechanism randomly chooses an action with probability ϵ. The challenge for an information system engineer is to fine-tune the balance between exploitation and exploration in order to ensure convergence of the learning process [28]. As an example, the information system engineer may implement a mechanism that decreases ϵ over time, thereby reducing the amount of exploration in order to facilitate convergence. However, for online learning this poses the challenge of when and how to increase ϵ again in order to capture non-stationary environments, i.e., environments in which the results of adaptation actions change over time [28].

Summarizing, the degree of automation of existing approaches is limited. Information system engineers have to perform manual activities that may require effort or may be difficult to perform due to design time uncertainty.

3 Policy-Based Online Reinforcement Learning Approach

Self-adaptive System Reference Model as Basis. Our approach enhances the MAPE-K model, a well-known reference model for self-adaptive systems [14,16,18]. As shown in Fig. 1 this model suggests conceptually structuring a self-adaptive system into two main elements: *system logic* and *self-adaptation logic*. The self-adaptation logic is further structured into four main conceptual activities that leverage a common *knowledge* base. The knowledge base includes information about the managed system and its environment (e.g., encoded

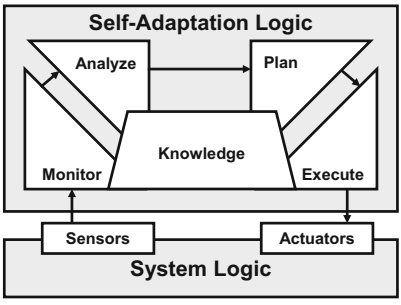

Fig. 1. MAPE-K reference model [16]

in the form of models at run time), as well as adaptation goals (requirements) and adaptation strategies or rules. The four activities are concerned with *monitoring* the system logic and the system's environment via *sensors*, *analyzing* the monitoring data to determine the need for an adaptation, *planning* adaptation actions, and *executing* these adaptation actions via *actuators*, thereby modifying the system logic at run time.

Policy-Based Reinforcement Learning Foundations. The fundamental idea behind policy-based RL is to directly use and optimize a parametrized stochastic *action selection policy* [23,29]. The action selection policy maps states to a probability distribution over the action space (i.e., set of possible actions). This means that actions are selected by sampling from this probability distribution. A *learning cycle* consists of a predefined number of n time steps. At the end of each learning cycle, the trajectory of n actions, states and rewards are used for a policy update. During a policy update, the policy parameters are perturbed based on the rewards received, such that the resulting probability distribution is shifted towards a direction which increases the likelihood of selecting actions which led to a higher cumulative reward.

Conceptual Overview of Approach. Figure 2 depicts the conceptual architecture of our approach, showing how the elements of policy-based RL are integrated into the MAPE-K loop. The dark-gray area indicates where the *action selection* of RL takes the place of the *analyze* and *plan* activities of MAPE-K. The learned *stochastic policy* takes the role of the self-adaptive system's *knowledge* base.

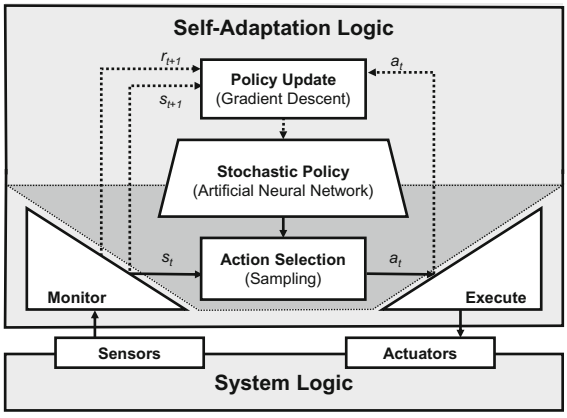

Fig. 2. Conceptual architecture of policy-based approach

At run time the policy is used by the self-adaptation logic to select (via sampling) an adaptation action a_t based on the current state s_t determined by the *monitoring* activity. Action selection determines whether there is a need for

an adaptation (given the current state) and plans (i.e., selects) the respective adaptation action to *execute*. A *policy update* utilizes the trajectory of actions a_t, states s_{t+1}, and rewards r_{t+1} to update the policy. In our approach, policy updates are performed via so-called policy gradient methods [28,29], because the policy is represented as an artificial neural network. In our architecture, rewards are computed by the monitoring activity, as this activity has access to all sensor information collected from the system and its environment.

Formalization of Approach. As mentioned above, the learning problem is stated in a declarative fashion. Typically, it can be formalized as a Markov decision process $MDP = (S, A, T, R)$, with

- S being the state space composed of a set of environment and system states $s \in S$ observable by monitoring via the system logic's sensors (e.g., system workload and performance of the system),
- A being the action space with a set of possible adaptation actions $a \in A$, i.e., possible ways the system may be adapted using the system logic's actuators (e.g., turning off or on different system features),
- $T : S \times A \times S \rightarrow [0, 1]$ being the transition probability among states with $T(s_t, a_t, s_{t+1}) = \Pr(s_{t+1}|s_t, a_t)$, which gives the probability that adaptation action a_t in state s_t will lead to a state s_{t+1}, and
- $R : S \rightarrow \mathbb{R}$, being a reward function which specifies the numerical reward the system receives in state s_t. The reward function expresses the learning goal to achieve, which in our case expresses maintaining the quality requirements of the system (e.g., performance should not fall below a given threshold).

Policy-based reinforcement learning finds a solution to the MDP in the form of a parametrized stochastic policy $\pi_\theta : S \times A \rightarrow [0, 1]$, giving the probability of taking adaptation action a in state s, i.e., $\pi_\theta(s, a) = \Pr(a|s)$. The policy's parameters (weights of the artificial neural network) are given as a vector $\theta \in \mathbb{R}^d$.

Regarding design time uncertainty, we assume that we know A, S, and R, but do not know T. More precisely, even if we do not know the exact states and thus state space S, we know the state variables. As an example, even if we do not know exact workloads of a web application (and maybe not even the maximum workload), we can express a state variable workload $w \in \mathbb{N}^+$. We assume that we do not know T due to design time uncertainty about how adaptation impacts on system quality. As an example, we may not have an exact understanding of how different configurations of the system perform under different workloads.

Proof-of-Concept Implementation To select a concrete policy-based RL algorithm for the implementation of our approach, we took into account two main considerations. First, as we assume we do not know the transition function T, we need to use a model-free variant of policy-based RL. Second, to facilitate online learning, we need an algorithm that continuously updates the policy without waiting for a final outcome, i.e., without waiting for reaching a terminal state. Actor-critic algorithms are a model-free variant of policy-based RL algorithms that use bootstrapping (i.e., knowledge is updated continuously without waiting for a final outcome). We use proximal policy optimization (PPO [26]) as

a state-of-the-art actor-critic algorithm. PPO is rather robust for what concerns hyper-parameter settings. Thereby, we avoid extensive hyper-parameter tuning compared to other actor-critic algorithms. In addition, PPO avoids too large policy updates by using a so called clipping function. A too large policy update may mean that RL misses the global optimum and remains stuck in a local optimum. To represent the actor and critic models of PPO, we used multi-layer perceptrons with two hidden layers of 64 neurons each (neurons in the input and output layers depended on the respective number of action and state variables).

4 Experimental Evaluation

To demonstrate the feasibility and applicability of our approach, the scope of our experiments is to analyze whether the system is able to learn and improve its self-adaptation logic at run time. We did not perform a comparative analysis with existing value-based approaches at this stage, because such comparison would be beyond the scope of the current paper. Such comparison would require the careful variation and analysis of a range of parameters for the value-based approach, including the setting of different exploration rates, as well as different levels and forms of quantization of the state space. In particular, one has to be careful not to perform unfair comparisons. As an example, the comparison may be strongly influenced by the chosen quantization (see Sect. 2). A too fine-grained quantization may mean the value-based approach exhibits extremely slow convergence. A too coarse-grained quantization may mean the value-based approach will not be able to distinguish between different states and thereby will not be able to optimize cumulative rewards.

4.1 Self-adaptive Web Application

Subject System. We use the auction web application *Brownout-RUBiS* as a subject system [17]. When a user requests a specific item, the application's recommendation engine provides a list of recommended items based on past auctions. Due to the resource needs of the recommendation engine, Brownout-RUBiS has to balance two quality requirements: maximizing the user experience by providing many recommendations, while minimizing the user-perceived latency. Therefore, the recommendation engine can be adapted by setting a so-called dimmer value $\delta \in [0, 1]$, which represents the per-request probability that the recommendation engine is activated. The dimmer value thus impacts on both quality requirements: A high rate of recommendations increases user experience, but at the same time also increases resource needs and thus may increase latency.

MDP of Subject System. We express the learning problem as input for our approach as shown in Table 1. We define the reward function $r_t \in R$ such that learning may find a good balance between low latency and high recommendation rates. We define the reward function such that a greater r_t is better and aim at *maximizing* the cumulative reward. We assume that user satisfaction will decrease for latencies higher than $\lambda_{\max}$ and thus penalize latencies above $\lambda_{\max}$.

Table 1. MDP of self-adaptive web application

State $s_t = (u_t, \alpha_t, \lambda_t)$	$u_t \in \mathbb{N}^+$:	monitored number of user requests
	$\alpha_t \in [0, 1]$:	monitored recommendation ratio
	$\lambda_t \in \mathbb{R}^+$:	monitored latency
Action $a_t \in A$	$A = \delta \in [0, 1]$:	adapt the dimmer value
Reward $r_t = \alpha_t \cdot f(\lambda_t)$	$\alpha_t \in [0, 1]$:	monitored recommendation ratio
	$\lambda_t \in \mathbb{R}^+$:	monitored latency
	$f(\lambda_t)$:	utility function, $f(\lambda_t) = 1$ if $\lambda_t \leq \lambda_{\max}$; $= 0$ if $\lambda_t > 2 \cdot \lambda_{\max}$; $= -\lambda_t / \lambda_{\max} + 2$ else *(= linearly decreasing reward)*

Experimental Setup. We deployed Brownout-RUBiS on a virtual machine with 64 GB RAM running Ubuntu 16.04.5. We used *httpmon* [17] as workload generator deployed on a separate virtual machine to generate different kinds of workloads. We synthesized workloads using different, representative workload patterns from the literature [19]: stable (constant number of requests), off/on (reflecting periodic batch processing), and cyclic (workload increases and decreases in periods). We also replayed an excerpt of a real-world workload trace [20].

We used ca. 4,600 learning cycles for each kind of workload, as this was a sufficiently high number to observe convergence. Each learning cycle used monitoring data from 128 consecutive time steps (the default setting of the PPO algorithm we use in our implementation, see Sect. 3). We set $\lambda_{\max} = 20$ ms as latency threshold, because it is low enough to require a dimmer value below 1.

Experimental Results. Figure 3 shows the results for each kind of workload. The diagrams show the state, action and reward at each learning cycle averaged over the observations of 128 time steps.

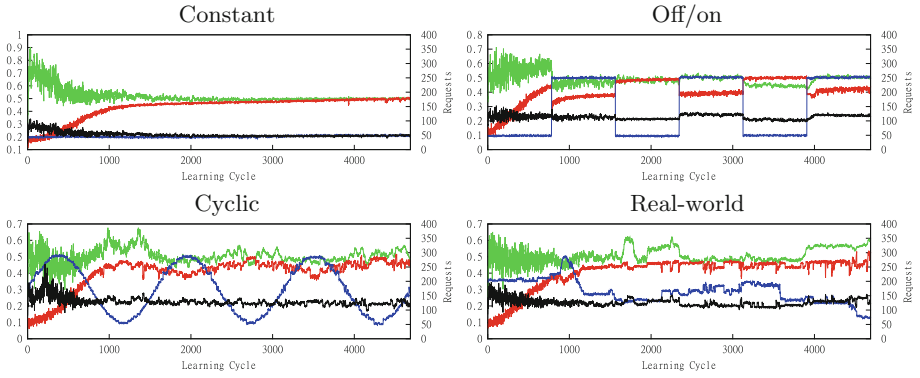

Fig. 3. Learning behavior for self-adaptive web application; **blue** = workload, **black** = latency; **green** = dimmer value; **red** = reward (Color figure online)

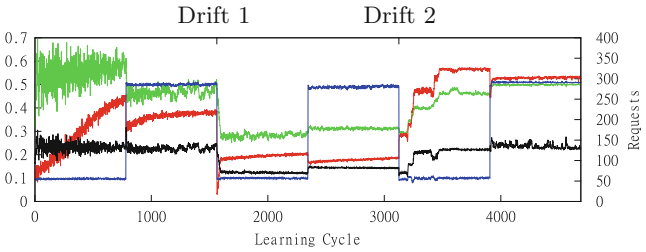

Fig. 4. Learning behavior in non-stationary environment; **blue** = workload, **black** = latency; **green** = dimmer value; **red** = reward (Color figure online)

The results show how our approach enables the system to adapt itself. The system automatically adapts the dimmer value depending on the workload, thereby optimizing the balance between latency and user experience (as visible in the increase of cumulative rewards). While at the beginning of the learning process, the adaptation of the dimmer value shows a high variance for all workload patterns, after some time the variance of adaptation actions becomes visibly lower. For the **constant workload**, the reward converges towards a value of around 0.47 after ca. 1,950 learning cycles and stabilizes at a dimmer value around 0.5, which leads to the highest recommendation ratio without violating the latency threshold. For the **off/on workload**, the reward increases over time for the off as well as the on workload settings. From the second iteration onwards, convergence can be observed. When comparing learning for the off and on periods separately, one can observe that learning is able to reuse knowledge about similar workloads over time. The **cyclic workload**, similar to the off/on workload, indicates how learning may generalize from already acquired knowledge. The overall observations are very much comparable to those of the off/on workload, except that the average reward increases and decreases more slowly, because the workload changes more smoothly. For the **real-world workload**, our approach is able to learn from previously experienced states and is able to keep the reward roughly at the same level by adjusting the dimmer value even though the workload changes over time. Especially if a similar workload reoccurs, our approach is able to quickly determine an effective adaptation action.

Figure 4 shows how our approach automatically captures non-stationary environments. After learning cycle 1562 ("Drift 1"), we reduced the virtual machine compute resources by half. This means that for the same dimmer value the system experiences a higher latency, because less compute resources are available. Our approach learns to decrease the dimmer value such that the latency threshold is not violated. After learning cycle 3125 ("Drift 2"), we increased the resources by 1.5. Again, the dimmer values are set accordingly. Note that our approach is able to capture this non-stationarity without explicitly monitoring the changes in compute resources and without explicitly changing the exploration rate.

4.2 Self-adaptive Process Monitoring System

Subject System. We use a predictive process monitoring system introduced in our earlier work [21]. The system uses neural network ensembles to predict process outcomes. In addition, the system computes so called reliability estimates, which give the probability that predictions are accurate. Only when the reliability is above a given threshold, the system issues an alarm in order to trigger a proactive process adaptation. Thereby the system aims to find a good trade-off between prediction accuracy and prediction earliness (later predictions are more accurate, but leave less time for adaptations). Experimental results show that lower thresholds may lead to higher savings in process costs, whilst posing the risk that such savings may not be achieved in all situations. Higher thresholds may capture more situations but can lead to lower savings. However, a good threshold is not known a priori. One solution is to determine the threshold using a sub-set of the training data [30]. Here, we use online RL as an alternative to learn when to trigger an alarm based on the predictions and their reliability.

MDP of Subject System. We express the learning problem as shown in Table 2. For defining $r_t \in R$, we penalize the system for late predictions (thereby incentivizing earliness) as well as for high process execution costs (following the cost model from [30]). Like above, we aim at *maximizing* the cumulative reward, and thus express the reward function such that a greater r_t is better.

Table 2. MDP of self-adaptive process monitoring system

State $s_t = (\pi_t, \rho_t)$	$\pi_t \in [0, 1]$:	relative prefix-length (*smaller values mean earlier in the process*)
	$\hat{o}_t \in \mathbb{R}$:	predicted numeric process outcome
	$\rho_t \in [0, 1]$:	prediction reliability
Action $a_t \in A$	$A = \{1, 0\}$:	1 = trigger alarm; 0 = do not trigger alarm
Reward $r_t = -(e_t + c_t)$	$e_t = 1.5$:	Penalization of late predictions
	$c_t \in \mathbb{N}$:	Process execution costs = 100 if false negative prediction (*contractual penalty*) = 50 if true positive prediction (*adapt. costs*) = 100 if false positive prediction (*adapt. + compensation costs*)

Experimental Setup. We use the implementation of the predictive process monitoring system presented in [21]. We selected the BPIC 2017 data set[1], because the process instances covered by the data set are sufficiently long to observe the effect of earliness. Also, the data set has a sufficient number of process instances and predictions to allow us to observe convergence of learning.

[1] https://www.win.tue.nl/bpi/doku.php?id=2017:challenge.

Experimental Results. Figure 5 shows the results by depicting how the rate of adaptations, earliness (in terms of relative prefix-length when an adaptation was made), process execution costs, and overall rewards evolve. Other than in the web application case, we do not show the development of the state variables, because the state variables do not change smoothly over time, but may be quite different between each consecutive time step.

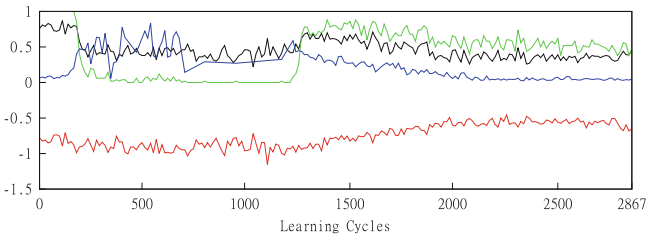

Fig. 5. Learning behavior for self-adaptive process monitoring system; **green** = rate of adaptations; **blue** = earliness; **black** = costs/100; **red** = overall reward/100 (Color figure online)

Like for the self-adaptive web application, the results show how our approach enables the system to adapt itself. It can be seen that the approach indeed is able to learn when to adapt in order to maximize rewards. The reward begins to increase after around 1,100 learning cycles and converges to a value of around −60 after 2,000 cycles. Also, the results show how the system learns the trade-off between accuracy and earliness. Up to learning cycle 250, the system always adapts as early as possible. However, this entails high costs due to low prediction accuracy at the beginning of process execution. Then, between learning cycles 250 and 1,250 the system learns that a very low rate of adaptations may lead to low costs. However, this does not help increase rewards, because many adaptations happen rather late (earliness around 0.5 on average). After learning cycle 1,250 the system starts learning that earlier predictions deliver higher rewards and thereby finds a trade-off between accuracy and earliness.

5 Discussion

5.1 Threats to Validity

Internal Validity. To observe whether policy-based RL in principle has the expected effect, we used a multi-layer perceptron as a simple neural network to represent the policy. In order not to trade the problem of finding a good quantization of the state space and a suitable setting of the exploration rate in value-based RL for hyper-parameter tuning in policy-based RL, we used only default hyper-parameter settings. Also, we repeated each of the experiments multiple times in order to assess potential random effects due to the stochastic nature of

the neural networks. Even though there were some differences in the speed of convergence, the learning behavior was consistent across these repetitions.

External Validity. We used two different kinds of self-adaptive information systems in our experiments. For the web application, we used state-of-the-art computer infrastructure and different workloads (both synthetic and real) to get realistic measurements. For the business process monitoring system, we used a large, public real-world data set. The learning problems for both systems differ in terms of the action variable (continuous vs. discrete). Still, both problems only consider a single action variable. For discrete action variables, this is due to the current limitations of our approach for what concerns the size of the action space (see below). For continuous action variables, our approach in principle is applicable to more than one action variable, because the used PPO algorithm can cope with that. We plan further experiments to confirm this.

5.2 Limitations

Handling Large Discrete Adaptation Spaces. The policy-based RL algorithms we use can handle a large action space during online learning, provided the action space is continuous. However, these algorithms do not naturally generalize over a set of non-continuous, i.e., discrete, actions and thus cannot extend to previously unseen actions [9]. Typically, self-adaptive information systems have large discrete action spaces, such as feature-oriented or architecture-based self-adaptive systems. As an example, take a system that offers ten optional system features that may be dynamically activated and deactivated in any combination. Its adaptation space thus contains $2^{10} = 1024$ adaptation actions. These 1024 adaptation actions cannot be represented as a continuous variable. For such self-adaptive systems with large discrete action spaces, our approach currently is not applicable. One solution may be to embed the discrete actions in a continuous space and use nearest-neighbor search to find the closest discrete actions [9].

Convergence of Reinforcement Learning. Performance of machine learning depends, to a large degree, on the amount of data available for learning. When used for self-adaptive systems, RL may require quite many learning cycles until the learning process converges [22]. In our experiments, learning for both subject systems required around 2,000 learning cycles (with data from 256,000 time steps) to converge. Until RL has converged, the system most likely executes inefficient adaptations, because not enough observations have yet been made. Inefficient adaptations may lead to negative effects, because they are executed in the live system [11]. To speed up convergence, one may aim to find good initial estimates for the learned knowledge [10,28,34] or perform offline learning via simulations of the system [31]. Still, online RL may not be applicable for systems that operate in an environment where the effects of the "trial-and-error" nature of RL may not be tolerable; e.g., if adaptation actions may harm the environment or if an adversary in the environment may maliciously manipulate input data.

6 Related Work

Existing approaches that use RL for online learning in self-adaptive information systems resort to value-based RL. We structure the discussion into how they represent the value function (also see Sect. 2).

Value Function as Lookup Table. Amoui et al. use SARSA to learn effective adaptation actions for a self-adaptive web application [1]. They radically quantize environment states by defining only two values for each of the state variables. To this end, thresholds defined by domain experts are employed. Huang et al. employ Q-Learning for the dynamic optimization of resource allocation in operational business processes [13]. In addition to optimizing the resource allocation for a single process instance, they propose an optimization across process instances by considering the global resource costs when updating the value table. Dutreilh et al. employ Q-Learning for autonomic cloud resource management [10]. They assume upper bounds for the state variables can be given; e.g., based on experimental observation. Bu et al. employ Q-Learning for the self-configuration of cloud virtual machines and applications [5]. To facilitate scalability, they define three discrete states, representing high, medium and low ranges of the respective state variable. Arabnejad et al. apply fuzzy Q-Learning and SARSA for cloud auto-scaling [2]. Environment states are quantized and thereby limited to small sets of states expressed in fuzzy logic. The benefit of fuzzy logic is that many states can be represented by only a few fuzzy states. However, their approach still requires identifying "discrete" fuzzy elements in the fuzzy set based on which RL operates. Caporuscio et al. propose using value-based RL for multi-agent service assembly [6]. The agents share state monitoring information and use Q-Learning with a tabular representation of the value function. Zhao et al. use Q-Learning in combination with case-based reasoning to generate and update adaptation rules [35]. They quantize continuous environment states using equidistant points. Wang et al. use multi-agent Q-Learning for adaptive service compositions [32]. They assume that the environment can be represented by a finite, discrete set of states. In contrast to the above approaches, our policy-based approach does not require discrete states or manual quantization, but it can directly handle large and continuous environments.

Value Function Approximation. Tesauro et al. [31] use Q-Learning with non-linear function approximation for autonomic cloud resource allocation. They use a neural network (multi-layer perceptron) for approximating the value function. They suggest using softmax or ϵ-greedy as exploration mechanism and observe that – for their specific subject system –, they can learn good policies without requiring exploration. Yet, they do not analyze whether this observation may generalize to other kinds of systems. Xu et al. use Q-Learning together with function approximation (via artificial neural networks) for autonomic cloud management [34]. Moustafa and Zhang use Q-Learning with linear function approximation (via linear regression) for QoS-aware web service composition [22]. Wang et al. use Q-Learning with function approximation via deep neural networks (recurrent neural networks) for adaptive service composition [33]. Silvander

propose using Q-Learning with function approximation via a deep neural network (DQN) for the optimization of business processes [27]. All these approaches use ϵ-greedy as exploration mechanism, requiring fine-tuning of the exploration rate. In contrast, our approach does not require explicitly controlling the exploration rate, but exploration is done automatically via probabilistic action selection.

7 Conclusion

We introduced and experimentally evaluated an online reinforcement learning approach to facilitate engineering of self-adaptive information systems. Our approach contributes to information system engineering by increasing the degree of automation. Concretely, our approach does neither require manually quantizing environment states nor manually having to determine suitable exploration parameters for the reinforcement learning algorithm to work. As future work, we will extend our approach to handle large discrete action spaces in order to capture additional types of self-adaptive systems. We will also investigate whether deep learning models facilitate better representation of the learned knowledge.

Acknowledgments. We cordially thank Claas Keller, Tristan Kley and Jan Löber for supporting us in carrying out the experiments, Zoltan Mann for comments on earlier versions of the paper, as well as the anonymous reviewers for their constructive comments. Research leading to these results received funding from the EU's Horizon 2020 research and innovation programme under grant agreements no. 780351 (ENACT), 731932 (TransformingTransport), and 871493 (DataPorts).

References

1. Amoui, M., Salehie, M., Mirarab, S., Tahvildari, L.: Adaptive action selection in autonomic software using reinforcement learning. In: 4th International Conference on Autonomic and Autonomous Systems (ICAS 2008), pp. 175–181. IEEE (2008)
2. Arabnejad, H., Pahl, C., Jamshidi, P., Estrada, G.: A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In: 17th International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2017), pp. 64–73. ACM (2017)
3. Aschoff, R., Zisman, A.: QoS-driven proactive adaptation of service composition. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 421–435. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25535-9_28
4. Barrett, E., Howley, E., Duggan, J.: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurrency Comput. Pract. Exp. **25**(12), 1656–1674 (2013)
5. Bu, X., Rao, J., Xu, C.: Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. IEEE Trans. Parallel Distrib. Syst. **24**(4), 681–690 (2013)
6. Caporuscio, M., D'Angelo, M., Grassi, V., Mirandola, R.: Reinforcement learning techniques for decentralized self-adaptive service assembly. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 53–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_4

7. Chen, T., Bahsoon, R.: Self-adaptive and online QoS modeling for cloud-based software services. IEEE Trans. Software Eng. **43**(5), 453–475 (2017)
8. D'Ippolito, N., Braberman, V.A., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: 36th International Conference on Software Engineering (ICSE 2014), pp. 688–699. ACM (2014)
9. Dulac-Arnold, G., Evans, R., Sunehag, P., Coppin, B.: Reinforcement learning in large discrete action spaces. CoRR abs/1512.07679 (2015)
10. Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., Truck, I.: Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In: 7th International Conference on Autonomic and Autonomous Systems (ICAS 2011), pp. 67–74 (2011)
11. Filho, R.V.R., Porter, B.: Defining emergent software using continuous self-assembly, perception, and learning. TAAS **12**(3), 16:1–16:25 (2017)
12. Franco, J.M., Correia, F., Barbosa, R., Rela, M.Z., Schmerl, B.R., Garlan, D.: Improving self-adaptation planning through software architecture-based stochastic modeling. J. Syst. Softw. **115**, 42–60 (2016)
13. Huang, Z., van der Aalst, W.M.P., Lu, X., Duan, H.: Reinforcement learning based resource allocation in business process management. Data Knowl. Eng. **70**(1), 127–145 (2011)
14. de la Iglesia, D.G., Weyns, D.: MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. TAAS **10**(3), 15:1–15:31 (2015)
15. Jamshidi, P., Camara, J., Schmerl, B., Kästner, C., Garlan, D.: Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots. In: 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2019). ACM (2019)
16. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
17. Klein, C., Maggio, M., Arzén, K.E., Hernández-Rodriguez, F.: Brownout: building more robust cloud applications. In: 36th International Confernce on Software Engineering (ICSE 2014), pp. 700–711. ACM (2014)
18. de Lemos, R., et al.: Software Engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
19. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**(4), 559–592 (2014)
20. Mann, Z.Á.: Resource optimization across the cloud stack. IEEE Trans. Parallel Distrib. Syst. **29**(1), 169–182 (2018)
21. Metzger, A., Neubauer, A., Bohn, P., Pohl, K.: Proactive process adaptation using deep learning ensembles. In: Giorgini, P., Weber, B. (eds.) CAiSE 2019. LNCS, vol. 11483, pp. 547–562. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21290-2_34
22. Moustafa, A., Zhang, M.: Learning efficient compositions for QoS-aware service provisioning. In: International Conference Conference on Web Services (ICWS 2014), pp. 185–192. IEEE Computer Society (2014)
23. Nachum, O., Norouzi, M., Xu, K., Schuurmans, D.: Bridging the gap between value and policy based reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS 2017), vol. 12, pp. 2772–2782 (2017)

24. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C.: A taxonomy of uncertainty for dynamically adaptive systems. In: 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), pp. 99–108 (2012)
25. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. TAAS **4**(2), 1–42 (2009)
26. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR abs/1707.06347 (2017)
27. Silvander, J.: Business process optimization with reinforcement learning. In: Shishkov, B. (ed.) BMSD 2019. LNBIP, vol. 356, pp. 203–212. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24854-3_13
28. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
29. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems 12 (NIPS 1999), pp. 1057–1063 (2000)
30. Teinemaa, I., Tax, N., de Leoni, M., Dumas, M., Maggi, F.M.: Alarm-based prescriptive process monitoring. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNBIP, vol. 329, pp. 91–107. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98651-7_6
31. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Comput. **10**(3), 287–299 (2007)
32. Wang, H., et al.: Integrating reinforcement learning with multi-agent techniques for adaptive service composition. TAAS **12**(2), 8:1–8:42 (2017)
33. Wang, H., Gu, M., Yu, Q., Fei, H., Li, J., Tao, Y.: Large-scale and adaptive service composition using deep reinforcement learning. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 383–391. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_27
34. Xu, C., Rao, J., Bu, X.: URL: a unified reinforcement learning approach for autonomic cloud management. J. Parallel Distrib. Comput. **72**(2), 95–105 (2012)
35. Zhao, T., Zhang, W., Zhao, H., Jin, Z.: A reinforcement learning-based framework for the generation and evolution of adaptation rules. In: International Conference on Autonomic Computing (ICAC 2017), pp. 103–112. IEEE Computer Society (2017)