DOI: 10.1561/9781680838251.ch6

Chapter 6

Online Reinforcement Learning for Self-Adaptive Smart IoT Systems

By Alexander Palm, Felix Feit and Andreas Metzger

6.1 Introduction

In this chapter we explain how Reinforcement Learning (RL) techniques can be leveraged to improve the way self-adaptive smart IoT systems (SIS) adapt at run-time.

The concept of self-adaptation facilitates developing software systems that are capable of maintaining their quality requirements even if the systems' environment changes dynamically [3, 18]. Self-adaptation thereby helps developing systems that can operate in a resilient way at run-time. To this end, a self-adaptive software system (such as a self-adaptive SIS) can modify its own structure, parameters and behavior at run-time based on its perception of the environment, of itself and of the fulfilment of its requirements. An example is a self-tuning thermostat for a Heating, Ventilation and Air Conditioning (HVAC) system. Based on its perception of the outdoor and indoor temperature it can control the strength of its heating and cooling devices in order to proactively reach a set point temperature to maximize user comfort. On the other hand it can learn to reduce energy by reducing heating and/or cooling strength when no user is present in the room.

To develop a self-adaptive SIS, software engineers have to develop self-adaptation logic that encodes when and how the system should adapt itself. Software engineers, for instance, may specify event-condition-action rules that determine which adaptation action is executed in response to a given environment change. Developing self-adaptation logic requires an intricate understanding of the software system and its environment, and how adaptations impact on system quality [6, 7]. Among other concerns, it requires anticipating the potential environment changes the system may encounter at run-time to determine 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 [7, 17]. In addition, while the principal 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 [7, 10]. One emerging way to address design time uncertainty is to employ online RL [1, 2, 4, 8, 12, 14, 22–24].

Online RL can learn the effectiveness of adaptation actions through interactions with the system's environment. This means that instead of software 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 software system engineer expresses the learning problem in a declarative fashion, in terms of the learning goals the system should achieve. Online RL thereby automates the manual engineering task of developing the self-adaptation logic.

Therefore, the remainder of this chapter is structured as follows: Section 6.2 motivates the application of RL in the realm of SIS by briefly introducing the topics of self-adaptive software systems (SASS) and RL. Section 6.3 combines the aforementioned topics and introduces the concept of policy-based RL and explains how it helps to address large continuous state spaces which is a main shortcoming of state-of-the-art approaches leveraging RL at run-time. Section 6.4 shows our experimental results after using our policy-based RL approach for the realization of a self-tuning thermostat in the smart building domain. Section 6.5 exemplifies how the reward function of a RL problem can be decomposed into several reward streams with different semantics to make decisions of an RL agent explainable. Finally, Section 6.7 concludes the chapter.

6.2 Fundamentals

In this section the fundamentals of self-adaptive software systems and Reinforcement learning are briefly introduced.

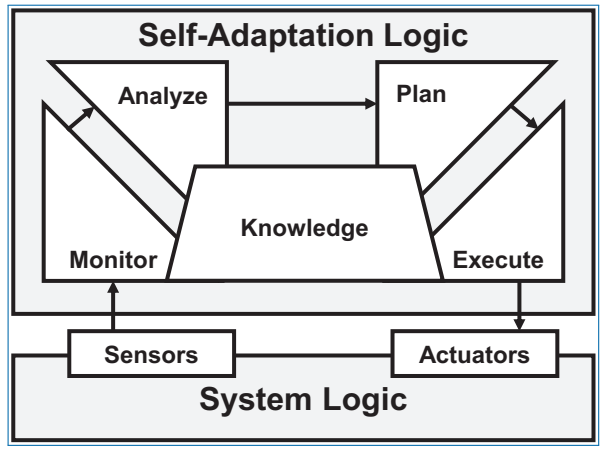

Figure 6.1. MAPE-K reference model for self-adaptive systems (based on [11]).

6.2.1 Self-adaptive Software Systems

A well-known reference model for self-adaptive systems is the MAPE-K model [11], which is depicted in Figure 6.1. Following this reference model, a self-adaptive software system can be logically structured into two main elements: the system logic (aka. the managed element) and the self-adaptation logic (the autonomic manager).

As shown in Figure 6.1, the self-adaptation logic can be further structured into four main conceptual activities that leverage a common knowledge base [9]. The knowledge base includes information about the managed system (e.g., encoded in the form of models at run-time), its environment, and its adaptation goals and adaptation policies (e.g., expressed as rules). The four activities are concerned with monitoring the system logic and the system's environment via sensors, analysing 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.

6.2.2 Reinforcement Learning

RL aims to learn suitable actions via an agent's interactions with its environment [20] as depicted in Figure 6.2. At a given time step t, the agent selects an action a (from its adaptation space) to be executed in environment state s. As a result, the environment transitions to s' at time step $t + 1$ and the agent receives a reward r for executing the action. The reward r together with the information about the next state s' are used to update the knowledge of the agent. The goal of RL is to optimize cumulative rewards. One fundamental problem in RL is the trade-off that must be made between *exploitation* (using current knowledge) and *exploitation*

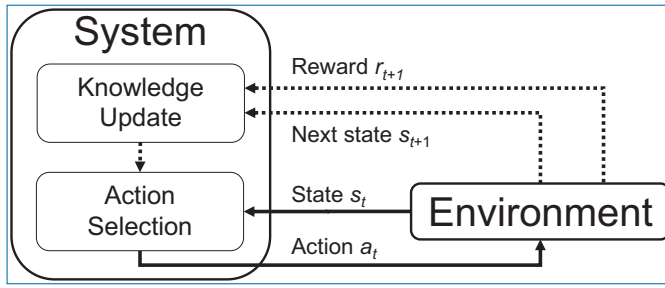

Figure 6.2. Schematic illustration of agent-environment interaction in RL (based on [20]).

(gathering new knowledge). For a self-adaptive service, "agent" refers to the self-adaptation logic of the service and "action" refers to an adaptation action [16].

6.3 OLE: Policy-based Online Reinforcement Learning

This sections introduces our online RL approach. In Section 6.3.1, we provide an overview, the conceptual ideas behind the approach how it differs from the state of the art. In Section 6.3.2, we explain how we prototypically realized the approach.

6.3.1 Overview of Our Approach

The innovative concept underlying the ENACT Online Learning Enabler (OLE) is that we use a fundamentally different type of reinforcement learning than what has been used in the state of the art. While the state of the art used value-based RL, we use policy-based RL. The main idea behind policy-based RL is to directly use and optimize a parametrized stochastic *action selection policy* [15, 21]. 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 (comprising the selected 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.

Figure 6.3 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.

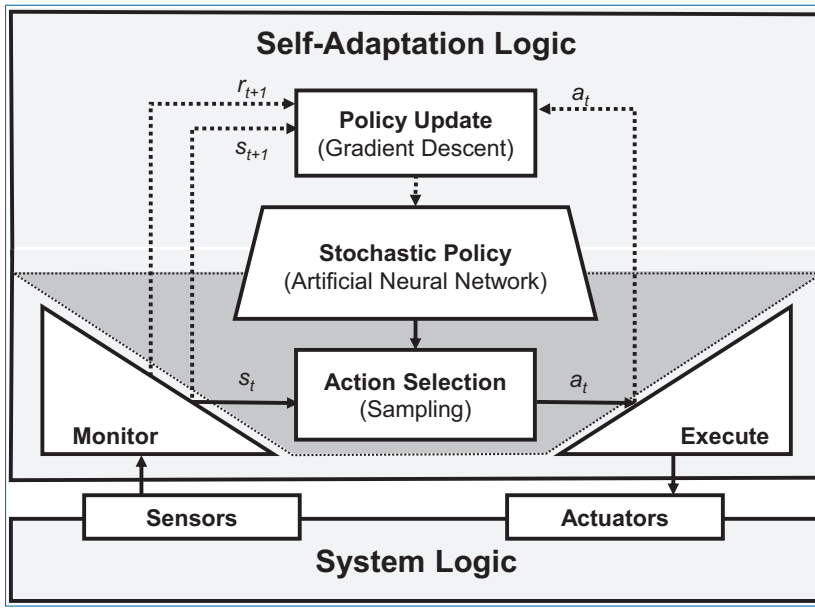

Figure 6.3. Conceptual architecture of policy-based approach.

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*. In our approach, state s_t may be determined using observations of both the system's environment and the system logic itself. This differs from the basic RL model, where only observations from the system's environment are considered to determine the state s_t. 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 [20, 21], because the policy is represented as an artificial neural network. Policy gradient methods update the policy according to the gradient of a given objective function, such as the average reward per learning cycle to give a simple example. 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.

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 an adaptation action a_t in state s_t will lead to a state s_{t+1}, and
- $R : S \rightarrow \text{IR}$, 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 $\theta \in \text{IR}^d$.

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 $w \in \text{IN}^+$. 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.

6.3.2 Prototypical Realization

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 [19]) 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).

6.4 Validation in the Smart Building Domain

To show the applicability of our policy-based RL approach on SIS we performed a series of experimental validations in the smart building domain (cf. Section 10.5). Therefore, the experimental setup is summarized in Section 6.4.1 before the underlying RL problem is formalized as a MDP in Section 6.4.2. Finally, Section 6.4.3 shows the results of our experiments.

6.4.1 Experimental Setup

In the according use case scenario we show that it is possible to learn a control strategy for a simulated HVAC system by means of policy-based RL. The simulated HVAC system is based on the ground floor of the KUBIK building (cf. Section 10.5) and comprises six multi-sensors providing information about user presence and temperature, as well as 5 fan coils whose capacity is accumulated to treat them as one single fan coil. An excerpt of the KUBIK specification showing the room used for the simulation is depicted in Figure 6.4.

The learned control strategy thereby should control the HVAC system in such a way that thermal comfort is achieved whenever the controlled room is occupied and energy consumption is minimized otherwise. The run-time of each experiment corresponds to one year of simulation, while one time-step of an experiment corresponds to one minute (i.e. total time-steps of one experiment: 524000). After several iterations of code improvements of the simulation we were able to simulate

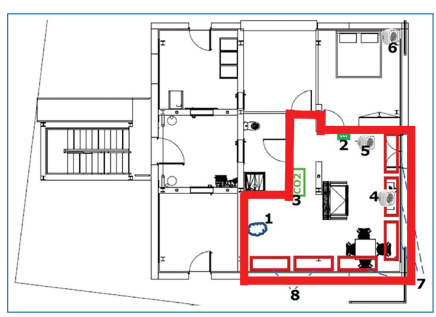

Figure 6.4. Excerpt of the KUBIK specification showing the room used for the simulation.

between 150 and 1000 time-steps per seconds depending on the hyperparameters (especially the size of the neural net) of the underlying algorithm.

As a baseline we implemented a simple on/off-controller, that heats or cools the room whenever the indoor temperature is not close enough to the user set-point and a user is present. If no user is present the thermostat controller remains inactive.

6.4.2 Problem Formalization as MDP

For the evaluation, in a first step we formulated the problem of learning an HVAC control strategy as a RL problem. The underlying Markov Decision Process (MDP) is thereby specified as follows:

State-space: The main state variables are stemming from the from simulation variables (e.g. indoor temperature). Furthermore, we created some crafted features resulting in variables relying on main state variables (e.g. deviation from setpoint). The variable predicted occupancy returns the probability that the room gets occupied within the next 30 min. This variable is computed based on the underlying occupancy pattern, as we assume that such variables might exist in modern HVAC systems (cf. [5]).

Action-space: As the main task of the control strategy is to properly control the HVAC device, the action-space comprises 7 discrete actions, where only one action could be selected at a time. The actions correspond to the different modes of the fan coil for heating and cooling, as well as a turning he device off. The different modes relate to different fan speeds resulting in different air flow rates and different temperatures concerning the integrated fluids for heating or cooling respectively.

Transition-dynamics: The transitions between the different states (depending on the selected action) are computed by the simulation according to the underlying thermal equations. As we employ model-free RL algorithms, the control strategy does not have access to the environmental model resulting from the simulation. It is learning through pure interaction with raw experience. The simulation could be treated as a real environment, with the main difference that with a real environment the run-time of an experiment would be much longer.

Reward: The main part of the MDP driving the algorithm in a direction to learn the right control strategy is the reward function. We defined the reward in such a way, that the algorithm gets a positive feedback whenever its control strategy keeps the indoor temperature close (i.e. within bounds of 1°C) to the user setpoint when a user is present and penalized otherwise (i.e. negative feedback corresponding to the deviation from the setpoint). As the control strategy should minimize energy consumption, the algorithm gets penalized according to the strength of the current action, whenever it performs an action other than turning the HVAC system off, if no user is present in the room (based on the simulated occupancy).

6.4.3 Results

We evaluate the results from two different perspectives: The domain perspective and the Reinforcement Learning perspective. For the domain perspective, we plotted the outdoor temperature curve and the indoor temperature curve showing the moving average of both variables. For the moving average we averaged the last 128 values to reduce the noise inside the diagram and improve the interpretability of the results. Furthermore, we plotted the average indoor temperature per occupancy phase. This gives us the option to see whether the learned control strategy is able to avoid heating or cooling actions in occupancy phases where no user is present and to keep the indoor temperature close to the user set-point whenever a user is present. Apart from this domain-related metric, we used the moving average reward of the last 10000 time-steps as a metric to evaluate the learning process from a RL-perspective. It is important to note that the values of the average reward are scaled to fit into the temperature diagram. The scaling factor is neglectable, as it is only the evolution of the reward curve that is important in this case. The exact evolution of the cumulative reward is shown in different plots to address solely the Reinforcement Learning perspective, when we compare the results of the RL approach to the baseline approach.

Figure 6.5 shows how the indoor temperature evolves according to the control strategy resulting from the baseline thermostat. As can be seen during the first

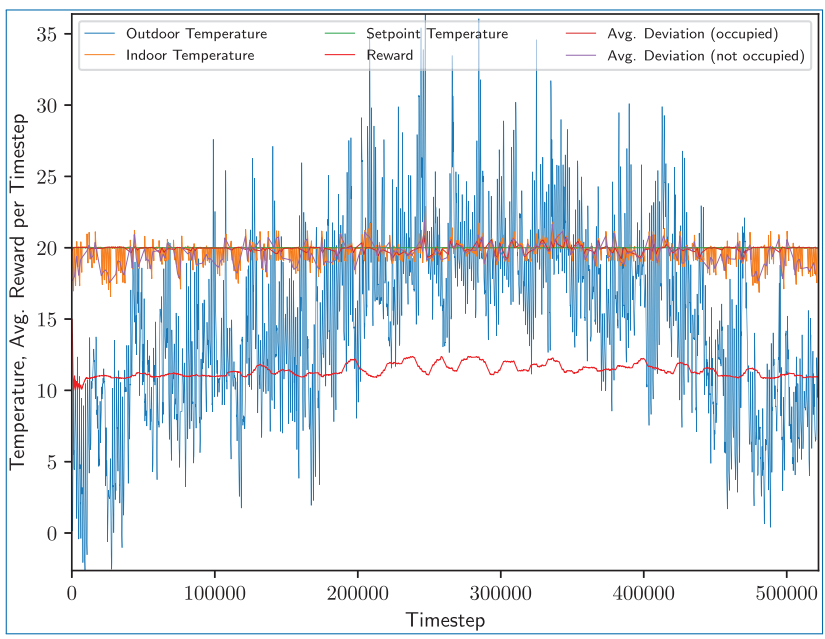

Figure 6.5. Temperature evolution of baseline thermostat.

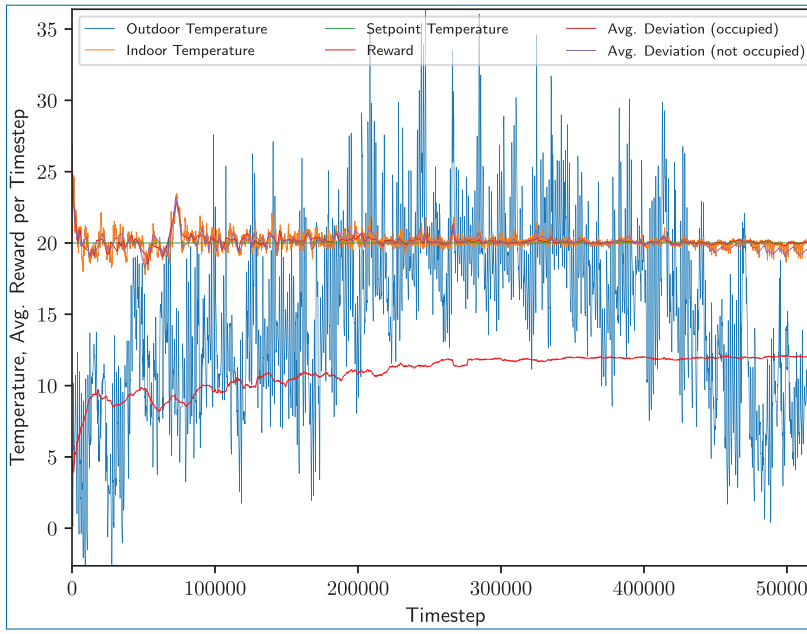

Figure 6.6. Temperature evolution of RL approach.

150.000 time steps, the indoor temperature drops if no user is present resulting from the heating device being turned off while the outdoor temperature is below the set point temperature. As the outdoor temperature becomes warmer (from time step 150.000 onwards) the indoor temperature is higher than the set point temperature (if not user is present) as the cooling device is turned off accordingly. In phases where a user is present the indoor temperature is kept around the set point temperature. However, from a RL perspective this leads to a non-optimal reward, because the thermostat controller is purely reactive and for every time step the indoor temperature is too far from the set point temperature this leads to a non-optimal reward.

In contrast, Fig. 6.6 shows how the indoor temperature evolves to a thermostat controller based on a policy-based RL approach. After a learning phase (until time step 260.000) the RL approach is able to keep the indoor temperature around the set point temperature if a user is present and reduce energy consumption by turning off the heating or cooling device otherwise. Especially during the end of the experiment, after learning has been converged and the approach can reuse its knowledge about low outdoor temperatures (from time step 450.000 onward), it can be seen that the same spikes can be observed as in 6.5. However, the drops in the indoor temperature are not as big as with the baseline approach and the reward is slightly higher.

Figure 6.7 shows the results from the RL perspective visualizing the cumulative reward evolution. The baseline approach outperforms the RL approach in terms of

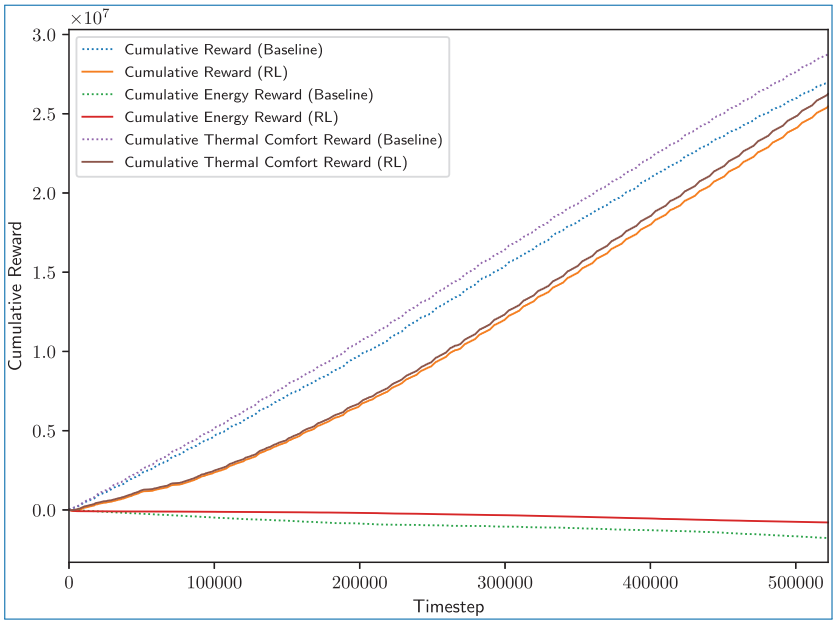

Figure 6.7. Cumulative reward (baseline vs. RL approach).

cumulative reward, resulting from the poor initial performance during the learning phase. This is due to the fact that the RL approach has no initial knowledge about a goal-directed behavior. However, after learning has converged (about time step 260.000) the increase in reward becomes more steep than that of the baseline approach, which can be seen from time step 400.000 onward.

After having shown that the RL approach is able to outperform the baseline approach after its learning process has converged, we did further investigate how it may perform if its knowledge has been initialized a priori. To do this we did set up a slightly modified version of the HVAC simulation, with the thermal dynamics being simplified (and not following complex thermal equations). To perform some kind of pretraining we let the RL approach learn with this simplified environment for the same amount of time steps as in the online experiment.

As it can be seen in Fig. 6.8 the pretrained version of the RL approach is able to adapt its control strategy to the real thermal dynamics pretty fast and after around 50.000 time steps the reward curve converges. This results from a goal-directed behavior with the pretrained RL approach being able to keep the indoor temperature around the set point temperature whenever a person is present and reduce energy consumption otherwise. The reason for the slightly better reward compared to the baseline approach can be seen in Fig. 6.9. The RL approach learns to avoid the indoor temperature moving too far from the set point temperature, to be able to reach the set point temperature within fewer time steps than the baseline approach.

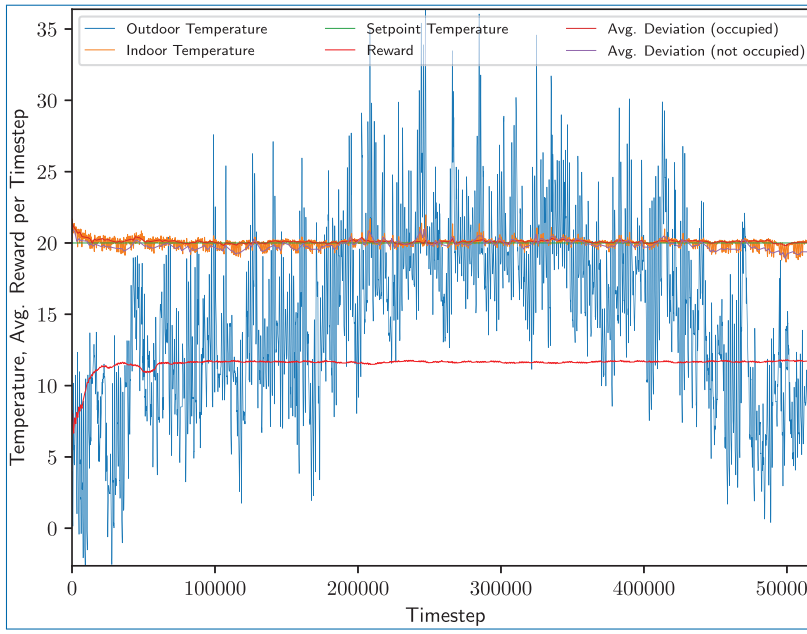

Figure 6.8. Temperature evolution of pretrained RL approach.

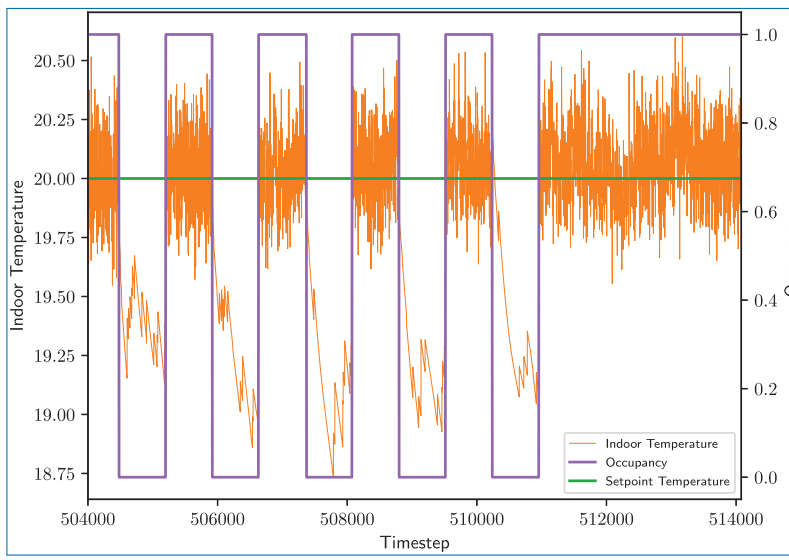

Figure 6.9. Temperature evolution of baseline thermostat.

This is done by proactively heating or cooling resulting in small penalty for consuming energy while the room is not occupied. However, this penalty is rather small to the penalty that would be received during a time step (with the room being occupied) where the indoor temperature is not around the desired set point temperature.

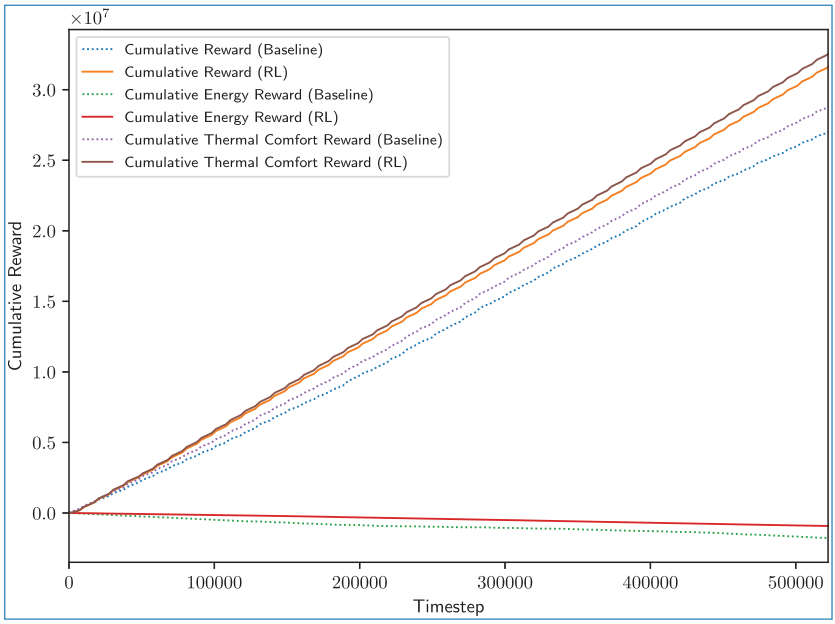

Figure 6.10. Cumulative reward (baseline vs. pretrained RL approach).

Finally, when having a look at the evolution of the cumulative reward as depicted in Fig. 6.10, the pretrained RL approach clearly outperforms the baseline approach.

6.5 Explaining Adaption Decisions via Reward Decomposition

Reward Decomposition is a method in which several value-based RL agents are trained in parallel on different aspects of an environment. At each time step the knowledge of the agents is aggregated to provide a global decision. To apply this method, it is necessary that the reward function of the examined environment can be decomposed into independent subfunctions. As a result, instead of returning a scalar value at each time step, the reward function returns a vector where each component reflects the reward of one subfunction or zero if there was no reward or punishment at the associated time step.

For each component of the reward vector an independent subagent is trained, which receives the global state as observation and as reward only one component of the reward vector. In order to derive the action of the overall agent, at each time step the action values of all subagents are summed up element-wise and on the basis of these aggregated action values an action is selected (e.g. by using epsilon greedy action selection).

This technique was applied to a simplified version of the HVAC environment (see Chapter 6.4) where the cooling capability was removed. In this environment the reward was split into the two subfunctions "thermal comfort" and "energy cost". The first of these functions always gives a negative value, i.e. a penalty if the person is present but the current temperature is not within a certain tolerance around the desired temperature. On the other hand, the second component "energy costs" contains a constant negative value if action "heating" was chosen in the last step. One value-based RL agent (e.g. DQN) is then trained for each of these two components of the reward vector. At each time step the action values of both agents are summed up for both actions "heating" and "not heating". The greater sum then marks the greedy action of the overall agent.

To generate explanations for actions, the action-values of the subagents can be put in relation to each other. For this purpose, the difference between the action-value of a particular action and the action-values of all alternative actions is calculated for each subagent. This is repeated for all actions and yields the relative importance per action per subagent. The higher the relative importance of an action, the more influence the subagent has on the selection of this action. By observing the behaviour and relative importance the reasoning of an agent can be deduced.

To further increase the explainability of the approach, the reward function can be broken down into situations instead of subfunctions. Each situation represents a set of special states of the environment. In addition, each situation receives a non-zero reward or punishment value that is always given when the agent is in that situation and zero otherwise. For example, the HVAC environment can be deconstructed into the following four situations:

- occupied and within the tolerance (reward: $+1$)
- occupied and out of tolerance (reward: -5)
- unoccupied and within the tolerance (reward: -1)
- unoccupied and out of tolerance (reward: $+0.1$)

For each of these situations, as before, a separate agent is trained and the relative importance is calculated. If the importance is then set in relation to the sum of the relative importance of all actions, the relative importance of an action in relation to a specific situation can be calculated. These values can then be summed up to obtain the relative importance of an action across all situations. Using these two metrics, the following natural language string can be generated for each time step:

"With my current knowledge I am 97% sure that action 'not heating' is better. Arguments in favour of action 'not heating' are the prevention of situation 'occupied and out of tolerance' (84%), the occurrence of situation 'occupied and within the tolerance' (9%), and the prevention of situation 'unoccupied and within the

tolerance' (4%). An argument in favour of action 'heating' is the occurrence of situation 'unoccupied and out of tolerance' (2%)."

The first sentence of this explanatory string contains the relative importance of an action across all situations (97%) and the following sentences describe the relative importance of an action in relation to a specific situation (84%, 9%, 4%, and 2%).

6.6 Synergies with Behavioural Drift Analysis

The main goal of our Online Reinforcement Learning approach is to learn an optimal control strategy for a MDP. Despite of evaluating the learned control strategy from a RL perspective, it needs to be evaluated from a domain perspective as well. The latter can also be used to guide the engineering of the reward function. As the behavioral drift analysis is based on a model capturing the desired control strategy in an abstract way, the computed signal might be used for the evaluation of the actual control strategy. Unexpected changes in the behavioral drift signal can then be interpreted as an indicator for context changes that make it impossible for the learning system to behave according to the obtained model. To showcase this synergy we derived a model for BDA based on the desired behavior described in Section 6.4.2 (cf. Figure 6.11).

The behavioural model depicted in Figure 6.11 represents the expected indoor temperature which depends on whether or not a person is present in the room and this, independently of the underlying temperature management system. It relies

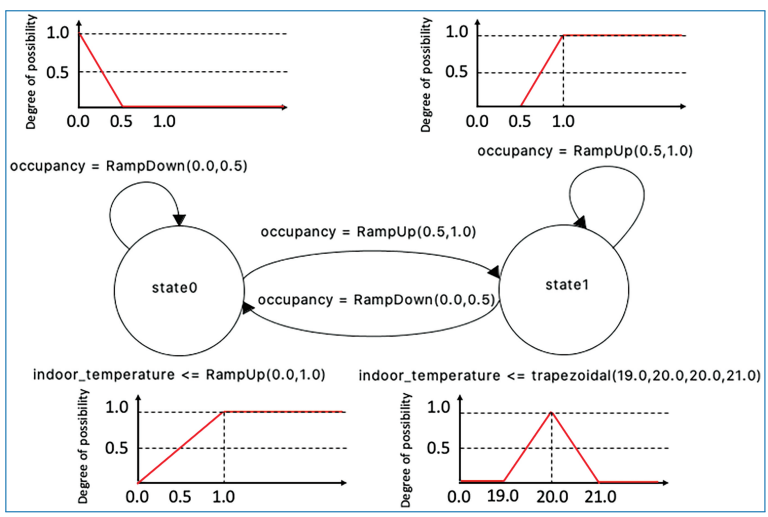

Figure 6.11. Possibilistic Input/Output Hidden Markov Model describing the expected indoor temperature depending on whether a person is present or no in a room.

on the possibility theory where distributions are defined as membership functions. The model defines the expected behaviour as follows: when a person is present in the room, the temperature must be equal to 20°C (state 1). When nobody is in the room, the temperature is expected to be greater than 1°C (state 0). In addition to fully accepted temperature values (where degree of possibility is equal to one), the model defines some tolerances. For instance, for the state 1, temperatures below 19.0°C and above 21°C are totally rejected (degree of possibility = 0) while temperatures in between 19.0°C and 21.0°C different from 20°C, while not being perfect, are not totally rejected (0< degree of possibility <1). In conjunction with temperature and occupancy sensor values, this possibilistic Input/Output Hidden Markov Model (IOHMM) is used to compute the behavioural drift as the likelihood (possibility measure) of the observation sequences to have been generated by the model, i.e. the likelihood that the temperature is managed in such a way that it remains within the accepted boundaries defined by the model.

6.7 Conclusion and Outlook

We motivated the application of Reinforcement Learning as a means to enable a software system to adapt itself to changing context situations in the realm of SIS. Furthermore we introduced a concrete realization of an Online Learning approach which overcomes the main shortcomings of state-of-the-art approaches (e.g. ability to handle continuous parameters as actions and avoid manual fine-tuning of exploration). Our policy-based Reinforcement Learning approach for a self-adaptation logic has been validated in the smart building domain by applying it to an HVAC control problem. The experiment results have shown that our approach is able to outperform static thermostat implementations by dynamically learning to control the heating and cooling devices of a smart building. This has been achieved by finding a trade-off between the maximization of user comfort and minimization of energy consumption. Additionally, we introduced our conceptual work on the process of decomposing a reward function of a RL problem into several reward streams with different semantics to make decisions of an RL agent explainable and proposed how our Online Learning approach can be enriched by the concept of Behavioral Drift Analysis.

As future work, we envision extending our approach for online reinforcement learning for self-adaptive Smart IoT Systems along the following two main dimensions:

Better Pre-training As we demonstrated above, pre-training the reinforcement learning enabler may deliver better performance during operations. On the one hand, the initial performance (directly after deployment to run-time) can

be increased. On the other hand, the overall speed of learning and learning performance can be increased, in particular in real-world situations where rewards are sparse. Yet, such offline pre-training again faces the uncertainty issue when formulating the source learning task to be learned in the offline setting. It is not possible due to design time uncertainty that this source learning task faithfully captures the actual online setting. To capture the problem of uncertainty, existing solutions thus make certain assumptions about the system and its uncertainty in order to be able to perform the training in the offline setting. This is also what we did above, by taking certain assumptions about the building domain and even taking real, historic data into account. However, while this may mean that the reinforcement learning enabler learns a policy that solves this specific problem (i.e., under the given assumptions), the learned policy can be useless or may even perform worse than a policy only trained online when applied to the actual problem at run-time (which may violate these assumptions), even if it is relatively similar. One approach to this problem is to leverage the emerging concept of deep meta reinforcement learning.

Coping with large discrete action spaces Existing online reinforcement learning solutions for self-adaptive services propose randomly selecting adaptation actions for exploration he effectiveness of exploration therefore directly depends on the size of the adaptation space, because each adaptation action has an equal chance of being selected. Some reinforcement learning algorithms can cope with a large space of actions, but require that the space of actions is continuous in order to generalize over unseen actions. Self-adaptive Smart IoT Systems may have large, discrete adaptation spaces; *e.g.* if their adaptations entail reconfigurations of many system features or a large set of discrete parameters. In the presence of such large, discrete adaptation space, random exploration thus may lead to slow learning at run-time. One approach to this problem is to leverage the structure of the adaption space to better guide the exploration process. In related work, we have demonstrated that such improved exploration is possible for cloud services [13]. It thus can serve as a promising basis for applying to Smart IoT Systems.

References

[1] Mehdi Amoui *et al.* "Adaptive action selection in autonomic software using reinforcement learning". In: *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE. 2008, pp. 175–181.

[2] Hamid Arabnejad *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)*. IEEE. 2017, pp. 64–73.

[3] Rafael Aschoff and Andrea Zisman. "Qos-driven proactive adaptation of service composition". In: *International Conference on Service-Oriented Computing*. Springer. 2011, pp. 421–435.

[4] Enda Barrett, Enda Howley, and Jim Duggan. "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. In: *Concurrency and Computation: Practice and Experience* 25.12 (2013), pp. 1656–1674.

[5] Enda Barrett and Stephen Linder. "Autonomous hvac control, a reinforcement learning approach". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2015, pp. 3–19.

[6] Tao Chen and Rami Bahsoon. "Self-adaptive and online qos modeling for cloud-based software services". In: *IEEE Transactions on Software Engineering* 43.5 (2016), pp. 453–475.

[7] Nicolas D'Ippolito *et al.* "Hope for the best, prepare for the worst: multi-tier control for adaptive systems". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 688–699.

[8] Xavier Dutreilh *et al.* "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow". In: *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*. 2011, pp. 67–74.

[9] Didac Gil de la Iglesia and Danny Weyns. "MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems". In: *TAAS* 10..3 (2015), 15:1–15:31.

[10] Pooyan Jamshidi *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)*, IEEE. 2019, pp. 39–50.

[11] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1 (2003), pp. 41–50.

[12] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. "A review of auto-scaling techniques for elastic applications in cloud environments". In: *Journal of grid computing* 12.4 (2014), pp. 559–592.

[13] Andreas Metzger *et al.* "Feature Model-Guided Online Reinforcement Learning for Self-Adaptive Services".In: *Service-Oriented Computing – 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings*. Ed. by Eleanna Kafeza *et al.* Vol. 12571. Lecture Notes in Computer Science. Springer, 2020, pp. 269–286. DOI: 10.1007/978-3-030-65310-1_20. URL: https://doi.org/10.1007/978-3-030-65310-1%5C_20.

[14] Ahmed Moustafa and Minjie Zhang. "Learning efficient compositions for qos-aware service provisioning". In: *2014 IEEE International Conference on Web Services*. IEEE. 2014, pp. 185–192.

[15] Ofir Nachum *et al.* "Bridging the Gap Between Value and Policy Based Reinforcement Learning". In: *Advances in Neural Information Processing Systems 12 (NIPS 2017)*. 2017, pp. 2772–2782.

[16] Alexander Palm, Andreas Metzger, and Klaus Pohl. "Online reinforcement learning for self-adaptive information systems". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2020, pp. 169–184.

[17] Andres J. Ramirez, Adam C. Jensen, and Betty H.C. Cheng. "A taxonomy of uncertainty for dynamically adaptive systems". In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2012, pp. 99–108.

[18] Mazeiar Salehie and Ladan Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4.2 (2009), pp. 1–42.

[19] John Schulman *et al.* "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[20] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[21] Richard S. Sutton *et al.* "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12 (NIPS 1999)*. 2000, pp. 1057–1063.

[22] Gerald Tesauro *et al.* "On the use of hybrid reinforcement learning for autonomic resource allocation". In: *Cluster Computing* 10.3 (2007), pp. 287–299.

[23] Hongbign Wang *et al.* "Integrating reinforcement learning with multi-agent techniques for adaptive service composition". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 12.2 (2017), pp. 1–42.

[24] Tianqi Zhao *et al.* "A reinforcement learning-based framework for the generation and evolution of adaptation rules". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE. 2017, pp. 103–112.