Continuous-Action Reinforcement Learning for Memory Allocation in Virtualized Servers

Luis A. Garrido[1], Rajiv Nishtala[2], and Paul Carpenter[1]

[1] Barcelona Supercomputing Center
{luis.garrido, paul.carpenter}@bsc.es
[2] Norwegian University of Science and Technology
rajiv.nishtala@ntnu.no

Abstract. In a virtualized computing server (node) with multiple Virtual Machines (VMs), it is necessary to dynamically allocate memory among the VMs. In many cases, this is done only considering the memory demand of each VM without having a node-wide view. There are many solutions for the dynamic memory allocation problem, some of which use machine learning in some form.
This paper introduces CAVMem (Continuous-Action Algorithm for Virtualized Memory Management), a proof-of-concept mechanism for a decentralized dynamic memory allocation solution in virtualized nodes that applies a *continuous-action* reinforcement learning. (RL) algorithm called Deep Deterministic Policy Gradient (DDPG). CAVMem with DDPG is compared with other reinforcement learning algorithms such as Q-Learning (QL) and Deep Q-Learning (DQL) in an environment that models a virtualized node.
In order to obtain linear scaling and be able to dynamically add and remove VMs, CAVMem has one agent per VM connected via a lightweight coordination mechanism. The agents learn how much memory to bid for or return, in a given state, so that each VM obtains a fair level of performance subject to the available memory resources. Our results show that CAVMem with DDPG performs better than QL and a static allocation case, but it is competitive with DQL. However, CAVMem incurs significant less training overheads than DQL, making the continuous-action approach a more cost-effective solution.

Keywords: reinforcement learning · memory · virtualization.

1 Introduction

Cloud infrastructures are built using virtualization technology, which provides isolation for concurrently running applications [5] and allows sharing of the available computing resources [5, 6, 3, 4]. When multiple VMs are active in a node, its physical memory is allocated among the VMs in order to optimize throughput and prevent memory starvation. This memory allocation problem is difficult to solve because the memory demand of a VM changes continuously. Many solutions for this problem are limited by the complex relationship between the

memory allocated to a VM, the applications' behavior and their performance, for which RL provides a good alternative.

Different approaches of RL have been applied to the resource management problem in virtualized nodes [10] which rely on the discretization of states and actions. Discretization introduces some limitations, mainly combinatorial explosion: as the number of states/action grows, the problem becomes unsolvable [18]. In the context of memory allocation, discretization restricts the granularity at which memory can be allocated, limiting the opportunities for better optimization. Another limitation of current RL approaches for resource management is that they deploy a single agent responsible for re-allocating memory. Such a centralized approach has scalability and flexibility restrictions, since it introduces a traffic bottleneck and single points of failure.

This paper presents CAVMem (Continous-Action Algorithm for Virtualized Memory Management), which serves as a proof-of-concept for a solution to the dynamic memory allocation problem using a distributed continuous-action RL formulation, avoiding the limitations of discretization and centralization. To the best of our knowledge, this is the first such formulation of the memory allocation problem. CAVMem is initially designed with DDPG, but other RL algorithms were also implemented, namely Q-Learning [13] (QL) and Deep Q-Learning [7] (DQL). These are compared in a model environment that simulates certain aspects of a virtualized computing node.

In summary, the contributions of this paper are:
(1) Formulation of the memory management problem as a distributed continuous-action Markov Decision Process (MDP). This formulation supports an unlimited and variable number of VMs.
(2) Development of a continuous-action off-policy model-free RL algorithm for dynamic memory allocation.
(3) Comparison between three RL approaches: a) CAVMem with DDPG (continuous action space), b) CAVMem with QL (tabular, non-continuous action) and c) CAVMem with DQL (non-continuous action). We also compare against the static policy that divides memory equally among VMs.

The rest of this paper is organized as follows. Section 2 provides background information on memory allocation and RL. Section 3 explains the design of CAVMem and its contributions. Section 4 explains our experimental methodology. Section 5 presents the results and discussion. Section 6 presents related work and Section 7 presents our conclusions and future work.

2 Background

2.1 Memory Management in Virtualized Nodes

Cloud services are built on top of virtualization technology, which make use of a hypervisor to multiplex certain physical resources such as CPUs and I/O interfaces and allocate others, such as memory. When a VM is created, it is allocated a portion of the node's memory. If the VM increases its memory demand, it may

exceed its initial allocation. In this case, the VM may swap data to its (virtual) disk device(s), suffering a significant performance loss. In this case, the VM is *underprovisioned* of memory. A VM with idle memory is *over-provisioned*. When one or more VMs are in this state while others are under-provisioned, it is necessary to re-allocate memory to re-balance the allocation and optimize overall system performance.

Analytical solutions for this problem require the analysis of many VM–application sets and combinations. The number of possibilities is huge, and attempting to obtain solutions in this way is intractable. Instead, heuristics are designed to exploit optimal allocations for known sets. Heuristics are also limited since sometimes they do not generalize well for all possible combinations. Reinforcement learning is a good alternative to overcome these limitations.

2.2 Reinforcement Learning: Markov Decision Process

Reinforcement learning problems involve an agent interacting with an environment, usually stochastic, with the goal of maximising its total (cumulative) discounted reward. These problems are typically modeled as Markov Decision Processes (MDP), which have the following elements:

- State space S, which contains all states in which the environment can be.
- Action space A, which contains the the actions the agent can take.
- Transition probability function, denoted by $P : S \times A \times S \to \mathbb{R}$. The function gives the probability that if the agent is in state s_i in timestep i and it takes action a_i, then it will transition to state s_{i+1} in the following timestep $(i+1)$.
- Reward function $R : S \times A \times S \to \mathbb{R}$, which gives the immediate reward obtained from the transition from state s_i via action a_i to new state s_{i+1}.

The objective of the agent is to learn the optimal policy $\pi : S \to A$ (state-to-action mappings) for which the discounted reward $r_t^\gamma = \Sigma_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k, s_{k+1})$ is maximized, where $\gamma \in [0, 1]$ is the discount factor. The agent learns the optimal policy by maximizing the (expected) state-action value function $Q^\pi(s, a)$, expressed by the Bellman equation [8]:

$$Q^\pi(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))] \tag{1}$$

The Q-function can be represented as 2D matrix of states and actions, where each entry $Q(s, a)$ represents the reward for an action at a given state. This approach is known as Q-Learning [13] *(QL)*. This requires quantizing both the state and action space into discrete values within the minimum and maximum range. The need to build the extensive 2D matrix of state-action space leads to a combinatorial explosion of the state–action space, which could quickly increase the learning time and memory complexity. To solve this, Mnih *et al.* [7] estimate the Q-function through a parametrized neural network (NN) function approximator, an approach called Deep Q-Learning *(DQL)*. However, both DQL and QL estimate the action as a discrete value.

In this context, Lillicrap *et al.* introduce a method to estimate the actions continuously called Deep Deterministic Policy Gradient [18] *(DDPG)*. DDPG is

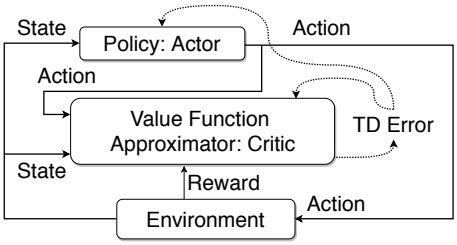

Fig. 1. Diagram of the actor–critic architecture for DDPG.

an off-policy learning algorithm [14], meaning that it learns the optimal policy interacting with the environment.

Figure 1 shows a high-level view of the DDPG learning agent. It includes two neural networks: *actor network* and *critic network*. The input to the actor network is the current state of the environment, while the input to the critic network is both the current state and the continuous actions generated by the actor. Using continuous actions for memory allocation is potentially beneficial since it allows to allocate memory at a finer granularity, and prevents the limitations imposed by discretization.

3 CAVMem: Algorithm for Virtualized Memory Management

In this section, we introduce CAVMem, a mechanism using continuous action RL for memory management in virtualized nodes. We formulate the problem as a Markov Decision Process. The MDP is formulated such that it does not impose limits on the number of VMs. It creates a RL agent per VM which has both a local view and a node-wide view, indirectly passing to a VM information on the behavior of other VMs. The reward is set to optimize aggregate performance while ensuring fairness among the VMs. In the following subsections, we detail how the MDP is formulated, its advantages and the learning mechanism.

3.1 Decentralized Strategy for Memory Management

Determining a priori the memory demands of VMs in virtualized servers is complicated because the nodes seldom have prior knowledge of applications on each VM or the number of concurrently active VMs. Instead, another approach is to let the VMs ask for the memory they need. In this case, each VM monitors its own resource utilization and bids for memory independently, resulting in a decentralized solution.

Decentralization has two advantages. First, CAVMem is designed to adjust itself to any amount of active VMs, a feature allowed by our MDP formulation. Second, CAVMem removes a single point of control, which allows to scale even beyond a single computing node.

State information from computing node and VMs

	Name	Range	Description
perVM	$msr_t^{VM_j}$	[-1, 1]	RAM miss rate of VM_j, rescaled to range [-1, 1]; i.e. a value of -1 means zero miss rate.
perVM	$M_t^{VM_j}$	[0, 1]	Fraction of total RAM allocated to VM_j.
Node	$avgMsr_t^{node}$	[-1, 1]	Average RAM miss rate across all VMs, equal to $\frac{1}{N}\sum_j msr_t^{VM_j}$.
Node	$msrTgt_t^{node}$	[-1, 1]	Target RAM miss rate for all VMs given by Equation 3.
Node	$totalMemUse_t^{node}$	[0, 1]	Fraction of the node's physical RAM that is allocated to VMs, equal to $\sum_j M_t^{VM_j}$.

Table 1. State inputs to the actor and critic networks of the DDPG learning agents.

3.2 Formulating the problem as an MDP

[**State space,** S]: Table 1 lists the five state variables for each agent, two of which belong to the VM and the other three are common to all VMs on the node. This state space definition allows the agent to have *some* information about the other active VMs through the information related to the node.

[**Action space,** A]: The action $a_t^{VM_j}$ chosen for VM_j in timestep t is referred to as the VM's **memory bid**, which ranges from -1.0 to 1.0. A positive bid is a request for more memory and a negative bid is an action to release memory. Concretely, the VM agent requests a total memory allocation equal to $(1 + a_t^{VM_j}) \times M_t^{VM_j}$. As discussed later, it may not be possible to fully satisfy the request, for instance when there is insufficient memory for all requests or the memory allocation for the VM would become too small.

[**Reward function,** R]: The reward function encourages a fair level of performance across the VMs by making all VMs suffer the same amount of swapping over each time step. This is done through a penalty (negative reward) proportional to the absolute difference between the RAM miss rate of the VM, given by $msr_{t+1}^{VM_j}$ and the miss rate target, $msrTgt_{t+1}^{node}$:

$$r_t^{VM_j} = -|msr_t^{VM_j} - msrTgt_t^{node}| \times k \tag{2}$$

The parameter k, which we set to 1, affects the learning rate. The miss rate target for the next timestep is given by:

$$msrTgt_{t+1}^{node} = avgMsr_t^{node} \times \sqrt{totalMemUse_t^{node}} \tag{3}$$

Here, $avgMsr_{node}$ is the average RAM miss rate across all active VMs, defined in Table 1. When it is non-zero and there is free memory, the factor $totalMemUse_{node}$ encourages the VMs to use more memory, via a well-known square-root rule of thumb.

Environment Constraints for Decentralized Control Figure 2 shows a high-level view of our system, including a module called the Bid Analyzer (BA), the VMs and the learning agents of each VM. The BA checks the bids of the

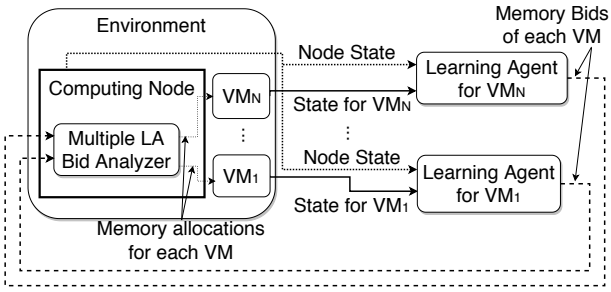

Fig. 2. Diagram of the DDPG learning Agents, one per VM.

learning agents before issuing them to the node. It prevents VM memory allocation from being less than a given threshold (set manually at 384 MB) and it prevents memory allocation from exceeding the node's capacity. Additionally, it also enforces bid prioritization for VMs that issue negative bids.

Action Space Exploration A classic problem in RL is the *exploration–exploitation dilemma* [14], which captures the need to exploit the "optimal" solution obtained so far but also explore potential better actions. To explore in continuous action domains, the common approach is to use an Ornstein–Uhlenbeck process [15, 16]. This process is a noise signal that creates a Brownian motion around the deterministic action generated by the DDPG agent. At every timestep, a noise signal is added to the action with a probability of ϵ. In our implementation, initially there's a phase of aggressive exploration in which $\epsilon = 1$. This probability reduces over time until it reaches $\epsilon = 0.001$. This process is called epsilon annealing, and it is used to transition from explorative policy to an exploitative one [14].

4 Experimental Framework

CAVMem was implemented in Python 3.6 with Tensorflow 1.12 [12] on a system with a 2.3 GHz Intel Core i7 processor and 16 GB of RAM. CAVMem is designed with an underlying DDPG implementation, but we also tested it with DQL and QL implementations for comparison, and a static allocation policy that divides memory equally among VMs. Whenever CAVMem is mentioned, it is assumed that the agents are DDPG implementations (continuous action).

The neural networks for DDPG are fully-connected with two hidden layers of 64 units with ReLU activation functions, while the output layers use Tanh. The learning rates for the actor and critic networks are 0.0001 and 0.001, respectively, using the Adam optimizer. The exploration phase lasts for 200 episodes, where each episode has 100 steps. All experiments run for 1000 episodes.

DQL has similar parameters, but it only has one learning rate of 0.001 for the Q-function approximator for 3 discrete actions. The QL implementation also consists of 3 actions, and it has each state space variable quantized in 10 discrete values. This results in a state-action matrix of 300,000 entries.

Label	Benchmark Name	Description
Scenario 1	VM1, VM2: *perlbench*	Both VMs run *perlbench* repeatedly.
Scenario 2	VM1: *perlbench*; VM2: *gcc*	VM1 runs *perlbench* repeatedly, VM2 runs *gcc*.
Scenario 3	VM1: *gcc*; VM2: *bzip2*	VM1 runs *gcc* repeatedly, VM2 runs *bzip2*.

Table 2. List of scenarios used for evaluation

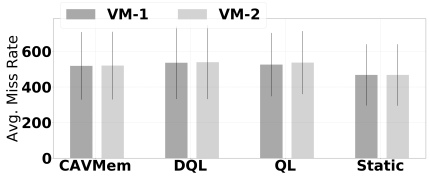

Fig. 3. Average Miss Rate for Scenario 1. Less is better

The learning agents are deployed in a simulation environment of a virtualized node, which include models of VMs and some aspects of system memory (capacity, RAM misses, utilization). This evaluation methodology allows to see the effectiveness of the MDP formulation, without intervention of specific hardware.

We evaluate CAVMem in scenarios consisting of different combinations of VMs executing SPEC CPU 2006 benchmarks [17], summarized in Table 2. Every scenario runs two VMs with 6 GB of RAM, with 1 GB allocated initially.

In every scenario, three metrics are evaluated: 1) the average miss rate (pages/second), 2) the average miss rate deviation, and 3) the overhead (time spent in seconds) associated to the training of each agent. Every metric is measured over the last 100 episodes of each experiment. It is desirable that the average miss rate is minimized by the agent, since this would avoid disk accesses on a real system. The miss rate deviation is to compare how well the agent learns the desired behavior according to the reward function. And the overhead allows us to estimate the cost to the deployment of each agent.

5 Results for Evaluation

5.1 Results for Scenario 1

Figure 3 shows the average miss rates for Scenario 1. CAVMem with *DDPG* has a miss rate 10% larger than *static*, while *DQL* and *QL* are higher by 14.7% and 13.9% respectively. Since the VMs are executing the same benchmark, an equal memory allocation is optimal resulting in *static* being the best in this case. Nevertheless, we see that *DDPG* performs the best among the learning agents.

Figure 4 shows the average miss rate deviations for Scenario 1. CAVMem tracks the target better than all other approaches. CAVMem has a miss rate deviation 82.2% smaller than *static*, and 66.1% and 75.4% smaller than *DQL* and *QL*, respectively.

Figure 5 shows the overheads for Scenario 1. *DQL* presents an overhead 10.64 and 15.0 times larger than CAVMem and *QL* respectively, a major consideration

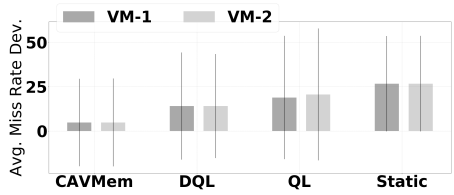

Fig. 4. Average Miss Rate Deviation for Scenario 1. Less is better

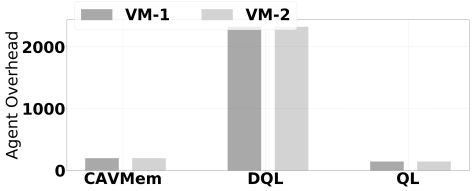

Fig. 5. Overhead (in seconds) for each agent in Scenario 1. Less is better

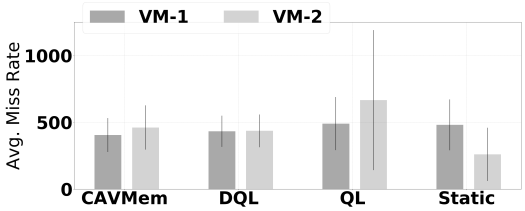

Fig. 6. Average Miss Rate for Scenario 2. Less and equal is better

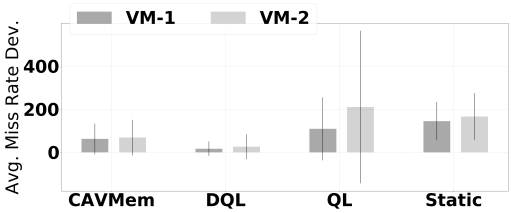

Fig. 7. Average Miss Rate Deviation for Scenario 2. Less is better

when deploying *DQL*. *QL* presents the smallest overhead, while CAVMem's is 37% larger than *QL*'s.

5.2 Results for Scenario 2

Figure 6 shows the average miss rates for Scenario 2. *QL* and *static* fail to balance the miss rates, which is an undesirable result, since they both benefit one VM while harming the other. CAVMem and *DQL* maintain the miss rate balance to a better degree by increasing the miss rate of both VMs. CAVMem minimizes the miss rate 6.41% below *DQL* for VM1 and 5.81% above *DQL* for VM2. Seemingly, CAVMem and *DQL* are competitive in performance.

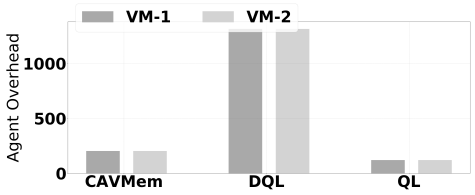

Fig. 8. Overhead (in seconds) for each agent in Scenario 2. Less is better

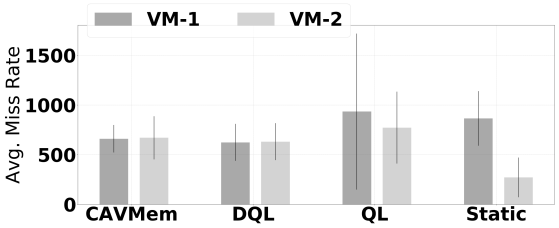

Fig. 9. Average Miss Rate for Scenario 3. Less and equal is better

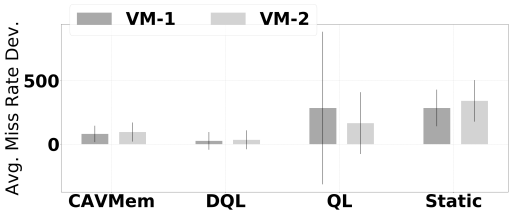

Fig. 10. Average Miss Rate Deviation for Scenario 3. Less is better

Figure 7 shows the miss rate deviations for Scenario 2. Here, *static* is further away from the target, while CAVMem deviates by 67.6% more than *DQL*, even though they both minimize the miss rate to similar values.

Figure 8 shows the overhead of the agents for Scenario 2. *DQL* maintains an overhead 5.44 and 9.96 times larger than CAVMem and *QL*, respectively. CAVMem's overhead is 70% larger than *QL*'s but performs similar to *DQL*. Thus far, CAVMem provides the best performance-overhead trade-off.

5.3 Results for Scenario 3

Figure 9 shows the average miss rates for Scenario 3. *DQL* and *static DQL* present imbalanced miss rates for the VMs, while CAVMem and *DQL* yield similar values, with CAVMem being 5.96% larger than *DQL*.

Figure 10 shows the miss rate deviations for Scenario 3. *DQL* tracks the miss rate targets better being 65.92% more accurate than CAVMem, while *QL* and *static* fail to do so. We omit the overhead charts for this case due to lack of space, but the experiments show that *DQL*'s overhead is 10.9 and 14.2 times larger than CAVMem's and *QL*'s, respectively. CAVMem's overhead is larger than *QL*'s by 28%, still maintaining the cost-effectiveness of CAVMem.

5.4 Discussion

We can summarize the findings as follows: 1) CAVMem with *DDPG* and *DQL* are able to minimize and balance the miss rates in a comparable way, better than *static* in most cases, while *QL* consistently fails to do so, 2) *DQL* tracks the performance target better than CAVMem with *DDPG*, while *static* and *QL* fail in most instances, 3) *DQL* has extremely high learning overheads when compared to CAVMem with *DDPG* and *DQL*, highlighting a performance-cost trade-off.

6 Related Work

Many solutions have been proposed to solve the resource allocation problem in cloud infrastructures [11, 10, 1, 3, 4, 16] but so far, there have not been too many efforts that use RL to exclusively solve the memory management problem in a single node, and much less continuous-action RL solutions. There are some notable RL-based solutions for the resource allocation problem that targets CPU and memory (and other resources) within a virtualized node. In [10], the authors implemented a model-based DQL algorithm for VM resource configuration, which included CPU time, virtual CPU and memory. CAVMem differs from these RL-based solutions in three ways: 1) CAVMem uses continuous-action RL exclusively for memory management, 2) CAVMem avoids discretization, and 3) it is decentralized.

7 Conclusions and Future Work

This paper proposes CAVMem as a proof-of-concept of a distributed MDP formulation for the memory allocation problem in virtualized nodes. Moroever, CAVMem also offers a continuous-action RL agent to solve the allocation problem, avoiding discretization and exploiting de-centralization. Our results show that the DDPG agent of CAVMem performs similar to the well-known DQL but with much less learning overhead, making it a very cost effective solution.

For future work, CAVMem should be deployed in a real computing node, and an exhaustive search of the parameter space of the learning agents is also necessary. Likewise, it is necessary to test CAVMem with more combinations of benchmarks and scenarios.

Acknowledgements

This research is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA) and the European Unions 7th Framework Programme under grant agreement number 610456 (Euroserver). It also received funding from the Spanish Ministry of Science and Technology (project TIN2015-65316-P), Generalitat de Catalunya (contract 2014-SGR-1272), and the Severo Ochoa Programme (SEV-2015-0493) of the Spanish Government.

References

1. W. Zhang, H. Xie, C. Hsu: Automatic Memory Control of Multiple Virtual Machines on a Consolidated Server. In: IEEE Transactions on Cloud Computing, vol. 5, no. 1, pp. 2–14. (2017).
2. J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin: Vconf: a reinforcement learning approach to virtual machine auto configuration. In: Proceedings of the 6th International Conference on Autonomic Computing (ICAC'09), pp. 137–146, 2009.
3. L.A. Garrido and P. Carpenter. vMCA: Memory Capacity Aggregation and Management in Cloud Environments. In: IEEE 23rd Intl. Conference on Parallel and Distributed Systems (ICPADS), 2017.
4. L.A. Garrido and P. Carpenter. Aggregating and Managing Memory Across Computing Nodes in Cloud Environments. In: Proceedings of the 12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC), 2017.
5. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski,G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of Cloud computing. Commun. ACM, vol. 53, no. 4, April 2010.
6. Q. Zhang, L. Cheng, and R. Boutaba, Cloud computing: state-of-the-artand research challenges,Journal of Internet Services and Applications,vol. 1, no. 1, pp. 718, 2010
7. V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis. Human-level control through deep reinforcement learning. Nature (2015). 518. 529-33. 10.1038/nature14236.
8. H. Van Hasselt. Reinforcement Learning in Continuous State and Action Spaces. In: Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, Berlin, Heidelberg, 2012.
9. G. Dulac-Arnold, R. Evans, P. Sunehag, B. Coppin. Reinforcement Learning in Large Discrete Action Spaces. CoRR abs/1512.07679 (2015).
10. J. Rao, X. Bu, C.Z. Xu, L. Wang, G.Y. Gang. VCONF: A reinforcement learning approach to virtual machines auto-configuration. Proceedings of the 6th International Conference on Autonomic Computing, pp. 137-146. ICAC 2009
11. X. Bu, J. Rao, C.Z Xu. Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach. In: IEEE Trans. Parallel Distributed Systems, pp. 681–690. April 2013.
12. M. Abadi et. al. TensorFlow: a system for large-scale machine learning. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation, 2016. USENIX Association, Berkeley, CA, USA, pp 265-283.
13. C.J.C.H. Watkins. Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England, 1989.
14. R. S. Sutton, A. G. Barto. Introduction to Reinforcement Learning (1st ed.). MIT Press, Cambridge, MA, USA. 1998.
15. E. Bibbona, G. Panfilo, P. Tavella. The Ornstein–Uhlenbeck process as a model of a low pass filtered white noise. Metrologia. 2008.
16. T. Li, Z. Xu, J. Tang, Y. Wang. Model-free control for distributed stream data processing using deep reinforcement learning. In Proceedings of Very Large Database Endowment. February, 2018.
17. J.L. Henning. SPEC CPU2006 benchmark descriptions. SIGARCH Computer Architecture News, pp. 1–17. 2006.
18. T.P. Lillicrap, J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D.Silver, D. Wierstra. Continuous control with deep reinforcement learning. CoRR. 2015.