Management of Heterogeneous Cloud Resources with Use of the PPO

Włodzimierz Funika\textsuperscript{1}(✉), Paweł Koperek\textsuperscript{1}, and Jacek Kitowski\textsuperscript{1,2}

\textsuperscript{1} Faculty of Computer Science, Electronics and Telecommunication, Department of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland \{funika, kito\}@agh.edu.pl, pkoperek@gmail.com

\textsuperscript{2} AGH, ACC CYFRONET AGH, ul. Nawojki 11, 30-950 Kraków, Poland

Abstract. Reinforcement learning 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 this paper we present the results of our attempt to use the recent advancements in Reinforcement Learning to automate the management of heterogeneous resources in an environment which hosts a compute-intensive evolutionary process. 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 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 other scenarios.

Keywords: Reinforcement learning · Heterogeneous cloud resources · Automatic management · Proximal policy optimization

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 [...] that adapts software configurations [...] and hardware resources provisioning [...] on-demand, according to the time-varying environmental conditions [6].
The area of the Reinforcement Learning (RL) techniques has been explored for a long time [12,22]. Initially the techniques and algorithms from this category could be only used in relatively simple problems. Handling more complex domains became possible with recent advancements in, e.g., computer games [18], robot control [11] or the game of Go [21]. One of the main drivers of progress has been the application of Deep Learning in various forms: Deep Q Learning [17], Asynchronous Actor-Critic Agents (A3C) [19], Proximal Policy Optimization [20]. 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 the same as the one the agent is going to operate in. Such an approach allowed to achieve results which have surpassed the performance of humans.

Such successes suggest that applying Deep Reinforcement Learning (DRL) in other domains can also render good results. One such area is the automatic scaling of distributed applications deployed to heterogeneous cloud resources. The cloud infrastructure becomes the environment where an automatic agent operates, its 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 are well defined. The goals of the system are also clear (e.g. reducing 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 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.

In our previous work [10] we have demonstrated how to leverage the described ideas to create a system capable of automatic scaling of homogeneous cloud infrastructure hosting a CPU-intensive workload. In this paper we extend this approach to heterogeneous cloud resources: the system can adjust not only the amount of resources but can also decide on the features of the added resources. The training does not require providing any additional information about the managed system or specifying resources capabilities. All decisions are derived from the experience gained from simulations. The system has been implemented as an extension to Semantic-Based Automatic Monitoring and Management (SAMM) monitoring software [9].

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

2 Related Work

Minimizing the monetary cost of cloud resources while maintaining 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 that can be used depending on the conditions of the environment which should be managed. The most distinctive approaches include: rule-based control [7,14] (action execution occurs when a condition defined a priori is met, search based optimization [16,24] (decisions form a large, finite search space and choosing among them is treated as a search problem), control theory-based [4] (control theory mechanisms are used to make a decision).

There have been a number of attempts to apply Reinforcement Learning techniques, which can be classified as search based optimization. In [23] authors explore applying variants of the Q-learning algorithm to provisioning cloud resources. They focus on a horizontally scaling infrastructure used to handle a stream of requests defined in a benchmark dataset. They demonstrate that a policy can be first trained using a simulator and then applied to a real cloud environment. In [6] 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 first one consists of two fully-connected hidden layers with 600 neurons each. The second one reuses those and adds an additional layer on top. Such a model is used to demonstrate the performance and adaptiveness of the discussed approach to the control of dynamic traffic. In [10] we demonstrated how a similar algorithm, the Proximal Policy Optimization (PPO) [20], can be used to horizontally scale cloud resources. The implementation has been limited to control resources of a single type.

Reinforcement Learning

One of the more active areas of research in machine learning is the Reinforcement Learning (RL) [13,22]. 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. The knowledge of the agent is built from observing the consequences of its actions. 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:

- 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 (DRL). 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 [17]). Alternatively, in case of policy gradient methods, neural networks can be used directly as the policy functions. The training process adjusts their weights (\(\Theta\)) based on the gradient of an estimated scalar performance objective function \(J(\Theta)\) in respect to those policy parameters:

\[
\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} J(\theta_k)
\]  

where \(\theta_k\) denotes policy’s parameters in the \(k\)-th iteration of the training process. The performance is usually understood as the reward returned from environment. There are multiple versions of policy gradient optimization. In our research we focus on the Proximal Policy Optimization (PPO) [20].

The aim of the algorithm is to calculate the parameter update in such a way, that it ensures that the difference to the previous version of the policy is relatively small. This goal is achieved through modification of the objective function. It is defined as follows:

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

where \(\mathbb{E}_t\) denotes calculating 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 state. \(r_t\) marks 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 \(a_t\) in state \(s_t\) by a stochastic policy and \(\Theta_{old}\) are the policy parameters before the update. The \(\text{clip}(r_t(\Theta), 1 - \epsilon, 1 + \epsilon)\) function keeps the value of \(r_t(\Theta)\) within some specified limits (clips it at the end of the range) and \(\epsilon\) is a hyperparameter with a typical value between 0.1 and 0.3.

In our previous research [10] we experimented with a number of policy gradient methods (Vanilla Policy Gradient, Proximal Policy Optimization, Trust-Region Policy Optimization) out of which the PPO rendered the best empirical results in the automated resources management.

3 Architecture

From a high-level perspective, the system under discussion creates a feedback loop in which the policy interacts with the environment under management. Its complete architecture is presented in Fig. 1.
The loop starts with collecting measurements about the resources which take part in executing the workload. Each of them is configured to start reporting relevant measurements as soon as it becomes online. The measurements often differ in their nature what influences how often their values are provided, 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 [2]. Graphite aggregates all the collected values into a single interval to create a consistent snapshot of the environment. This interval in our case is set to one minute.

Next, the measurements are collected by the SAMM experimental monitoring and management system [9]. SAMM enables experimenting with new approaches to management automation. It allows to easily add support for new types of resources, integrate new algorithms and technologies and observe their impact on the observed system. In our case, after retrieving raw data points, SAMM 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. They are being used to describe the current state of the cloud environment and are further passed to the Policy Evaluation Service.

After a decision is taken, SAMM uses cloud vendor’s API to implement it. It takes into the account the constraints of the environment (e.g. resource changes are adjusted to adhere to the warm-up and cool-down restrictions).

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 small, medium or large VM (deficient resources are used to handle the workload given the current system state), removing resources - shutting down a small, medium, large VM (excessive resources is used given the current state of the system), doing nothing (a proper amount of resources is allocated).
One should remember that not always it is possible to immediately execute an action. This process is always subject to environment constraints. 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, those 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:

- processing is organized into many independent tasks,
- the number of tasks which are yet to be executed can be monitored,
- the tasks which have been interrupted before finishing (e.g. in case the processing VMs are shutdown) are rescheduled,
- the tasks are considered idempotent, i.e. executing them multiple times does not change the end result,
- resources used to generate the workload are not under automatic management (prevents from accidental termination of the workload).

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

4 Policy Training

One of the main challenges in the design of an autonomous management system is organization of the training process. Using an environment with real cloud resources would be the best solution. Unfortunately, with this approach the cost of creating a DRL policy becomes a major 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 the overall cost of running the system. To avoid this issue, we decided to use a simulation as an isolated, safe training environment. Regardless of the decisions made, their consequences are not applied to any production system. This allows experimenting even with the actions that may lead to catastrophic events. Using a simulator allows to significantly cut the computational monetary cost of training resources compared with the actual system. Since a simulation is isolated, the process can be replicated and parallelized to allow for evaluation of multiple agents at the same time. The flow of time in a simulation can be changed what allows to reduce the time required to conduct training. The behavior of the environment and the workload are fully deterministic and can be easily repeated if needed.
The policy training process has been implemented using a separate environment depicted in Fig. 2.

![Diagram of the training system](image)

**Fig. 2.** Components of the training system; arrows denote interactions between them.

The simulator has been implemented following the results of our prior research [10]. The main process utilizes the CloudSim Plus simulation framework [8]. To decouple it from other components and allow for easy reuse, it is additionally wrapped with the interface provided by the Open AI Gym framework [5]. This helps to easily launch experiments with various RL algorithms independently of the presented architecture.

We simulated a single datacenter capable of hosting Virtual Machines (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 1 virtual machine of each type active and run until all the scheduled tasks were completed (there was an artificial deadline). We attempted to use a few different workloads. The best results in training were achieved by using a set of 1551 jobs generated specifically for the purpose of our experiment. The jobs were organized 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 what ensured that there is always a cool-down period of time at the end. We considered such workload typical in our sample environment.

The training objective was defined as maximizing the following reward function: $F(T_S, T_M, T_L, T_Q) = -(T_S \cdot 0.2 + T_M \cdot 0.4 + T_L \cdot 0.8 + 0.036 \cdot T_Q)$ which was the negative cost of resources used for processing. $T_S$, $T_M$, $T_L$ denote the number of hours of running small, medium or large VMs (with an hourly cost respectively $0.2$, $0.4$ and $0.8$). The reward included paying penalties for missing SLA targets, $0.036$ for each of $T_Q$ hours spent by tasks waiting for execution. Waiting time or waiting queue size was not limited. In order to reduce the training time, the simulation time was speeded up sixty times. The training algorithm followed the Proximal Policy Optimization procedure described in Sect. 2. The progress of training (reward obtained in the subsequent simulations) is depicted in Fig. 3.
5 Experiment Design

In order to evaluate our approach, we designed an experiment in which we wanted to compare our policy to another algorithm. The overall objective was to perform sample computations while limiting the cost of the used cloud resources.

As a sample workload, we have used the `pytorch-dnn-evolution` tool [3]. This is a tool which attempts to discover an optimal structure of a Deep Neural Network (DNN) to solve a given problem (e.g. 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: the first one which attempts to find the hardest to solve small subset of the large training dataset, the second one which evolves the DNNs and uses the small subset in the individual evaluation. Reducing the amount of data required to conduct the evaluation allows to greatly speed up the comparison between individuals and enables using the evolutionary approach.

This algorithm produces a high number of relatively small tasks. They are independent from each other and can be easily processed in parallel on a cluster of machines. 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 small size of a single job description. 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 processing the evaluation tasks without additional configuration. The number of tasks varies over time, which allows to potentially reduce the cost of running the evolutionary process by reducing the amount of the used resources (VMs) when the demand for resources drops. In our case, the evolutionary process tries to find an optimal architecture of neural network which recognizes hand written 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. The MNIST dataset [15] has been used as the training set from which subsets are selected.

As a compute infrastructure we have used the Amazon Web Services Elastic Compute Cloud (AWS EC2) [1]. The managed environment consisted of three Auto Scaling Groups 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, have been running on a separate VM.

To provide a reference point for the results obtained with the use of the presented policy, we also attempted to manage the pytorch-dnnevo workload with the use of a rule-based policy configured within the Auto Scaling Group. This cloud vendor feature starts and stops virtual machines based on the CPU usage or currently running machines. A new machine is started whenever the metric value is higher than a pre-defined threshold. If the value drops below the threshold, one of the running machines is terminated. We found empirically that a threshold value of 75% average CPU usage renders the best results.

6 Experiment Results

In Fig. 4 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 first policy had a slower execution (by 16.1% - 24 min) but a lower resources cost (by 12.9% - $1.28). The cost of the additional infrastructure 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 costs. In that context, the PPO-trained policy rendered better results. It traded additional processing time for lowering the overall cost or 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 pauses between submitting 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 introduce changes and was able to launch machines of different types in the same time. As soon as the processing load was decreasing, it started to reduce the amount of used resources. It seemed that most of the time all resource types were treated similarly (the number of small, medium and large VMs was increased and decreased in the same time).

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, using multiple rules and 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 a RL-based policy has an advantage of being able to take into account multiple factors without having to specify some special parameters for each of them, e.g., the thresholds. The training process was flexible and can be easily reused to create policies for other, similar workloads.

7 Conclusions and Further Research

In this paper we have presented a novel approach to automating heterogeneous resource allocation. We proposed an architecture of a monitoring system which exploits recent advancements in the Deep Reinforcement Learning field. Through an experiment the AWS Elastic Compute Cloud, we explained how to train a policy with use of the PPO algorithm and deploy it to a real-world cloud infrastructure. We demonstrated that using such a policy can render better results comparing with a traditional threshold-based one. However, depending on the amount of the managed resources due to the additional cost of the additional VM, the overall improvement might be reduced. The DRL based approach also had other advantages (no manually set thresholds, easy including multiple decision factors, easy reuse in context of similar applications).

The approach we have used to train the policy delivered good results. The resulting policy could manage a sample AWS-based infrastructure. Using a simulator allowed to run many more interactions with a simulated environment than
it would be possible in a real environment. In the same time the cost of training has been greatly reduced comparing to running a copy of a production version of the managed application.

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 grace period and was capable of starting or stopping only a single VM at a time. 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 very slow to shutdown the unused resources after the workload stopped completely).

We plan to continue the work on extending the described approach. Further work includes adding a policy improvement loop which would allow to dynamically adjust the policy to a changing workload and would remove the requirement of training the policy prior to the deployment. We also aim to extend the range of available actions to enable adding or removing more resources at once.

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 Science and Higher Education. 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.

References

1. Amazon Web Services Elastic Compute Cloud. https://aws.amazon.com/ec2/. Accessed 30 Dec 2019
2. Graphite Project. https://graphiteapp.org/. Accessed 28 Nov 2019
3. PyTorch DNN Evolution. https://gitlab.com/pkoperek/pytorch-dnn-evolution. Accessed 01 Dec 2019
4. Ashraf, A., et al.: 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, December 2012
5. Brockman, G., et al.: OpenAI Gym (2016). http://arxiv.org/abs/1606.01540
6. Chen, L., et al.: AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In: SIGCOMM 2018, New York, USA, pp. 191–205 (2018)
7. Ferretti, S., et al.: Qos-aware clouds. In: 2010 IEEE 3rd International Conference on Cloud Computing, pp. 321–328, July 2010
8. Filho, M.C.S., et al.: Cloudsim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness. In: IFIP/IEEE Symposium on Integrated Network and Service Management, pp. 400–406, May 2017
9. Funika, W., et al.: Towards autonomic semantic-based management of distributed applications. Comput. Sci. 11, 51–64 (2010)
10. Funika, W., Koperek, P.: Evaluating the use of policy gradient optimization approach for automatic cloud resource provisioning. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K. (eds.) PPAM 2019. LNCS, vol. 12043, pp. 467–478. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43229-4_40
11. Gu, S., et al.: Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: Proceedings 2017 IEEE International Conference on Robotics and Automation (ICRA), Piscataway, NJ, USA. IEEE, May 2017
12. Kaelbling, L.P., et al.: Reinforcement learning: a survey. CoRR cs.AI/9605103 (1996). http://arxiv.org/abs/cs.AI/9605103
13. Kitowski, J., et al.: Computer simulation of heuristic reinforcement learning system for nuclear plant load changes control. Comput. Phys. Commun. 18, 339–352 (1979)
14. Koperek, P., Funika, W.: Dynamic business metrics-driven resource provisioning in cloud environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2011. LNCS, vol. 7204, pp. 171–180. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31500-8_18
15. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010). http://yann.lecun.com/exdb/mnist/
16. Minarolli, D., Freisleben, B.: Distributed resource allocation to virtual machines via artificial neural networks. In: 2014 Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, pp. 490–499. IEEE Computer Society, Washington, DC (2014)
17. Mnih, V., et al.: Playing Atari with deep reinforcement learning (2013). http://arxiv.org/abs/1312.5602
18. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (2015)
19. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: Balcan, M.F., Weinberger, K.Q. (eds.) Proceedings of the 33rd International Conference on Machine Learning, vol. 48, pp. 1928–1937. PMLR, 20–22 June 2016
20. Schulman, J., et al.: Proximal policy optimization algorithms. CoRR abs/1707.06347 (2017). http://arxiv.org/abs/1707.06347
21. Silver, D., et al.: Mastering the game of go without human knowledge. Nature 550, 354–359 (2017)
22. Sutton, R.S.: Temporal credit assignment in reinforcement learning. Ph.D. thesis (1984)
23. Wang, Z., et al.: Automated cloud provisioning on AWS using deep reinforcement learning. CoRR abs/1709.04305 (2017). http://arxiv.org/abs/1709.04305
24. Xiong, P., et al.: SmartSLA: cost-sensitive management of virtualized resources for CPU-bound database services. IEEE Trans. Parallel Distrib. Syst. 26, 1441–1451 (2014)