



Automated cloud resources provisioning with the use of the proximal policy optimization

Włodzimierz Funika¹ · Paweł Koperek¹  · Jacek Kitowski^{1,2}

Accepted: 31 October 2022 / Published online: 10 November 2022
© The Author(s) 2022

Abstract

Many modern applications, both scientific and commercial, are deployed to cloud environments and often employ multiple types of resources. That allows them to efficiently allocate only the resources which are actually needed to achieve their goals. However, in many workloads the actual usage of the infrastructure varies over time, which results in over-provisioning and unnecessarily high costs. In such cases, automatic resource scaling can provide significant cost savings by provisioning only the amount of resources which are necessary to support the current workload. Unfortunately, due to the complex nature of distributed systems, automatic scaling remains a challenge. Reinforcement learning domain has been recently a very active field of research. Thanks to combining it with Deep Learning, many newly designed algorithms improve the state of the art in many complex domains. In this paper we present the results of our attempt to use the recent advancements in Reinforcement Learning to optimize the cost of running a compute-intensive evolutionary process by automating the scaling of heterogeneous resources in a compute cloud environment. We describe the architecture of our system and present evaluation results. The experiments include autonomous management of a sample workload and a comparison of its performance to the traditional automatic threshold-based management approach. We also provide the details of training of the management policy using the proximal policy optimization algorithm. Finally, we discuss the feasibility to extend the presented approach to further scenarios.

Keywords Reinforcement learning · Policy gradient optimization · Heterogeneous cloud resources · Automatic management

✉ Paweł Koperek
pkoperek@gmail.com

Extended author information available on the last page of the article

1 Introduction

Many software systems designed nowadays exploit the cloud computing infrastructures which offer high availability, security and the flexibility to allocate the resources on-demand. The last factor often drives the decision to implement a specific system using cloud resources as it allows to greatly reduce the costs of running a distributed application. Such elasticity unfortunately requires paying the price of designing the application to handle scaling events, e.g., changing the number of virtual machines (*horizontal scaling*) or adding or removing RAM, CPU or storage (*vertical scaling*). Deploying the application requires also creating a policy which will define the conditions under which the system should be scaled and which resources should be utilized in such a case. It might be possible to create a configuration which will work correctly over a long period of time if the environment shows stable seasonal usage patterns. Unfortunately, in many cases such patterns do not exist, what calls for using an automatic scaling policy. We can define it as *a dynamic process, often operating on a Physical Machine (PM), that adapts software configurations (e.g., threads, connections and cache, etc) and hardware resources provisioning (e.g., CPU, memory, etc) on-demand, according to the time-varying environmental conditions* [1].

The area of the Reinforcement Learning (RL) techniques has been explored for a long time [2, 3]. Initially the techniques and algorithms from this category could be only used in relatively simple problems. It was assumed that the environment can be observed with use of only a few metrics and there are not that many actions to execute. Handling more complex domains became possible with recent advancements in, e.g., computer games [4], robot control [5] or the game of Go [6]. One of the main drivers of progress has been the application of Deep Learning in various forms, e.g., Deep Q Learning [7], or Policy Gradient Optimization (PGO) methods like proximal policy optimization (PPO) [8], Phasic Policy Gradient [9]. One of the main advantages of the mentioned methods is the ability to learn through observing and interacting with an environment which is similar to or is the same as the one the agent is going to operate in.

One of the most popular methods in DRL is the Deep Q Learning [7]. It is an extension of a classic Q Learning algorithm in which the policy is a given state (s) always chooses an action (a) of highest *quality*. The quality is defined through a Q function ($Q(s, a)$), which given a combination of state and action can provide a numeric value which can be easily used to compare between different actions. It is quite common that function is hard to define analytically, hence it is approximated through an iterative algorithm. Unfortunately such an approach is not well suited for problems in which there are many actions and possible states. The Deep Q Learning attempts to solve that issue by using a neural network to approximate the Q -function. This approach rendered interesting results, however has some shortcomings. It cannot be easily applied to environments where not all information is known and included in the state, in which probabilistic policies are preferable. Q-Learning focuses on satisfying the Bellman equation, hence it indirectly optimizes the policy's behavior. Those shortcomings can be addressed

by using a PGO method, which optimize the policy's parameters directly and generate probabilistic policies. Such an approach allowed to surpass the performance of humans in a computer game [10], control 3D bodies in a simulated arena [11, 12] or solve Rubik's cube using a robotic arm [13]. There are many variants of the PGO methods: Advantage Actor-Critic (A2C) [14], Asynchronous Actor-Critic Agents (A3C) [14], Deterministic Policy Gradient (DPG) [15], Soft Actor Critic [16] or proximal policy optimization (PPO) [8].

Many successful experiments suggest that Deep Reinforcement Learning (DRL) is a generic approach what encourages further research and application to other domains. One area which could benefit from that, is the automatic scaling of distributed applications deployed to a cloud infrastructure. The cloud resources become the environment where an automatic agent operates, their state becomes the state which is subject to change. Cloud vendor API calls become the actions the agent can potentially execute. Measurements and metrics which can be used to determine the mentioned state are driven by the technologies used to implement the application and thanks to that they are well defined. The goals of the system are also clear (e.g., reduce RAM consumption, CPU load, request latency, cost of resources) what helps to translate them into a *reward function*. Such a reward function becomes the feedback mechanism for the agent and allows to evaluate the impact of the executed actions. Thanks to that the agent does not need to rely on any prior knowledge and can use a process of trial-and-error experiments to discover the optimal management policy.

Unfortunately, with this approach the cost of creating the DRL policy becomes the main disadvantage. The training algorithm needs to go through multiple iterations of interacting with the managed system and observing its responses. Especially at the beginning the actions chosen for execution might be quite random, what can easily destabilize the observed application, even make it completely unusable for the end users. Since such a situation is unacceptable in a production system, the training requires a separate, duplicate environment. This increases unfortunately the overall cost of running the system. Another approach, which does not incur such high costs, is to use an artificial, simulated environment. In such a scenario, the observed workloads can be replayed multiple times in a reliable fashion. Additionally, the flow of time in such an environment can be controlled what allows to speed up the simulation and reduce the training time. Usually, the DRL training processes can also benefit from increasing the number of iterations, what can help obtain a more efficient policy.

Both Deep Q Learning and PGO methods have been applied to automatic control of cloud resources [17–21] Both approaches have their strengths and weaknesses. It is important to evaluate which of them can render superior results in a given environment. Unfortunately, in many cases PGO methods are not taken into account. They are very rarely used in the context of real-world test-beds. One example is [22]. In that paper, the Deep Deterministic Policy Gradient (DDPG) algorithm is used to create a control policy which sets thresholds of the traffic flow control system. While that use case allowed to optimize the work of a cluster of machines, it addressed only the networking aspect of data processing. Unfortunately in that example the controlled cluster is homogeneous. All the available resources have the same configuration. In many cases,

though, especially in the context of automatic scaling of applications deployed to very popular cloud infrastructures, there could be potentially employed resources of more than just one type. Depending on the context of a situation, using a more or less powerful resource might be the most beneficial. In our previous work we evaluated various algorithms from the PGO family [23] and the DQN approach and observed that the PPO provides the best results in our environment. We have also demonstrated how to leverage the described ideas to create a system capable of automatic scaling of a homogeneous cloud infrastructure hosting a CPU-intensive workload. In the current paper, we extend this approach to heterogeneous cloud resources: the system can adjust not only the amount of resources (i.e., the number of virtual machines) but can also decide on the type of the resources (i.e., the type of the virtual machines) to adjust. The training does not require providing any additional information about the managed application or specifying resources capabilities. All decisions are derived from the experience gained from simulations. The automatic management system under discussion has been implemented as an extension to Semantic-based Automatic Monitoring and Management (SAMM) monitoring software [24] and includes a decision-making component utilizing a PGO training method.

The contributions of this paper are as follows:

- We provide an extension to an existing automatic management system [25] with capabilities to manage heterogeneous resources.
- We provide the design and implementation of a control policy capable of controlling heterogeneous resources and a policy training procedure based on the Deep Reinforcement Learning approach.
- We demonstrate the correctness of the presented approach. We train a policy based on an *long-short term memory* (LSTM) [26] Deep Neural Network and deploy it to a real world cloud-based application. While DRL and LSTM have been combined previously together, they were used in simulated environments. In this paper we demonstrate how a system leveraging an LSTM policy trained with the use of a PGO method can be used in a real cloud environment.
- We analyze the efficiency of the new policy: compare the cost of resources used by the policy in a sample scenario with the cost of resources used by a threshold-based policy, typically available in a cloud environment.

The paper is organized as follows: in Sect. 2 we review the related work. Section 3 describes the design and architecture of the environment under discussion whereas Sect. 4 explains the policy training procedure. Section 5 discusses the design of the experiment and the environment it was executed in. Section 6 provides the experiment results and their discussion. Section 7 summarizes our research and outlines further work.

2 Related work

In this section we present the research which provides a broader context for our work.

2.1 Automatic resources management

Minimizing the monetary cost of cloud resources while maintaining the business requirements (sometimes defined through Quality-of-Service metrics) is a very complex task and has been an active research area for years. There are many different approaches available. The decision on which of them should be chosen, depends on the features of the environment which should be managed (e.g., granularity of managed resources, available actions, etc.). The mentioned approaches can be categorized in multiple ways. In [1] authors described a taxonomy which uses the following features to classify automatic management systems:

- *self-awareness*—the capability to obtain and maintain information about the state of the system. There are many kinds of awareness: *interaction awareness*, *time awareness*, *stimulus awareness*, *goal awareness*, *meta awareness*.
- *self-adaptivity*—the capability to adjust own decision policy to new circumstances. Depending on the area which those decisions affect, we can distinguish between variants of *self-configuring*, *self-healing*, *self-protecting* and *self-optimizing*.
- *architectural patterns*—the way the process of auto-scaling is structured (what are the components and how do they interact with each other). The three best known approaches are: *Feedback loop* [27], *Observe-Decide-Act* (ODA) [28], *Monitor-Analysis-Plan-Execute* (MAPE-K) [29]
- *QoS modeling*—the controls which allow to adjust the managed system and a model which connects those controls with QoS metrics. The models can be categorized into three groups: static (connection between metrics and how they affect the way resources are allocated is defined prior to starting the system, e.g., [30]), semi-dynamic (based on machine learning models [31] or simulation [32]), dynamic (resource allocation policies based on statistical analysis of historical workload logs, e.g., [33]).
- *granularity of control*—what are the basic objects which are under the control of the agent and are used to perform scaling: virtual machine [34], container [35], application [36].
- *decision-making*—the specification of process which leads to taking a decision about changing the resource allocation. It should include a definition of objectives and their representations, an algorithm of how to reason and search for a decision, a definition of controls which need to be adjusted.

There have been multiple attempts to tackle this issue, with the most distinctive being: rule-based control [37, 38] (an action execution occurs when a condition defined a priori is met), control theory-based [39] (control theory mechanisms are used to make a decision), search-based optimization [31, 40–43] (decisions form a large, finite search space and choosing among them is treated as a search problem; Machine Learning-based attempts are also included in this category).

The approach presented in this paper can be classified as *self-optimizing goal, stimulus aware*, with a dynamic QoS modeling capability. The control agent we

have developed adjusts the capacity of the system by adding and removing Virtual Machines. The policy is created by using a Deep Reinforcement Learning-type algorithm which allows to classify it among search-based optimization.

2.2 Reinforcement learning

One of the more active areas of research in machine learning is the Reinforcement Learning [2, 44]. Its primary focus is to discover a policy for agents which autonomously take actions within a specific environment. The policy maximizes a reward whose value is returned to the agent. The process of training an agent relies on executing a series of actions. After each of them the agent observes their consequences and builds up its own knowledge. There is no *supervising entity* providing feedback on how taking a certain action is better than taking others. This distinguishes this approach from *supervised learning*. RL is also different than *unsupervised learning* which focuses on discovering the internal structure of a collection of unlabeled data.

Over the years many different approaches to RL were proposed. We can broadly categorize them as:

- *Policy-based* and *value-based* which focus on training different components of the decision-making mechanism. The former explicitly builds a representation of the policy (a function which maps a state to an action which should be taken in that state). The latter creates a value function, which can be used e.g., to compare the values of different actions, and that in turn can be used to make a decision.
- *Monte-Carlo* and *Temporal Difference* in which the training update is implemented either after the control episode finishes (Monte-Carlo), or after one or more steps are executed in an episode (Temporal Difference).
- *Online* and *offline* which differ in when the agent's policy is changed. In case of the *online* approach, an update happens after every step, in the *offline* case—after the full *episode* (i.e., when the training scenario is finished, the environment needs to restart and the reward is presented to the agent).
- *Model-based* and *model-free* which differ in how the environment is modeled by the agent. In the former approach an explicit model is created (e.g., through reward estimations or specification of state transitions), in the latter one-creating such a model is not necessary (the decision-making process assumes that it is sufficient to have a sample of information about state transitions).

Combining Deep Learning techniques with the *model-free* approach became popular recently and resulted in creating so called *Deep Reinforcement Learning*. In this approach, neural networks can be used to create an approximation of a function which is a part of an algorithm (e.g., the Q-function in [7]). Alternatively, in the case of policy gradient methods, neural networks can be used directly as the policy functions. The training process adjusts their weights (Θ) based on the gradient of an estimated scalar performance objective function $J(\Theta)$ in respect to those policy parameters:

$$\Theta_{k+1} = \Theta_k + \alpha \nabla_{\Theta} J(\Theta_k) \quad (1)$$

where Θ_k denotes policy's parameters in the k -th iteration of the training process. The performance is usually understood as a reward returned from the environment. There are multiple versions of policy gradient optimization. In our research we focus on the *proximal policy optimization (PPO)* [8].

Algorithm 1 Proximal Policy Optimization - Policy Training Algorithm

GAE - generalized advantage estimation method [12]
AdamOptimizer - weight optimization algorithm [45]
ExecuteAction - function which applies the selected action to the environment and returns the observed reward
 b = training batch size
 I = total interactions count
 M = optimization epochs
 $n = \frac{I}{b}$
 $last_gae = 0$
for $i=1, 2, \dots, n$ **do**
 for $s=1, 2, \dots, T$ **do**
 $a_s = \pi_{\Theta_{old}}(State)$
 $r_s = ExecuteAction(a_s)$
 end for
 // \hat{A} values are used in L^{CLIP} optimization
 for $s=T, T-1, \dots, 1$ **do**
 $\hat{A}_s = last_gae = GAE(r_s, last_gae)$
 end for
 $\Theta = AdamOptimizer(L^{CLIP}, \Theta_{old}, M)$
 $\Theta_{old} = \Theta$
end for

It can be outlined as the algorithm presented in Algorithm 1. It is the basic variant of the algorithm presented in [8]. The aim of the algorithm is to calculate the policy parameter update in such a way that it ensures that the new parameters are not changed by more than a predefined threshold value. That helps to avoid a situation where a single parameter update changes the policy in such a negative way, that it would require many training iterations to recover. This goal is achieved through modification of the objective function which is defined as follows:

$$J(\Theta) = L^{CLIP}(\Theta) = \mathbb{E}_t [\min(r_t(\Theta)A_t, \text{clip}(r_t(\Theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (2)$$

where:

- \mathbb{E}_t denotes calculating the average over a batch of samples at timestamp t ,
- A_t is an estimator of the advantage function which helps to evaluate which action is the most beneficial in a given environment state,
- r_t marks the probability ratio $r_t(\Theta) = \frac{\pi_{\Theta}(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}$ in which $\pi_{\Theta}(a_t|s_t)$ denotes the probability of taking an action a in state s by a stochastic policy and Θ_{old} are the policy parameters before the update,

Fig. 1 Visualization of the *clip* function

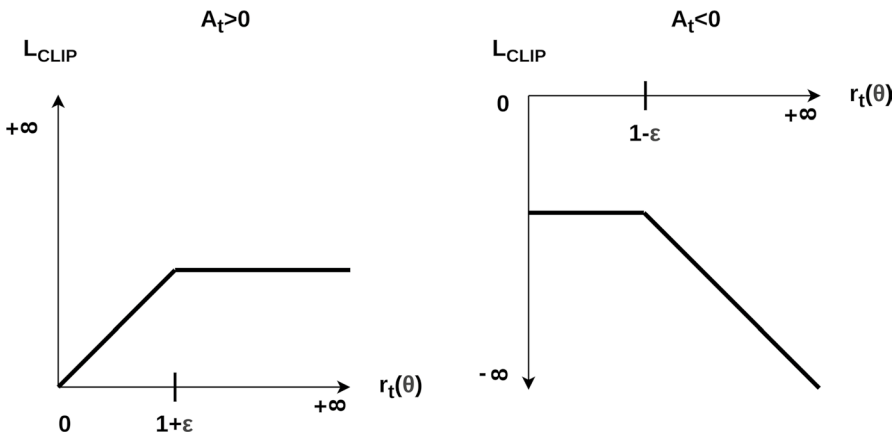
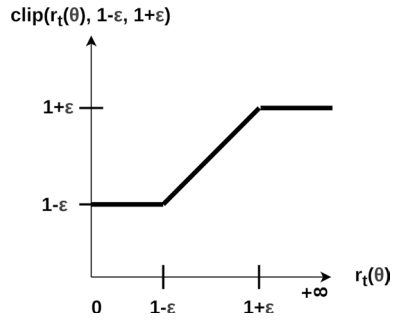


Fig. 2 Visualization of the influence of using the *clip* function on the L^{CLIP} . r_t as a ratio of probabilities is always greater than 0 hence we analyze two cases: when $A > 0$ (left) and $A < 0$ (right)

- $clip(r_t(\Theta), 1 - \epsilon, 1 + \epsilon)$ function keeps the value of $r_t(\Theta)$ within some specified limits (*clips* it at the end of the range), as shown in Fig. 1,
- ϵ is a hyperparameter with a typical value between 0.1 and 0.3.

The improvements to the training progress offered by the PPO algorithm come from the introducing of the *clip* function. The clipped results (after applying the *clip* function) are compared to the un-clipped r_t and the smaller value is chosen. This means that part of the change is ignored only if it would have a too big positive influence on the result of L^{CLIP} (the negative influence is unchanged). That influence is depicted in Fig. 2.

Another important component of PPO is the *advantage function* (A_t) which helps to value how good an action compared with other available actions for a specific state is. To estimate the advantage function values the *generalized advantage estimation* (GAE) method [12] is being employed. The advantage function is valued in the units corresponding to the ones used in the reward function.

The policy ($\pi_{\Theta_{old}}$) is typically implemented as a neural network, which parameters are shared with another network whose is employed by the GAE.

2.3 Applying deep reinforcement learning to automatic resources management

In practice, the complexity of cloud systems constantly increases, which makes it increasingly harder to model accurately. This translates to issues with applying classic approaches to automatic control (e.g., threshold-based rules, which also become more and more complex). That encourages using Machine Learning techniques, where a policy can be trained directly by using the collected measurements. This includes using the most recent advancements in the form of the DRL. Applying DRL in this context has a number of advantages. DRL is capable of creating complex decision-making policies due to the use of DNN. Using a training process which exploits the gradient descent algorithm allows to adjust the policy parameters to various optimization objectives. The past experiences can be memorized thanks to the use of experience replay buffers.

The DRL approach has been already used to automate many different tasks in resource management. In [17, 46–49] the authors train a policy which allows to allocate tasks (or jobs) to specific servers. [18, 50] demonstrate how to create an agent which can allocate VMs to sustain the workload under management. In [51] authors explain how a DRL agent can be used to control resources of Mobile Edge Computing (both compute and networking) resources. [52] presents an agent which is capable of horizontally and vertically scaling resources used to process a given workload.

DRL policies can be trained to optimize a variety of objectives, including: resources cost [18, 19], resource utilization [52], service latency [49], energy consumption [17, 48], task turnaround time [46]. It is also possible to optimize a combination of such objectives, e.g., service latency and energy consumption together [51]. Most often, the trained policies are evaluated by observing their behavior in the context of a sample workload running in a simulated environment [17, 19, 46, 48, 49, 51, 52]. The performance in many cases is very promising, however it is relatively rare for the control policies to be tested in real-world systems [18, 47, 50].

The Deep-Q-Learning [7] approach seems to be most widely exploited in the resources management domain [17–19, 52]. Methods which derive from that approach, e.g., Double Deep Q-Learning [50, 51], Stack Autoencoder Q-Network [47], continuous-time Q-learning for SMDP [48], Dueling Deep Q-Learning [49] are also relatively popular.

There are also examples of using Deep-Q-Learning with other techniques, e.g., in [53] authors described a novel, multilevel hybrid architecture in which agents are trained with the use of that algorithm. The described system has been used to manage allocation of workloads to cloud resources and focused specifically on the VM placement problem. It allocated a VM to a specific host when it had arrived at the cloud system, then monitored the cloud resources and their SLAs and if necessary relocated the VM to optimize profit or energy goals or to meet an SLA. The proposed architecture consists of three levels of control components:

Node Controllers (which dynamically adjust configuration to satisfy demand on each node), Lead Nodes (which are higher level controllers for groups of Node Controllers), Data Center Controllers (which manage the Lead Node controllers). Authors demonstrated that Deep-Q-Learning can be used to train agents working on all of those levels. To validate the presented approach an experiment in a simulator has been conducted. Its results demonstrated a significant improvement in reducing SLA violations compared to an established heuristic (Modified Best Fit Decreasing).

In [54] a modified version of the Deep-Q-Learning algorithm is presented. The modification includes Successive Over-Relaxation(SOR). Authors analyze the performance of the new algorithm by training policies to play games available in the Atari collection. Additionally they attempt to create a policy to horizontally scale resources (virtual machines) used by cloud applications. The experiments are performed in a simulated environment which uses HTTP logs from ClarkNet and NASA servers as workload. The experiments showed an improvement over the basic version of DQN approach.

There are also first examples of the use of policy gradient methods. In [22] a system for automatic traffic optimization (*AUTO*) is presented. Authors implement it with the use of the Deep Deterministic Policy Gradient (DDPG) training algorithm, which utilizes two neural networks: the *actor* (responsible for making decisions) and the *critic* (used to evaluate the actor's decisions). The former one consists of two fully connected hidden layers with 600 neurons each. The latter one reuses the two mentioned layers and adds an additional fully connected layer on top. Such a model is used to demonstrate the performance and adaptability of the discussed approach to the control of dynamic traffic flow consisting of web search and data mining requests.

In [20] authors use a combination of Convolutional Neural Networks (CNN) and fully connected layers to create a policy which is used to schedule processing jobs in a data center. They use the Advantage Actor-Critic (A2C) algorithm to train it. The resulting policy is evaluated using the *average job waiting time* and *average job slowdown* metrics. The experiments are carried out in a simulated data center cluster containing a number of nodes with two resources: CPU and memory. Authors concluded that the proposed method performed better than the widely used Shortest Job First (SJF) and Tetris [55] approaches.

In [21] a system for automatic allocation of Spark framework executors to virtual machines is developed. Authors use two algorithms for the policy training: Deep Q-Learning and a policy gradient method called REINFORCE. In order to evaluate how policies generated by the mentioned algorithms perform, a simulated environment has been created. It represents a cloud-deployed Spark cluster which executes jobs from the BigDataBench benchmark suite. The pricing model was similar to the AWS EC2 instance pricing in Australia. The experiments showed that the training using the PGO method was more stable than DQN and allowed to achieve superior results in terms of cost-efficiency and lower average job duration. In some cases though, the classic algorithms e.g., Integer Linear Programming (ILP) and Adaptive Executor Placement (AEP) were able to outperform the DRL policies. The results were very promising, yet limited to a single type of a workload (Spark framework).

In [25] we demonstrated how a similar algorithm from the policy gradient optimization family, the proximal policy optimization (PPO) [8], can be used to horizontally scale cloud resources. Initially, we have also experimented with the very popular DQN approach, however in our environment we found it hard to generate a policy which would reasonably scale the resources. On the other hand, PPO allowed to quite quickly reach a stable result. Figure 3 presents a sample training progress for both algorithms. The control policies were trained using a 100,000 steps in the simulated environment. The episode length depends on actions taken by the policy. In the case of DQN the episodes varied in length with some episodes taking many simulation steps to complete. That resulted in a reduced number of completed episodes, compared to the PPO algorithm.

In our previous work the implementation was limited to control resources of a single type. In the present paper we extend that approach to include resources of other types and employ a recurrent network to represent the policy. We explain how a policy can be trained using a synthetically created, simulated workload, and then present how the results of training in a simulated environment can be transferred to a real-world environment. We provide an evaluation of the behavior of the control policy deployed to manage AWS cloud resources used by a scientific application. Unfortunately, automatic scaling with the use of experimental management systems is typically tested in simulated environments and rarely deployed to real cloud infrastructures. Such results also differ with regard to many details, e.g., the type of managed workflow, for this reason direct comparison is very hard. To provide a reference point on the quality of decisions made by the presented system, we compare it with a threshold-based management policy which is available in the employed cloud infrastructure.

3 Architecture

In this section, we present the architecture of the automatic management system under discussion. The system manages cloud infrastructure resources which are used to host a distributed application and uses RL techniques to create a decision policy.

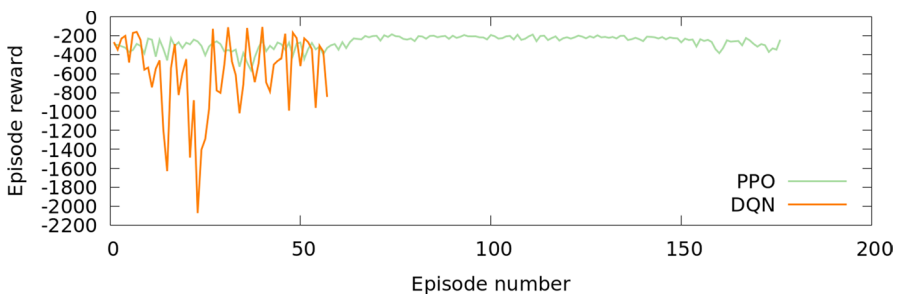


Fig. 3 Training progress of DQN and PPO: the reward obtained in subsequent episodes of the resource management simulation

One of the main challenges in the design of an autonomous management system is how to organize the training process. Using an environment with real cloud resources—nearly exactly the same as in which the policy is going to operate, would be the best solution. Unfortunately, such a scenario would usually introduce significant additional costs which may outweigh the benefits of automatic resource allocation. Using the actual, production environment is not an option either. Using an untrained policy would most likely lead to a significant degradation of performance due to its poor decisions and consequently to business losses. To avoid such a situation, we decided to use a simulation as a training environment, which is a common solution to this problem [56]. This allows to train the policy in an isolated, safe environment. Regardless of the decisions made, their consequences do not influence a real infrastructure, what allows experimenting even with the actions that lead to catastrophic events.

Using a simulator had a big influence on the discussed system's architecture. It required introducing an interface between the policy and the environment it was operating in. Such an interface had to create an abstraction which would hide whether the policy was accessing a simulation or real cloud resources. The capabilities delivered by the simulator and the API built on top of the cloud vendor libraries had to be aligned to each other.

From a high-level perspective, a system in which an autonomous control policy is deployed, needs to use some form of a feedback loop. Such a feedback loop comprises, on the one hand, a stream of actions triggered by the policy and, on the other hand, an information about the state of the observed environment, which allows to understand the consequences of those actions. The presented system also follows this pattern. First, the information about the state of cloud resources is obtained by monitoring the components of the managed application. That data is then aggregated into a form that can be used by the neural network acting as the control policy. Finally, the output from the network is interpreted as the identifier of the action to execute. This action is then implemented in the managed cloud infrastructure through an available cloud API. The components of the system and the implemented feedback loop are presented in Fig. 4.

The loop starts with collecting the measurements about the resources which take part in executing the workload (marked with number 1 on the diagram). Each of the resources is configured to start reporting relevant measurements as soon as the resource becomes online. The measurements often differ in their nature, which influences how often their values are delivered, e.g., the amount of free RAM and CPU usage is reported every 10 s while the virtual machine (VM) count—once per minute. To simplify the implementation of collecting of those raw measurements, we introduced the Graphite monitoring tool [57] (marked with number 2 on the diagram). Graphite aggregates all the collected values into a single interval to create a consistent snapshot of the environment. In our case this interval is set to one minute.

Next, the measurements are passed to the SAMM monitoring and management system [24] (marked with number 3 on the diagram). SAMM enables experimenting with new approaches to management automation. It allows to easily add support for new types of resources, relevant metrics, integrate new algorithms and technologies and observe their impact on the observed system. In our use case, SAMM is used

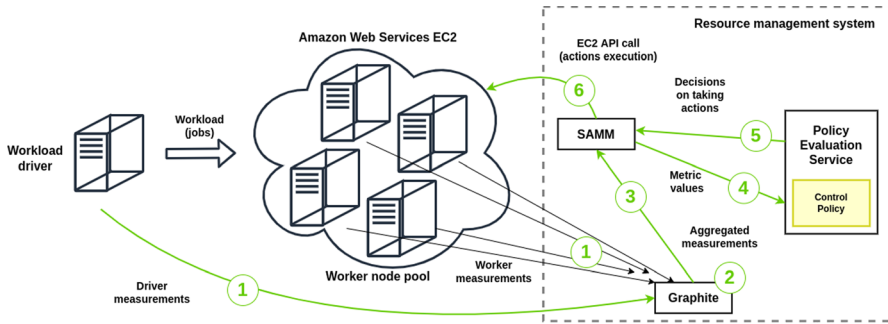


Fig. 4 Components of the discussed system. Arrows denote interactions between them

to pass information between the other elements of the system. It periodically polls measurements which portray the current state of the system and aggregates measurements into metrics used by the decision policy. Next, it delivers the current state of the system in a form of metric values, to the *Policy Evaluation Service* (marked with number 4 on the diagram). Finally, it retrieves decisions (marked with number 5 on the diagram) and executes them through the cloud vendor API (e.g., Amazon Web Services API) taking into account the environment constraints (e.g., the warm-up and cool-down periods; marked with number 6 on the diagram). SAMP calculates values of the following metrics: ratio of allocated cores, average CPU utilization, 90th percentile of CPU utilization, average RAM utilization, 90th percentile of RAM utilization, ratio of jobs waiting for processing to the number of jobs submitted, ratio of jobs waiting for processing to the number of jobs submitted in the last monitoring interval.

The *Policy Evaluation Service* provides decisions on how to change the allocation of resources based on the results of evaluation of the observed system state. The decisions are made according to the policy trained with the use of the PPO algorithm. The results of the evaluation may include:

- *starting a new VM of a specific type*—deficient resources are used to handle the workload under the current system state,
- *removing resources—shutting down VM of a specific type*—in the current state of the system, the resources are underutilized,
- *doing nothing*—a proper amount of resources is allocated.

One should remember that implementing the change to the resources allocation is always subject to the environment constraints. Not always it is possible to immediately execute an action. We might need to wait for a while because: the system is in a *warm-up* or *cool-down* (a period of inactivity to allow to stabilize the metrics after the previous action has been executed), the previous request might still be being fulfilled, the request failed and needs to be retried in some time. In order to be able to train a policy which can cope with such limitations, the factors need to be involved in the simulation used for training.

The described system makes a few assumptions about the workload it helps to manage. First of all, the workload needs to be organized into many independent tasks, otherwise it would not be possible to distribute the work among a number of resources. It is also necessary to provide a possibility to monitor the number of tasks which are yet to be executed. This implies creating an explicit queue of jobs which are submitted for processing. If such a component does not exist in the controlled system, it needs to be created. Most importantly, the tasks need to be idempotent (i.e., executing them multiple times does not change the end result) and the scheduling subsystem needs to be able to track their progress. Since the resources can be added and removed at any point in time, an interruption of a task before it terminates successfully (e.g., in case the processing VMs are shutdown) needs to be treated as a normal, common situation. In case of any failure, the scheduler should automatically reschedule the relevant tasks. For safety reasons, the resources that are administering the workload (e.g., accepting the input requests) should be isolated, and not included into the automatic management. This should prevent the workload from getting accidentally terminated.

Fulfilling the monitoring requirements may require introducing *extensions* to the software which generates the workloads and *instrumenting the resources* which are used to create tasks.

4 Using simulation in policy training

The system presented in Sect. 3 requires a policy to operate. Such a policy could be potentially trained as the first stage of managing a system. Using an environment with real cloud resources would be the best solution for the purpose of training. In such a case, the agent would be able to learn about all of the details of the controlled environment. Unfortunately, as mentioned before, with this approach the cost of creating a DRL policy would become a major challenge. The training algorithm needs to go through multiple iterations of interacting with the managed system and observing responses what significantly increases the resource consumption and, consequently, the costs of training.

In our system a simulation environment has been chosen as a foundation of the training process. This approach has huge advantages of cost efficiency and isolating the training from production environments. From the training perspective it has also a range of interesting properties. Since the simulation is isolated, the training process can be replicated and parallelized to allow for evaluation of multiple agents at the same time. This increases the number of interactions which can be tested by the policy within a given amount of time. The flow of time in a simulation can be changed (e.g., sped up) which allows to further reduce the time required to conduct training. The behavior of the environment and that of the workload are fully deterministic and can be easily repeated if needed. This makes the training predictable and repeatable and helps to tune the training algorithm parameters.

The policy training process has been implemented using an environment which was different from the real-world one. This environment is depicted in Fig. 5.

The simulator can replay any workload written in the Standard Workload Format, including the workload collection available as the Parallel Workloads Archive [58]. Jobs are submitted for processing according to the order and timing defined in the workload traces. Their actual execution is simulated. This allows the system to behave differently, depending on the actions taken by the trained agent. The mentioned pre-recorded workloads may span over many months or years and can include huge numbers of jobs to process. In order to make the training process faster, the flow of time can be speeded up. In such a case, the events occurring in the simulation are not processed immediately when they happen. Instead, events which happen within the same, configured time interval, are grouped together and processed as a batch. This may result in inaccuracy of the simulation (e.g., if time flow is speeded up 1000 times, new events can be scheduled for processing only in the next group, i.e., in the group which includes events from the following 1000 time units). The simulator also allows to adjust other parameters, e.g., the cost of resources by type, SLA penalties (queue wait penalty), maximum counts of VMs per type, etc.

The simulation includes a single datacenter with a configurable number of host machines. Hosts have uniform configuration, each of them can support multiple Virtual Machines (VMs). The resources used by the virtual machines are directly mapped to the resources of the hosts. In other words the simulation does not allow to over-provision simulated hosts. The number of virtual machines available at the beginning of the simulation is configurable as well.

The simulator has been implemented following the results of our prior research [23]. Its main process utilizes the CloudSim Plus simulation framework [59]. To decouple the simulator from other components and allow for easy reuse, it is additionally wrapped with the interface provided by the Open AI Gym framework [60]. This helps to easily launch experiments with various RL algorithms independently of the system we have developed.

5 Experimental design

In order to evaluate our approach, we designed an experiment in which we wanted to compare our policy with a different approach (threshold-based control policy). The overall objective was to perform sample computations while limiting the cost of the

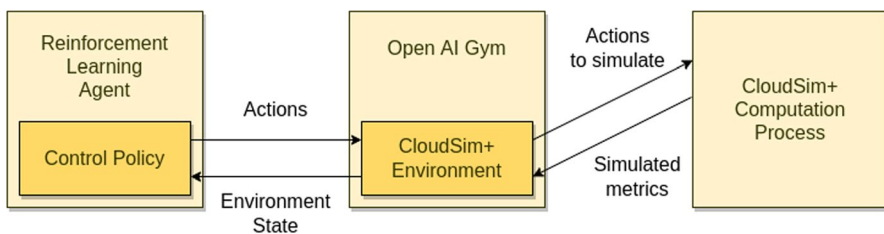


Fig. 5 Components of the training system; arrows denote interactions between them

used cloud resources. The diagram which explains the experiment is presented in Fig. 6.

First, we have identified a sample workload which we believe could benefit from automated resource provisioning. Then we prepared a simulated training environment, which resembled the target environment in which the real application would be managed. We prepared a simulation workload and conducted a training of the control policy (phase 1 on the diagram). Next we have deployed the sample application into a publicly available cloud infrastructure and configured our system to manage that application using the trained policy (phase 2 on the diagram). Finally, to provide a reference point on the performance of our solution, we have attempted

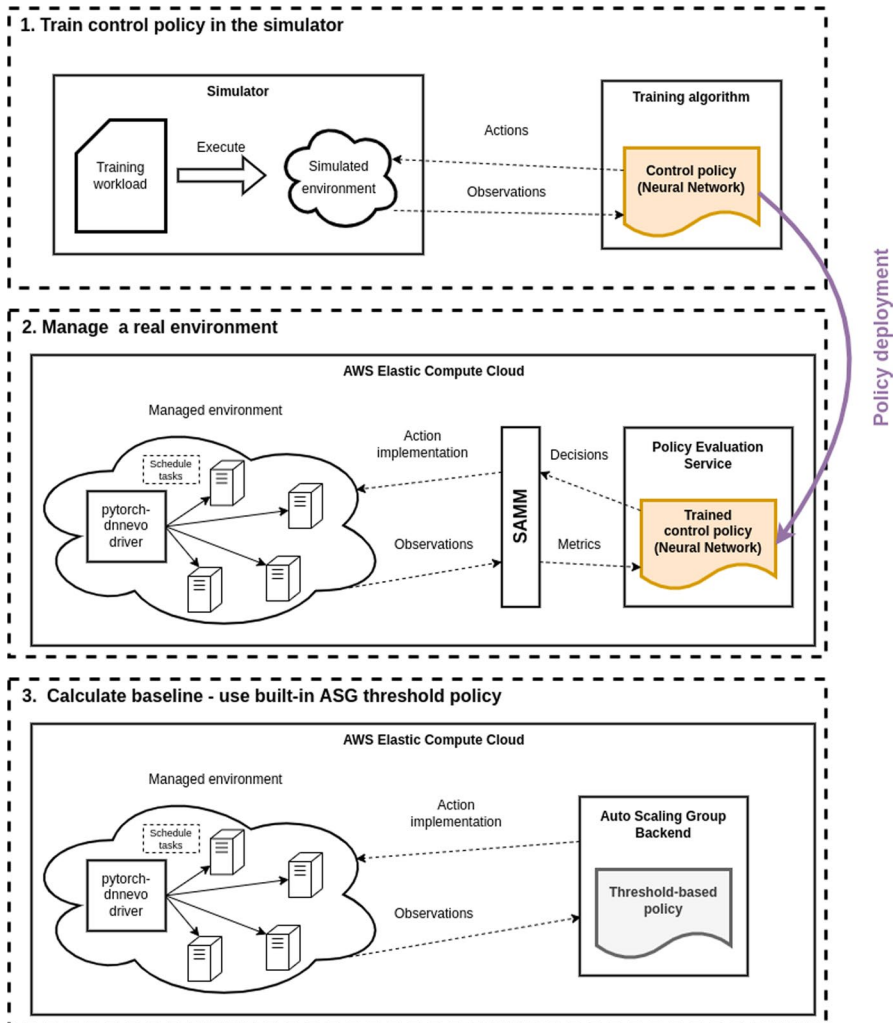


Fig. 6 Phases of the discussed experiment

to manage the sample application with the use of a publicly available tool: the threshold-based scaling policies provided in Auto-Scaling Groups (phase 3 on the diagram).

5.1 Workload

As a sample workload, we have used the *pytorch-dnn-evolution* tool [61]. This is a tool which attempts to discover an optimal structure of a Deep Neural Network (DNN) to solve a given problem (e.g., to categorize images in a given set) using a co-evolutionary algorithm. Such an approach can be used for domains where supervised learning techniques can be used, i.e., there are well-defined training and test datasets. Unfortunately, due to the size of those datasets, in many such problems, evolution-based methods are costly and time consuming. The evaluation of individuals (complete DNNs), which is required for the evolution process to progress, includes training them over the mentioned large datasets. To mitigate this issue, the co-evolutionary algorithm interleaves two evolutionary processes. Such an approach is possible due to the fact that an absolute objective function is not always necessary to identify which individuals should be promoted to the next iteration of evolution. This can be achieved by comparing the fitness of individuals with the use of an approximation of such a function. Using a subset of the original training set (the so called *fitness predictor*) is one of such solutions. The *first evolutionary process* evolves the DNNs to find the best neural network architecture for a particular task. Since in many cases training over the complete dataset would be too costly to repeat in the context of the whole population, training is being conducted over a small portion of the dataset (the *fitness predictor*). The elements of that dataset need to be carefully chosen. In a way, we can describe them as samples which are *the hardest* from the point of view of the evaluation. This is the purpose of the *second* evolutionary process: it aims at discovering such subset of the initial training dataset. It uses the best DNN from the first process to evaluate potential subsets. The subset which receives the lowest evaluation score becomes the subset which is used by the *first* process to evaluate DNNs. In this approach the amount of data used to conduct the evaluation. This in turn results in greatly speeding up the comparison between individuals and thus makes the evolutionary approach a viable option for problems which can be translated into supervised learning processes. The described evolutionary process is depicted by Fig. 7.

The evolutionary algorithm produces a high number of relatively small tasks that are independent of each other and can be easily processed in parallel on a cluster of machines. The workload scheduling is resilient to task failures and reschedules tasks in case processing them have not succeeded. The capacity of the job queue is in practice infinite thanks to the small size of a single job description. This means we can safely assume that none of the tasks is going to be dropped due to technical limitations of the queue system. Each task is going to successfully complete, regardless of how many times it needs to be restarted. Those features help to implement support for scaling events: each virtual machine used to conduct training can be safely shut down at any time. New machines can be added and start the processing of the

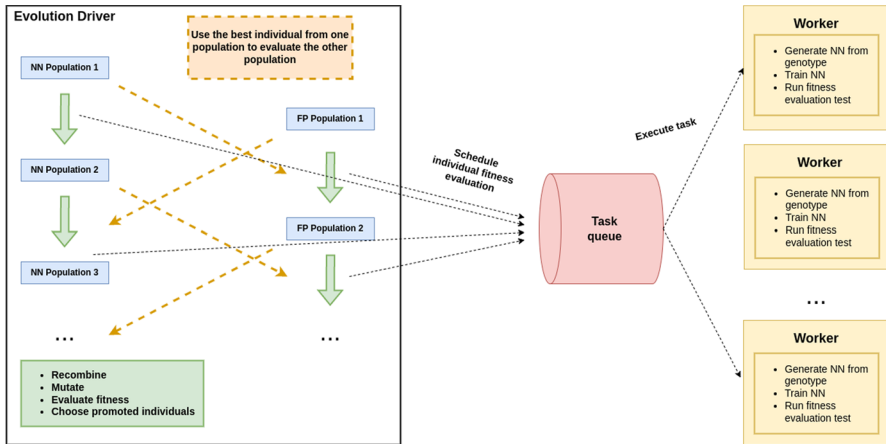


Fig. 7 Evolutionary process used as a sample workload. In the experiment described in Sect. 5, *pytorch-dnneo* is deployed and managed in a publicly available cloud environment (AWS)

evaluation tasks *ad hoc*, without additional configuration. The number of tasks varies over time and is hard to predict upfront. This renders an opportunity to reduce the cost of running the evolutionary process by reducing the amount of the used resources (VMs) when the demand for them drops.

In our case, the evolutionary process tried to find an optimal architecture for a neural network which recognizes handwritten digits. We have ran 20 iterations of evolution over a population of 32 individuals and 16 fitness predictors (subsets of 2000 images from the large training set). The evaluation of a single neural network comprised the training over 10 iterations of a given fitness predictor. We used the MNIST dataset [62] as the training set from which subsets are selected.

5.2 Target environment

As a compute infrastructure we have used the Amazon Web Services Elastic Compute Cloud (AWS EC2) [63]. The managed environment consisted of three Auto Scaling Groups of *m5a.large*, *m5a.xlarge* and *m5a.2xlarge* virtual machines which could have up to 10 instances each. All VMs were running in the US North Virginia region and in the same availability zone to avoid the problems with network latency added by multi-zone setups. The workload driver, together with SAMM and Graphite, has been running on a separate VM.

As mentioned above, for the purpose of the training process, we have simulated a single datacenter capable of hosting VMs of three types: small, medium and large. Their specification followed the configuration of Amazon's *large* (2 core CPU and 8 GB of RAM), *xlarge* (4 core CPU and 16 GB of RAM) and *2xlarge* (8 core CPU and 32 GB of RAM) EC2 instances. Each simulation started with one virtual machine of each type active and ran until all the scheduled tasks were completed (no artificial deadline was imposed). In order to reduce the training time, the simulation time was speeded up sixty times.

5.3 Policy training

We attempted to use a few real-world workloads from the Parallel Workloads Archive [58]. However, the best results in the training were achieved by using a set of 1551 jobs generated specifically for the purpose of our experiment. The jobs scheduling pattern resembles a single run of evolution in *pytorch-dnn-evolution*. We organized the jobs into 21 batches (10 batches of 100 and 11 batches of 50 jobs) submitted every 8 min. Every job requested 360 s on a single CPU core. The final job has been added 30 min after the final batch which ensured that there is always a cool-down period of time at the end. Such a dataset, on the one hand, was similar to the real-world workload (the jobs were submitted in multiple batches which generated *spikes* of activity). On the other hand, it differed from the real workload with the actual numbers of batches and their size. Since the number of jobs was low, the simulation time was shorter compared with other recorded workloads. We believe such a training dataset allowed to focus on the general features of the environment which is under control (e.g., the latency of the VM control mechanism, job submission spikes), while it reduced the simulation time. This in turn allowed to increase the number of simulations which enabled to obtain an improved control policy.

The training objective was defined as maximizing the following reward function:

$$F(V) = - \sum_{x \in V} (T_x \cdot C_x) - T_Q \cdot C_Q \quad (3)$$

where:

- $F(V)$ is the negative cost of resources used for processing,
- V denotes a set of possible VM sizes. In our experiments it includes S , M or L which represent *small*, *medium* or *large* VMs, accordingly,
- T_x denote the number of hours of running VMs of size x ,
- C_x is the hourly cost of running a machine of size x . In our case $C_S = \$0.2$, $C_M = \$0.4$ and $C_L = \$0.8$,
- T_Q —the hours spent by tasks waiting for execution,
- C_Q —the hourly penalty for missing SLA targets when a task is waiting for execution. The cost of 0.036 US dollars is accrued for every second of a delay between submitting task for execution and actual execution. There were no limitations on the waiting time or the waiting queue size.

The training algorithm used to create the control policy follows the *proximal policy optimization* procedure as described in Sect. 2.2.

5.4 Policy neural network model

We have experimented with different architectures of the neural network used as a decision policy. The best results have been obtained with the use of the LSTM architecture [26]. LSTM is a type of a recurrent neural network, which means it passes

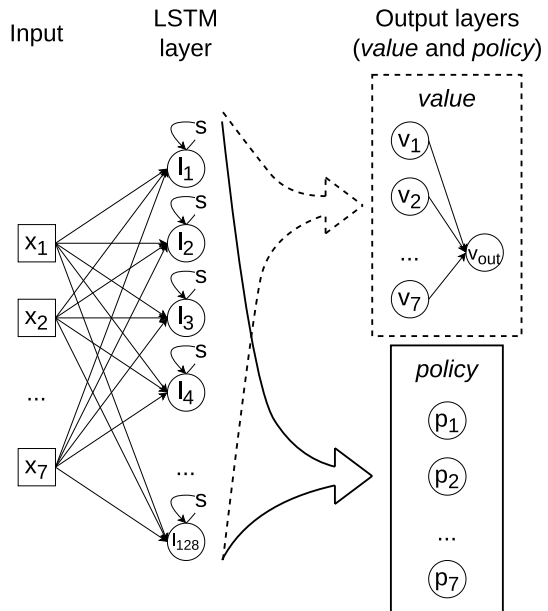
the output of a layer back to its input. This makes it well-suited to process data in form of sequences, as it has access to the previously made decisions. One large drawback of recurrent networks is that they are likely to suffer from the problem of exploding [64] or vanishing gradients [65]. While the network error is back-propagated, its value can become high, which over a number of iterations accumulates as the weight value. At some point it becomes so huge that is not possible to represent it as a number in computer memory anymore. On the other hand, the network error can become so small, that it will not be able to affect the value of the weights in a meaningful way. LSTM networks try to mitigate those issues by enhancing the internal structure of a single neuron. The additional components allow to control the flow of the information within the network, e.g., it can be multiplied by a small number (*forgotten*) while passing.

In our experiment, the policy neural network included LSTM cells and feed forward layers which allow to interpret the output of LSTM cells in different contexts (as the policy or the value function). The LSTM layer included 128 cells. The complete network architecture is presented in Fig. 8. The network contains two outputs: *value* and *policy*. The former is used in the GAE algorithm to estimate the advantage function, while the latter is used to determine the action taken by the policy.

The progress of training that model (the reward obtained in the subsequent simulations) is depicted in Figs. 9 and 10. The first chart shows how the reward evolved over the course of training and demonstrates a clearly visible tendency for growth. The second chart focuses on the length of simulation and has been smoothed by averaging the metric values over 10 subsequent simulations.

Both figures show that initially, for approximately 500 iterations, the policy was not making good decisions. This resulted in a high cost of resources within the early

Fig. 8 Neural network trained in the experiment. x_n denote the network inputs, l_n cells of the LSTM layer, s the output of a cell which is passed back to input, v_n and p_n the neurons of the *value* and *policy* outputs respectively



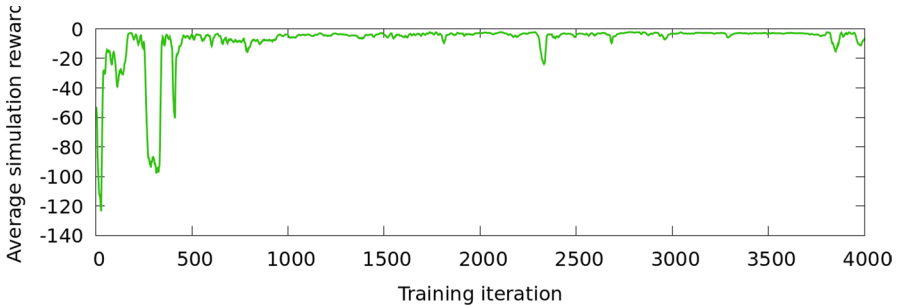


Fig. 9 Policy training progress—reward obtained during training

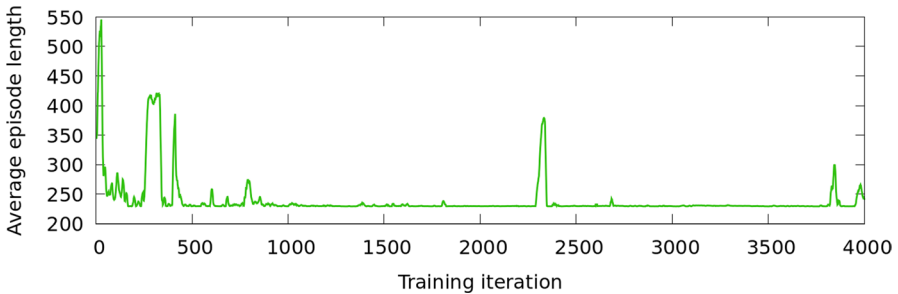


Fig. 10 Policy training progress—simulation episode length during training

simulations and a high number of simulation steps. Both factors gradually improved over time. The cost plateaued on single-digit values and simulation length at 230 iterations. Such a number of iterations is driven by the workload. It is not possible to finish the calculations earlier, because the last, very short job is scheduled at iteration 229. Since this job is short it successfully finishes in the next iteration. This means that iteration 230 is the earliest possible iteration at which the simulation can be finished.

The policy training algorithm parameter values are given in Table 1.

The implementation of the training process based on the source code of the Open AI Baselines project [66] allowed us to speed up the development time and ensure the correctness of the algorithms.

5.5 Using threshold-based policy

We have found that it is challenging to find performance reports of similar automated management systems in real (not simulated) environments. In order to provide a reference point for the results obtained with the use of the presented control policy, we also attempted to manage the *pytorch-dnnevo* workload with the use of a rule-based policy configured within the ASG. This feature provided by the cloud vendor allows to start and stop virtual machines based on the CPU usage of currently running machines. The

Table 1 Parameters of the policy training process

Parameter name	Initial value	Final value
Value function coefficient	1	0.0005
Lambda	0.95	0.97
Gamma	0.99	0.99
Training timesteps	10^6	10^7
Clipping factor	0.2	0.2
Learning rate	0.0003	0.0003
Batch size (number of steps in sequence at the input of the neural network)	2048	250
Optimization epochs	3	4
Simulator speedup	60	60

The Initial value column shows the initial parameter values. As a starting point we have used the values provided in [8]. The Final value column contains the final parameters obtained through an empirical iterative trial-and-error process

user can define a threshold which is compared periodically with the average CPU usage of all virtual machines running within the ASG. If the CPU usage is above the specified threshold a new virtual machine is started. Conversely, if the CPU usage drops below the threshold, one of the running machines is terminated.

The workload generated by the *pytorch-dnnevo* framework has its own unique characteristic. The driver machine performs only cheap, simple operations of genotype recombination, mutation or fitness comparison. The time-costly operation of individual evaluation is performed solely by the workers. Unfortunately the evaluation requests are not evenly distributed over time. They are submitted by the driver in groups whenever evaluation of the whole population is required. This means that workers' resources are fully allocated only at the beginning and are fully released after all individual evaluations are done. This means that the CPU load of a worker machine oscillates between very low (5–15%) and very high values (85–100%). Choosing a policy threshold value around the low end would force the policy to scale the number of workers up at the beginning and keep such a configuration till the end of the workload. On the other hand, choosing a threshold value from the high end would make the policy eager to remove resources, what might result in a very slow progress. A value between 20 and 80% enables the threshold policy to add (or remove) resources when they are needed (or obsolete). The exact value influences the sensitivity of the policy to the load changes. We found empirically that a threshold value of 75% average CPU usage allows to achieve the lowest resource costs when managing the sample *pytorch-dnnevo* workload.

6 Experimental results

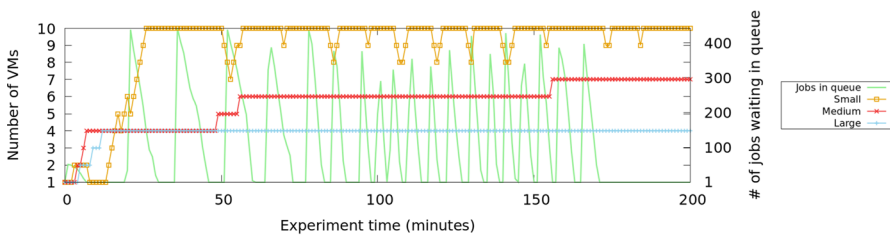
In Fig. 11 we present the course of the experiment. We show how many virtual machines of different types were active at a given point in time compared to what was the actual number of jobs waiting for processing. The shape of the charts (the

steps) is caused by an artificial delay introduced after executing an action (the cool-down period).

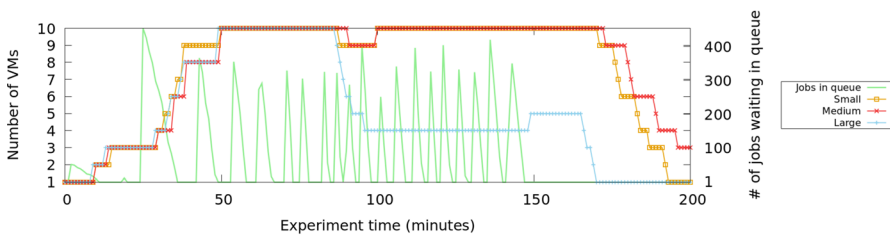
The overall results of the experiment are as follows: the experiment runtime was 173 min with the cost of resources equal to \$8.67 for the PPO-trained policy, and respectively 149 min and \$9.95 for the threshold-based approach. The PPO-trained policy had a slower execution (by 16.1%—24 min) but a lower resources cost (by 12.9%—\$1.28). The cost of the infrastructure required to manage the workload and the management components is the same in both cases (an additional VM to host other elements of the system). The main objective of the policy was to conduct the computations while minimizing the costs. In this context, the PPO-trained policy allowed to obtain a lower cost. It traded off the additional processing time for lowering the overall cost of resources.

The PPO-trained policy maintained a similar number of VMs of all types running most of the time. Occasionally it would attempt to reduce the amount of small VMs what seemed to be a result of the pauses between submitting the jobs of subsequent evolution iterations. However, those drops would get quickly compensated. The number of medium and large machines was relatively stable.

The threshold-based policy was more eager to perform scaling operations and able to launch machines of different types at the same time. As soon as the processing load was decreasing, the policy started to reduce the amount of used resources. Most of the time all resource types were treated similarly (the number of small, medium and large VMs were increased and decreased at the same time). This is caused by the fact that all VM types were scaled based on the same metric—CPU utilization.



(a) Policy trained with use of PPO



(b) Threshold-based policy

Fig. 11 Number of started VMs in context of jobs waiting in the queue

The way that the PPO-trained policy was deciding to perform actions, allowed it to achieve higher resource utilization throughout the experiment. The overall number of allocated cores remained stable after the initial increase (Fig. 12) and was lower by 25.76% on average (49.93% for the PPO-trained, and 66.24% for threshold-based policy). This helped to prevent over-provisioning visible in the case of threshold-based policy. The average CPU utilization in the former case has been equal to 0.69, while in the latter one it was equal to 0.61. This can be understood as a 13.11% improvement. Respectively, the average percentage of memory used was equal to 2.33 and 1.99, which is a 17.08% improvement. The average CPU load during the experiment (for both policies) is depicted in Fig. 13, while the average memory usage is presented in Fig. 14. Interestingly, both policies seemed to reach similar utilization values after 90 min of processing (about half of the workload executed), what suggests that they were both able to discover the infrastructure configuration which captured the near-optimal trade-off between cost and speed of processing.

We acknowledge that this might not be a fully fair comparison, e.g., it might be possible to fine tune the threshold to avoid the described initial slow-down. Alternatively, implementing a policy which could use multiple thresholds might achieve even better results. This experiment shows, however, that the use of a PPO-trained policy renders results which are on-par with a well-established approach. Using an RL-based policy has an advantage of being able to take into account multiple factors without having to specify special parameters for each of them, e.g., the thresholds.

The training process proved to be flexible and can be easily reused. To create policies for other, similar workloads, one needs to adjust few elements. First, a dataset with jobs which could be simulated in the training process, has to be created. This can be achieved in various ways: one could record sample jobs which are executed in a real environment or simply artificially create them in line with expectations about the real workload. Depending on the chosen platform and SLAs, the reward function might also need to be adjusted (e.g., by including more VM types in the V set). Finally, the monitoring of the system to be automatically controlled, might need to be extended to include metrics which are used to describe the state of that system.

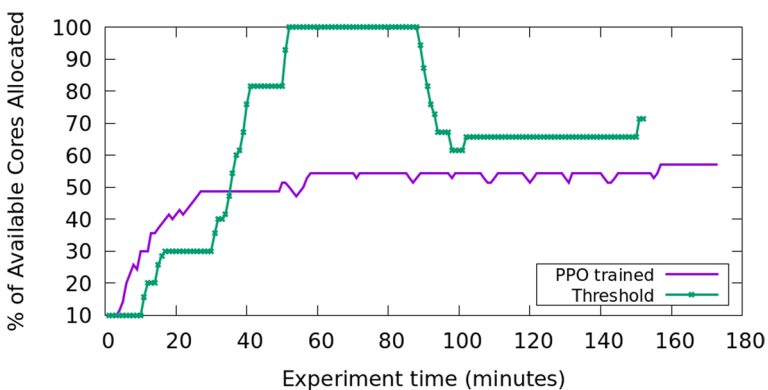


Fig. 12 Cores allocated by a control policy out of the cores available to the system throughout the experiment

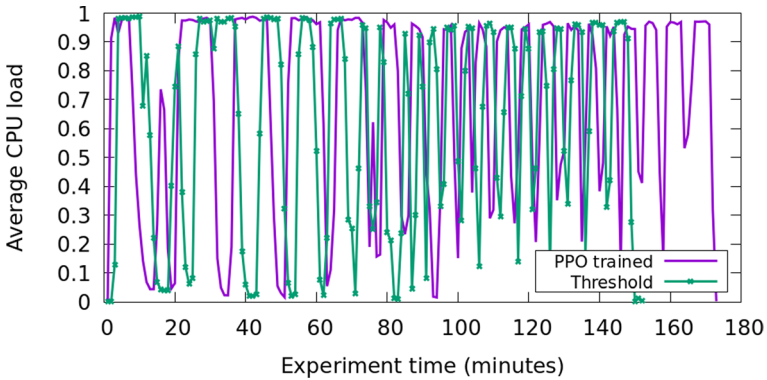


Fig. 13 Average CPU load for the cores allocated by a control policy

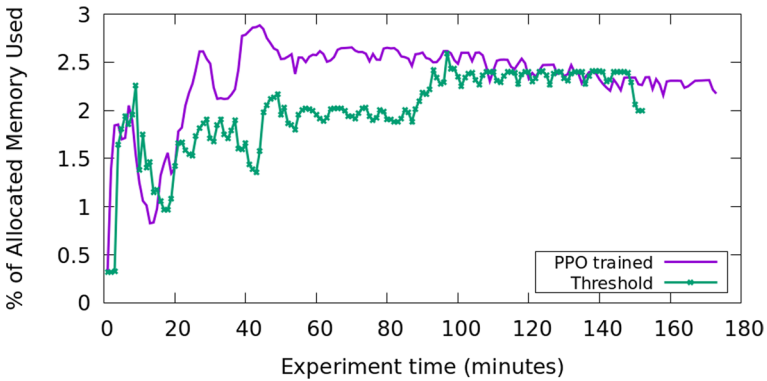


Fig. 14 Percentage of all cores available to the system allocated by the control policy

7 Conclusions and further research

In this paper we have presented a novel approach to automating the heterogeneous resource allocation. We proposed an architecture of a monitoring and management system which exploits recent advancements in the Deep Reinforcement Learning field. Through an experiment in the AWS Elastic Compute Cloud, we explained how to train a policy with the use of the PPO algorithm and deploy it to a real-world cloud infrastructure. We demonstrated that the use of such a policy can render better results compared with a traditional threshold-based one. One needs to remember that the observed cost reduction depends on many factors, e.g., on the amount of the managed resources (if that number is low, the benefits of automated scaling may not be significant). Due to the additional cost of the additional VM, the cost improvement expressed as a percentage of the initial resources spend might not be as high as reported in case of smaller infrastructures. Applying the presented approach in a scenario where more resources are being used would render better absolute results, in other words would provide bigger resource cost savings. The DRL-based

approach also had other advantages. We did not have to manually set thresholds of the policy, which may depend largely on the workload which is being managed. We can easily include other metrics at the input of the trained policy. Since the policy does not contain any hard-coded parameters, it can be reused in the context of other, similar applications.

The approach we have used to train the policy delivered a good outcome. The resulting policy could manage a sample AWS-based infrastructure, while the training time was not prohibitively long. The use of the simulator allows to run many more interactions with the resources than it would be possible in a real environment. At the same time the cost of training has been greatly reduced compared to running a copy of a production version of the managed application. It is possible to further reduce the training time by running multiple simulations in parallel. Simulations are independent of each other and rely only on CPU-based calculations, which makes it easy and relatively cheap to run multiple of them at the same time.

We have identified some issues which require further work. Our resource allocation policy was unable to react to changes in the environment fast enough. It was limited by having to wait through the resource allocation grace period after executing an action and was capable of starting or stopping only a single VM of a given type at a time. This issue could be mitigated by including actions which affect multiple instances of resources of the same type. Such a solution could also help to reach even better results in the cost optimization.

Although the training process delivered good results, it was still limited in a number of ways. The parameters of the training procedure (e.g., learning rate, λ , γ and the clipping factor in the PPO algorithm) had to be fine-tuned to our specific case. Otherwise, the process might end up with an exploding or vanishing gradient or a policy converging to a local minimum (e.g., using only a single action all the time). There is no indication how the input is being used by the training algorithm or the policy, what may lead to creating a huge and very expensive to train neural network model. It is possible that e.g., one of the metrics could be completely removed from the input because its values are mostly ignored. This might result in creating a smaller, easier to train model.

In line with our expectations, the policy was able to make good decisions only in situations, to which it was exposed in the prior training (e.g., was rather slow to shutdown the unused resources after the workload had stopped completely). Unfortunately, since the training process has been done offline (outside of the cloud environment) it may be very hard to update the policy after deployment. One might argue that a simple solution to this problem is to allow the network to be continuously trained while it is operating in the cloud environment. However, such an approach has one significant disadvantage: due to the nature of the training process, the updated policy might not make decisions as good as the current one. This means that we would be risking to introduce potentially disastrous changes into the environment, where such changes should be avoided at all cost. To mitigate this issue, the performance of the new version of the policy needs to be verified prior to the deployment to the managed environment. One way to do this is to compare it with the previous, currently deployed one, e.g., to simulate the behavior of both policies in the same environment with the same entry conditions and compare the rewards

after finishing the simulation. Another advantage of such an approach is that the decision policy becomes closer aligned to the environment it controls. New information is constantly being added to the representation of the policy (e.g., in the case of DNN—to the neural network weights).

Acknowledgements The research presented in this paper was supported by the funds assigned to AGH University of Science and Technology by the Polish Ministry of Education and Science. The experiments have been carried out on the PL-Grid infrastructure resources of ACC Cyfronet AGH and on the Amazon Web Services Elastic Compute Cloud.

Data availability The datasets used, generated and analyzed during the current study are available in publicly accessible repository [58] or can be provided from the corresponding author on a reasonable request.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Chen T, Bahsoon R, Yao X (2018) A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput Surv* 51(3):61–16140
2. Sutton RS (1984) Temporal credit assignment in reinforcement learning. PhD thesis, University of Massachusetts Amherst
3. Kaelbling LP, Littman ML, Moore AW (1996) Reinforcement learning: a survey. *CoRR* [arXiv:cs.AI/9605103](https://arxiv.org/abs/cs.AI/9605103)
4. Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, Petersen S, Beattie C, Sadik A, Antonoglou I, King H, Kumaran D, Wierstra D, Legg S, Hassabis D (2015) Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533
5. Gu S, Holly E, Lillicrap T, Levine S (2017) Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 3389–3396. IEEE International Conference on Robotics and Automation (ICRA), Washington, DC, USA. <https://doi.org/10.1109/ICRA.2017.7989385>
6. Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A, Chen Y, Lillicrap TP, Hui F, Sifre L, van den Driessche G, Graepel T, Hassabis D (2017) Mastering the game of go without human knowledge. *Nature* 550(7676):354–359. <https://doi.org/10.1038/nature24270>
7. Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M (2013) Playing atari with deep reinforcement learning. NIPS Deep Learning Workshop 2013. [arXiv:1312.5602](https://arxiv.org/abs/1312.5602)
8. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal policy optimization algorithms. *CoRR* [arXiv:abs/1707.06347](https://arxiv.org/abs/1707.06347)
9. Cobbe K, Hilton J, Klimov O, Schulman J (2020) Phasic policy gradient. *CoRR* [arXiv:abs/2009.04416](https://arxiv.org/abs/2009.04416)

10. OpenAI, Berner C, Brockman G, Chan B, Cheung V, Dębiak P, Dennison C, Farhi D, Fischer Q, Hashme S, Hesse C, Józefowicz R, Gray S, Olsson C, Pachocki J, Petrov M, Pinto HPdO, Raiman J, Salimans T, Schlatter J, Schneider J, Sidor S, Sutskever I, Tang J, Wolski F, Zhang S (2019) Dota 2 with Large Scale Deep Reinforcement Learning. [arXiv:1912.06680](https://arxiv.org/abs/1912.06680). <https://doi.org/10.48550/ARXIV.1912.06680>
11. Heess N, TB D, Sriram S, Lemmon J, Merel J, Wayne G, Tassa Y, Erez T, Wang Z, Eslami SMA, Riedmiller M, Silver D (2017) Emergence of Locomotion Behaviours in Rich Environments. [arXiv:1707.02286](https://arxiv.org/abs/1707.02286). <https://doi.org/10.48550/ARXIV.1707.02286>
12. Schulman J, Moritz P, Levine S, Jordan M, Abbeel P (2015) High-dimensional continuous control using generalized advantage estimation. [arXiv:1506.02438](https://arxiv.org/abs/1506.02438). <https://doi.org/10.48550/ARXIV.1506.02438>
13. OpenAI Akkaya I, Andrychowicz M, Chociej M, Litwin M, McGrew B, Petron A, Paino A, Plappert M, Powell G, Ribas R, Schneider J, Tezak N, Tworek J, Welinder P, Weng L, Yuan Q, Zaremba W, Zhang L (2019) Solving Rubik's Cube with a Robot Hand. [https://doi.org/10.48550/ARXIV.1910.07113](https://arxiv.org/abs/1910.07113). [arXiv:1910.07113](https://arxiv.org/abs/1910.07113)
14. Mnih V, Badia AP, Mirza M, Graves A, Harley T, Lillicrap TP, Silver D, Kavukcuoglu K (2016) Asynchronous methods for deep reinforcement learning. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. ICML'16, pp. 1928–1937. JMLR.org, New York, NY, USA
15. Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M (2014) Deterministic policy gradient algorithms. In: Xing EP, Jebara T (eds) Proceedings of the 31st International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 32, pp 387–395. PMLR, Beijing, China. <http://proceedings.mlr.press/v32/silver14.html>
16. Haarnoja T, Zhou A, Abbeel P, Levine S (2018) Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: Dy J, Krause A (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 1861–1870. <https://proceedings.mlr.press/v80/haarnoja18b.html>
17. Cheng M, Li J, Nazarian S (2018) Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In: 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 129–134. <https://doi.org/10.1109/ASPDAC.2018.8297294>
18. Wang Z, Gwon C, Oates T, Iezzi A (2017) Automated cloud provisioning on AWS using deep reinforcement learning. CoRR [arXiv:abs/1709.04305](https://arxiv.org/abs/1709.04305)
19. Pereira dos Santos, José Pedro and Wauters, Tim and Volckaert, Bruno and De Turck (2021) Filip: Resource provisioning in fog computing through deep reinforcement learning. In: 2021 IFIP/IEEE International Symposium on Integrated Network and Service Management, Proceedings. 2021 IFIP/IEEE International Symposium on Integrated Network and Service Management, Proceedings, p. 7. <https://im2021.ieee-im.org/>
20. Liang S, Yang Z, Jin F, Chen Y (2020) Data centers job scheduling with deep reinforcement learning. In: Lauw HW, Wong RC-W, Ntoulas A, Lim E-P, Ng S-K, Pan SJ (eds) Advances in Knowledge Discovery and Data Mining. Springer, Cham, pp 906–917
21. Islam MT, Karunasekera S, Buyya R (2022) Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. *IEEE Trans Parallel Distrib Syst* 33(7):1695–1710. <https://doi.org/10.1109/TPDS.2021.3124670>
22. Chen L, Lingys J, Chen K, Liu F (2018) Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pp 191–205. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3230543.3230551>
23. Funika W, Koperek P (2020) Evaluating the use of policy gradient optimization approach for automatic cloud resource provisioning. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K (eds) Parallel Processing and Applied Mathematics. LNCS 12043, pp 467–478. Springer, Cham
24. Funika W, Kupisz M, Koperek P (2010) Towards autonomic semantic-based management of distributed applications. *Comput Sci* 11:51–64
25. Funika W, Koperek P, Kitowski J (2020) Automatic management of cloud applications with use of proximal policy optimization. In: Computational Science - ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part I, pp. 73–87. Springer, Berlin. https://doi.org/10.1007/978-3-030-50371-0_6

26. Hochreiter S, Schmidhuber J (1997) Long short-term memory. *Neural Comput* 9(8):1735–1780
27. Brun Y, Di Marzo Serugendo G, Gacek C, Giese H, Kienle H, Litoiu M, Müller H, Pezzè M, Shaw M (2009) Engineering self-adaptive systems through feedback loops. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) *Software Engineering for Self-Adaptive Systems*, pp. 48–70. Springer, Berlin. https://doi.org/10.1007/978-3-642-02161-9_3
28. Hoffman H (2013) Seec: A framework for self-aware management of goals and constraints in computing systems (power-aware computing, accuracy-aware computing, adaptive computing, autonomic computing). PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA
29. IBM (2005) An Architectural Blueprint for Autonomic Computing. Technical report
30. Huber N, Brosig F, Kounev S (2011) Model-based self-adaptive resource allocation in virtualized environments. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '11*, pp 90–99. ACM, New York
31. Minarolli D, Freisleben B (2014) Distributed resource allocation to virtual machines via artificial neural networks. In: *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. PDP '14*, pp 490–499. IEEE Computer Society, Washington, DC, USA
32. Wickremasinghe B, Calheiros RN, Buyya R (2010) Cloudbanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp 446–452. 24th IEEE International Conference on Advanced Information Networking and Applications, Washington, DC, USA
33. Kim S, Kim J-S, Hwang S, Kim Y (2013) An allocation and provisioning model of science cloud for high throughput computing applications. In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. CAC '13*. ACM, New York, pp 27–1278
34. Qu C, Calheiros RN, Buyya R (2015) A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. CoRR [arXiv:abs/1509.05197](https://arxiv.org/abs/1509.05197)
35. Rodriguez MA, Buyya R (2018) Containers orchestration with cost-efficient autoscaling in cloud computing environments. CoRR [arXiv:abs/1812.00300](https://arxiv.org/abs/1812.00300)
36. Fernandez H, Pierre G, Kielmann T (2014) Autoscaling web applications in heterogeneous cloud infrastructures. In: *Proceedings of the 2014 IEEE International Conference on Cloud Engineering. IC2E '14*. IEEE Computer Society, Washington, DC, USA, pp 195–204
37. Koperek P, Funika W (2012) Dynamic business metrics-driven resource provisioning in cloud environments. In: Wyrzykowski R, Dongarra J, Karczewski K, Waśniewski J (eds) *Parallel Processing and Applied Mathematics. LNCS 7204*. Springer, Berlin, pp 171–180
38. Ferretti S, Ghini V, Panziri F, Pellegrini M, Turrini E (2010) Qos-aware clouds. In: *2010 IEEE 3rd International Conference on Cloud Computing*, pp 321–328
39. Ashraf A, Byholm B, Porres I (2012) Cramp: cost-efficient resource allocation for multiple web applications with proactive scaling. In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp 581–586
40. Xu C-Z, Rao J, Bu X (2012) URL: a unified reinforcement learning approach for autonomic cloud management. *J Parallel Distrib Comput* 72(2):95–105. <https://doi.org/10.1016/j.jpdc.2011.10.003>
41. Xiong P, Chi Y, Zhu S, Moon H, Pu C, Hacigumus H (2014) Smartsla: cost-sensitive management of virtualized resources for cpu-bound database services. *IEEE Trans Parallel Distrib Syst* 26:1441–1451
42. Venticinque S, Nacchia S, Maisto SA (2020) Reinforcement learning for resource allocation in cloud datacenter. In: Barolli L, Hellinckx P, Natwichai J (eds) *Advances on P2P, parallel, grid, cloud and internet computing*. Springer, Cham, pp 648–657
43. Ding D, Fan X, Zhao Y, Kang K, Yin Q, Zeng J (2020) Q-learning based dynamic task scheduling for energy-efficient cloud computing. *Future Gener Comput Syst* 108:361–371. <https://doi.org/10.1016/j.future.2020.02.018>
44. Kitowski J, Mościński J (1979) Computer simulation of heuristic reinforcement learning system for nuclear plant load changes control. *Comput Phys Commun* 18:339–352
45. Kingma DP, Ba J (2017) Adam: a method for stochastic optimization
46. Guo W, Tian W, Ye Y, Xu L, Wu K (2021) Cloud resource scheduling with deep reinforcement learning and imitation learning. *IEEE Internet Things J* 8(5):3576–3586. <https://doi.org/10.1109/JIOT.2020.3025015>

47. Zhang Y, Yao J, Guan H (2017) Intelligent cloud resource management with deep reinforcement learning. *IEEE Cloud Comput* 4(6):60–69. <https://doi.org/10.1109/MCC.2018.1081063>
48. Liu N, Li Z, Xu Z, Xu J, Lin S, Qiu Q, Tang J, Wang Y (2017) A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. *CoRR arXiv:abs/1703.04221*
49. Li M, Yu FR, Si P, Wu W, Zhang Y (2020) Resource optimization for delay-tolerant data in blockchain-enabled iot with edge computing: a deep reinforcement learning approach. *IEEE Internet Things J* 7(10):9399–9412. <https://doi.org/10.1109/JIOT.2020.3007869>
50. Bitsakos C, Konstantinou I, Koziris N (2018) Derp: a deep reinforcement learning cloud system for elastic resource provisioning. In: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp 21–29. <https://doi.org/10.1109/CloudCom2018.2018.00020>
51. Shan N, Cui X, Gao Z (2020) “drl + fl”: An intelligent resource allocation model based on deep reinforcement learning for mobile edge computing. *Comput Commun* 160:14–24. <https://doi.org/10.1016/j.comcom.2020.05.037>
52. Kardani-Moghaddam S, Buyya R, Ramamohanarao K (2021) Adrl: a hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Trans Parallel Distrib Syst* 32(3):514–526. <https://doi.org/10.1109/TPDS.2020.3025914>
53. Hummida A, Paton N, Sakellariou R (2022) Scalable virtual machine migration using reinforcement learning. *J Grid Comput* 20. <https://doi.org/10.1007/s10723-022-09603-4>
54. John I, Bhatnagar S (2020) Deep reinforcement learning with successive over-relaxation and its application in autoscaling cloud resources. In: 2020 International Joint Conference on Neural Networks (IJCNN), pp 1–6. <https://doi.org/10.1109/IJCNN48605.2020.9206598>
55. Grandl R, Ananthanarayanan G, Kandula S, Rao S, Akella A (2014) Multi-resource packing for cluster schedulers. In: Proceedings of the 2014 ACM Conference on SIGCOMM. SIGCOMM '14. Association for Computing Machinery, New York, pp 455–466. <https://doi.org/10.1145/2619239.2626334>
56. Rzaşa W (2017) Predicting performance in a paas environment: a case study for a web application. *Comput Sci* 18(1):21
57. Graphite Project (2011) <https://graphiteapp.org/>. Accessed 15 Feb 2022
58. Feitelson D (2005) Parallel Workloads Archive. <https://www.cs.huji.ac.il/labs/parallel/workload/>. Accessed 29 Oct 2021
59. Filho MCS, Oliveira RL, Monteiro CC, Inácio PRM, Freire MM (2017) Cloudsim plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management. IFIP/IEEE Symposium on Integrated Network and Service Management, Washington, DC, USA, pp 400–406
60. Brockman G, et al (2016) OpenAI Gym. [arxiv:1606.01540](https://arxiv.org/abs/1606.01540)
61. PyTorch DNN Evolution (2018) <https://gitlab.com/pkoperek/pytorch-dnn-evolution>. Accessed 02 March 2022
62. LeCun Y, Cortes C (2010) MNIST handwritten digit database. <http://yann.lecun.com/exdb/mnist/>
63. Amazon Web Services Elastic Compute Cloud (2020) <https://aws.amazon.com/ec2/>. Accessed 02 March 2022
64. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. *IEEE Trans Neural Netw* 5(2):157–166. <https://doi.org/10.1109/72.279181>
65. Hochreiter S (1998) The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int J Uncertain Fuzziness Knowl-Based Syst* 6:107–116. <https://doi.org/10.1142/S0218488598000094>
66. Dhariwal P, Hesse C, Klimov O, Nichol A, Plappert M, Radford A, Schulman J, Sidor S, Wu Y, Zhokhov P (2017) OpenAI Baselines. GitHub

Authors and Affiliations

Włodzimierz Funika¹ · Paweł Koperek¹  · Jacek Kitowski^{1,2}

Włodzimierz Funika
funika@agh.edu.pl

Jacek Kitowski
kito@agh.edu.pl

- ¹ Institute of Computer Science, Faculty of Computer Science, Electronics and Telecommunication, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
- ² ACC CYFRONET AGH, AGH, ul. Nawojki 11, 30-950 Kraków, Poland