A Deep Q-learning Scaling Policy for Elastic Application Deployment

Fabiana Rossi

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
f.rossi@ing.uniroma2.it

Abstract The ability of cloud computing to provide resources on demand encourages the development of elastic applications. Differently from the popular threshold-based solutions used to drive elasticity, we aim to design a flexible approach that can customize the adaptation policy without the need of manually tuning various configuration knobs. In this paper, we propose two Reinforcement Learning (RL) solutions (i.e., Q-learning and Deep Q-learning) for controlling the application elasticity. Although Q-learning represents the most popular approach, it may suffer from a possible long learning phase. To improve scalability and identify better adaptation policies, we propose Deep Q-learning, a model-free RL solution that uses a deep neural network to approximate the system dynamics. Using simulations, we show the benefits and flexibility of Deep Q-learning with respect to Q-learning in scaling applications.

Keywords: Deep Q-learning · Elasticity · Reinforcement Learning · Self-adaptive systems.

1 Introduction

The dynamism of working conditions calls for an elastic application deployment, which can be adapted in face of changing working conditions (e.g., incoming workload) so to meet stringent Quality of Service (QoS) requirements. Cloud providers, e.g., Amazon, Google, that support multi-component applications usually use static thresholds on system-oriented metrics to carry out the adaptation of each application component. As shown in [11,12], the manual tuning of such scaling thresholds is challenging, especially when we need to specify critical values on system-oriented metrics and the application exposes its requirements in terms of user-oriented metrics (e.g., response time, throughput, cost). Differently from the popular static threshold-based approaches, we aim to design a flexible policy that can adapt the application deployment, according on user-defined goals, without the need of manually tuning various configuration knobs.

In this paper, we use Reinforcement Learning (RL) to adapt the application deployment. RL allows to express *what* the user aims to obtain, instead of *how* it should be obtained (as required by threshold-based policies). Most of the existing works consider model-free RL algorithms, e.g., Q-learning, to drive the application

J. Manner, S. Haarmann, S. Kolb, N. Herzberg, O. Kopp (Eds.): 13th ZEUS Workshop, ZEUS 2021, Bamberg, held virtually due to Covid-19 pandemic, Germany, 25-26 February 2021, published at http://ceur-ws.org

deployment (e.g., [2,5]). However, one of the main issue with Q-learning is the possibly long learning phase, which is especially experienced when the number of system states increases. An approach to boost the learning process is to provide the agent with system knowledge. By exploiting it, the agent can more easily find a suitable trade-off between system- and user-oriented metrics (i.e., resource utilization and target response time). Therefore, together with Q-learning, we propose Deep Q-learning [8]. It uses deep neural networks to approximate the system dynamics and to drive the application elasticity. Using simulation, we demonstrate the advantages of the Deep Q-learning solution, which learns a suitable scaling policy while meeting QoS requirements expressed in term of target application response time (i.e., $R_{\max}$).

2 Related Work

The existing elasticity solutions rely on a wide set of methodologies, that we classify in the following categories: mathematical programming, control theory, queuing theory, threshold-based, and machine-learning solutions. The mathematical programming approaches exploit methods from operational research in order to solve the application deployment problem (e.g., [13,21]). The main drawback of these solutions is scalability; since the deployment problem is NP-hard, other efficient solutions are needed. As surveyed in [15], few solutions use control theory to change the replication degree of application containers (e.g., [3]). The critical point of the control-theoretic approaches is the requirement of a good system model, which can sometimes be difficult to be formulated. The key idea of queuing theory is to model the application as a queuing system with inter-arrival and service times having general statistical distributions (e.g., [4,10]). Nevertheless, queuing theory often requires to approximate the system behavior, discouraging its adoption in a real environment. Most of the existing solutions exploit best-effort threshold-based policies based on the definition of static thresholds for adapting the application deployment at run-time (e.g., [7,16]). Although a threshold-based scaling policy can be easily implemented, it is a best-effort approach that moves complexity from determining the reconfiguration strategy to the selection of critical values that act as thresholds. In the last few years, machine learning has become a widespread approach to solve at run-time the application deployment problem. RL is a machine learning technique by which an agent can learn how to make good (scaling) decisions through a sequence of interactions with the environment [17]. After executing an adaptation action in the monitored system state, the RL agent experiences a cost that contributes to learning how good the performed action was. The obtained cost leads an update of a lookup table that stores the estimation of the long-term cost for each state-action pair (i.e., the Q-function). Many solutions considered the classic model-free RL algorithms (e.g., [9,18]), which however suffer from slow convergence rate. To tackle this issue, different model-based RL approaches have been proposed, e.g., [14,19]. They use a model of the system to drive the action exploration and speed-up the learning phase. Although model-based RL approaches can overcome the slow

convergence rate of model-free solutions, they can suffer from poor scalability in systems with a large state space. In this solution, the lookup table has to store a separate value for each state-action pair. An approach to overcome this issue consists in approximating the system behavior, so that the agent can explore a reduced number of system configurations. Integrating deep neural networks into Q-learning, Deep Q-learning has been widely applied to approximate the system dynamics in a variety of domains, e.g., traffic offloading [1]. However, to the best of our knowledge, it is so far poorly applied in the context of application elasticity. Differently from all the above contributions, in this paper we propose an application scaling policy based on Deep Q-learning; then, we compare it against Q-learning.

3 RL-based Scaling Policy

RL is a special method belonging to the branch of machine learning. It refers to a collection of trial-and-error methods by which an agent must prefer actions that it found to be effective in the past (*exploitation*). However, to discover such actions, it has to explore new actions (*exploration*).

At each discrete time step i, according to the monitored metrics, the RL agent determines the application state and updates the expected long-term cost (i.e., Q-function). We define the application state as $s = (k, u)$, where k is the number of application instances, and u is the monitored CPU utilization. We denote by $\mathcal{S}$ the set of all the application states. We assume that $k \in \{1, 2, ..., K_{\max}\}$; being the CPU utilization (u) a real number, we discretize it by defining that $u \in \{0, \bar{u}, ..., L\bar{u}\}$, where $\bar{u}$ is a suitable quanta and $L \in \mathbb{N}$ such that $L \cdot \bar{u} = 1$. For each state $s \in \mathcal{S}$, we define the set of possible adaptation actions as $\mathcal{A}(s) \subseteq \{-1, 0, 1\}$, where $+1$ is the *scale-out* action, -1 the *scale-in* action, and 0 is the *do nothing* decision. We observe that not all the actions are available in any application state, due to the lower and upper bounds on the number of application instances (i.e., 1 and $K_{\max}$, respectively). According to an action selection policy, the RL agent identifies the scaling action a to be performed in state s. The execution of a in s leads to the transition in a new application state (i.e., s') and to the payment of an immediate cost. We define the immediate cost $c(s, a, s')$ as the weighted sum of different normalized terms, such as the *performance penalty*, c_{perf}, and *resource cost*, c_{res}. Formally, we have:

$$c(s, a, s') = w_{\text{perf}} \cdot c_{\text{perf}} + w_{\text{res}} \cdot c_{\text{res}} \tag{1}$$

where w_{perf} and w_{res}, $w_{\text{perf}} + w_{\text{res}} = 1$, are non negative weights that allow us to express the relative importance of each cost term. The *performance penalty* is paid whenever the average application response time exceeds the target value $R_{\max}$. The *resource cost* is proportional to the number of application instances. We can observe that the formulation of the immediate cost function $c(s, a, s')$ is general enough and can be easily customized with other QoS requirements.

The received immediate cost contributes to update the Q-function. The Q-function consists in $Q(s, a)$ terms, which represent the expected long-term cost

that follows the execution of action a in state s. The existing RL policies differ in how they update the Q-function and select the adaptation action to be performed (i.e., action selection policy) [17]. In [14], for example, we propose a model-based RL approach that enriches the RL agent with a model of the system to drive the exploration actions and speed up the learning phase. Since determining the system model can be a not trivial task, in this paper we consider two model-free RL solutions to adapt the application deployment. First, we consider the simple model-free Q-learning (QL) algorithm that uses a table (i.e., Q-table) to store the Q-value for each state-action pair. The Q-table allows to store the real experience without approximation. However, this approach may suffer from slow convergence rate when the number of state-action pairs increases. Then, to tackle this issue, we present a Deep Q-learning (DQL) approach that combines Q-learning with deep neural networks. The neural network allows to approximate the Q-function using a non-linear function; in such a way, the agent can directly compute $Q(s, a)$ using s and a, instead of performing a Q-table lookup. By using a Q-function approximation, the RL agent does not need to explore all the state-action pairs before learning a good adaptation policy.

Q-learning. At time i, the QL agent selects the action a to perform in state s using an ϵ-greedy policy on $Q(s, a)$; the application transits in s' and experiences an immediate cost c. The ϵ-greedy policy selects the best known action for a particular state (i.e., $a = argmin_{a \in A(s)} Q(s, a)$) with probability $1 - \epsilon$, whereas it favors the exploration of sub-optimal actions with low probability, ϵ. At the end of time slot i, $Q(s, a)$ is updated using a simple weighted average:

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha \left[c + \gamma \min_{a' \in \mathcal{A}(s')} Q(s', a') \right] \tag{2}$$

where $\alpha, \gamma \in [0, 1]$ are the *learning rate* and the *discount factor*, respectively.

Deep Q-learning. DQL uses a multi-layered neural network, called Q-network, to approximate the Q-function. For each time slot i, the DQL agent observes the application state and selects an adaptation action using an ϵ-greedy policy and the estimates of Q-values, as Q-learning does. Note that DQL is model-free: it solves the RL task directly using samples, without explicitly modeling the system dynamics. In a given state s, the Q-network outputs a vector of action values $Q(s, \cdot, \phi)$, where ϕ are the network parameters. By approximating the Q-function, the RL agent can explore a reduced number of system configurations before learning a good adaptation policy. At the end of each time slot i, the Q-network is updated by performing a gradient-descent step on $(y_i - Q(s, a, \phi_i))^2$ with respect to the network parameters ϕ_i. y_i is the estimated long-term cost, defined as $y_i = c + \gamma \cdot min_{a'} Q(s', a', \phi_i)$, where γ is the discount factor. When only the current experience is considered, i.e., (s, a, c, s'), this approach is too slow for practical real scenarios. Moreover, it is unstable due to correlations existing in the sequence of observations. To overcome these issues, we consider a revised DQL algorithm that uses a replay buffer and a separate target network to compute y_i [8]. To perform experience replay, the agent store its experience in a buffer with finite capacity. A mini-batch of experience is drawn uniformly at random from the replay buffer to remove correlations in the observation sequence

and to smooth over changes in the data distribution. In the classic DQL, the same Q-network is used both to select and to evaluate an action. This can lead to select overestimated values, resulting in overoptimistic estimates. To prevent this, two networks are used (i.e., on-line and target network) and two value functions are learned. The on-line network is used to determine the greedy policy and the target network to determine its value. The target network parameters are updated to the on-line network values only every τ steps and are held fixed between individual updates.

4 Results

We evaluate the proposed deployment adaptation solutions using simulations. Without lack of generality, at each discrete time step i, we model the application as an $M/M/k_i$ queue, where k_i is the number of application replicas. We set the service rate μ to 120 requests/s. As shown in Fig. 1, the application receives a varying number of requests. It follows the workload of a real distributed application [6]. The application expresses the QoS in terms of target response time $R_{\max} = 15$ ms. The RL algorithms use the following parameters: $\alpha = 0.1$ and discount factor $\gamma = 0.99$. We discretize the application state with $K_{\max} = 10$ and $\bar{u} = 0.1$. To update the application deployment, QL and DQL use an ϵ-greedy action selection policy, with $\epsilon = 0.1$. DQL uses a replay memory with capacity of 50 observations and a batch size of 30; the target Q-network update frequency is $\tau = 5$ time units. We use Deeplearning4j[1] library to implement the neural networks. Correctly configuring the Q-network is an empirical task, which requires some effort and several preliminary evaluations. In particular, we use ReLu as the neuron activation function; due to its non-linear behavior, it is one of the most commonly used function. To initialize the Q-network weights, we use the Xavier method [20]. To avoid weights to diminish or explode during network propagation, this method scales the weight distribution on a layer-by-layer basis. To this end, it uses a normal distribution with centered mean and standard deviation scaled to the number of layer's input and output neurons. The Q-network architecture

[1] https://deeplearning4j.org/

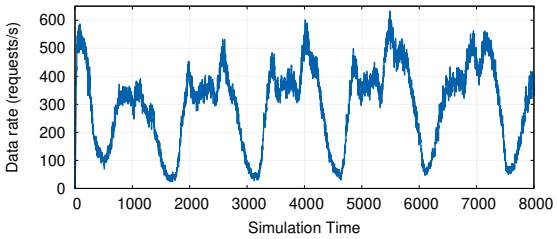
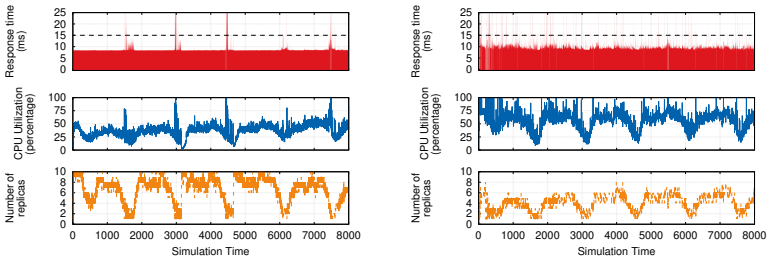


Figure 1: Workload used for the reference application.

Table 1: Application performance under scaling policies.

Elasticity Policy	*Configuration*	$R_{\max}$ *violations (%)*	*Average CPU utilization (%)*	*Average number of replicas*	*Median of Response time (ms)*
QL	$w_{\text{perf}} = 1, w_{\text{res}} = 0$	3.15	56.20	4.17	9.51
	$w_{\text{perf}} = 0.5, w_{\text{res}} = 0.5$	13.26	51.27	5.13	8.81
	$w_{\text{perf}} = 0, w_{\text{res}} = 1$	42.27	65.56	3.90	11.88
DQL	$w_{\text{perf}} = 1, w_{\text{res}} = 0$	1.05	36.85	6.63	8.38
	$w_{\text{perf}} = 0.5, w_{\text{res}} = 0.5$	39.23	66.64	3.58	11.11
	$w_{\text{perf}} = 0, w_{\text{res}} = 1$	88.51	89.18	1.58	$+\infty$
DQL with pre-trained network	$w_{\text{perf}} = 1, w_{\text{res}} = 0$	1.39	35.54	7.41	8.35
	$w_{\text{perf}} = 0.5, w_{\text{res}} = 0.5$	31.91	58.53	4.30	9.06
	$w_{\text{perf}} = 0, w_{\text{res}} = 1$	83.48	86.12	1.71	$+\infty$

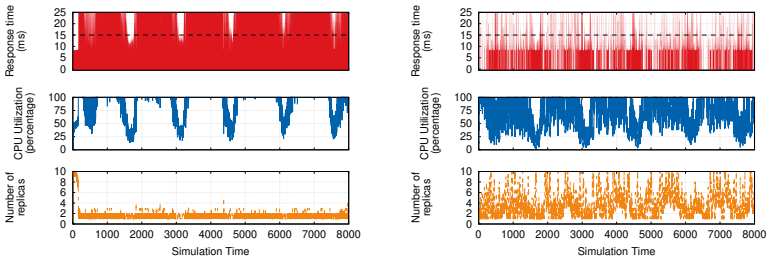


(a) Deep Q-learning (b) Q-learning

Figure 2: Application performance using the weights $w_{\text{res}} = 0$ and $w_{\text{perf}} = 1$.

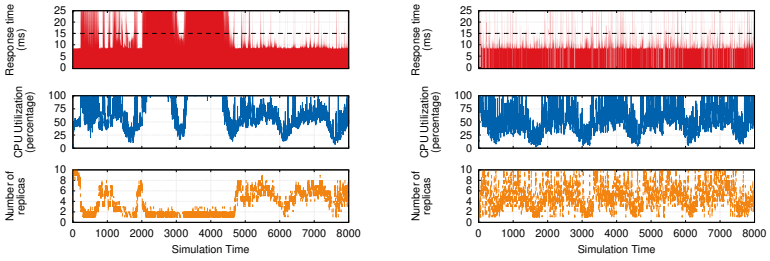
is fully-connected with 4 layers having $\{2, 15, 15, 3\}$ neurons (i.e., there are 2 hidden layers).

Table 1 summarizes the experimental results, including also the application performance obtained with a pre-trained DQL. We can see that the application has a different performance when different weights for the cost function are used (Eq. 1). We first consider the set of weights $w_{\text{perf}} = 1$ and $w_{\text{res}} = 0$: in this case, optimizing the application response time is more important than saving resources. As shown in Fig. 2b, the QL solution often changes the application deployment performing scaling operations. Moreover, the application response time exceeds $R_{\max}$ for 3.15% of the time. Conversely, taking advantage of the approximated system knowledge, the DQL solution learns a better elasticity policy that successfully controls the application deployment (Fig. 2a). It registers 1.05% of $R_{\max}$ violations and a median of the application response time lower than the target application response time (i.e., 8.38 ms). We now consider the case when saving resources is more important than meeting the $R_{\max}$ bound, i.e., $w_{\text{res}} = 1$ and $w_{\text{perf}} = 0$. Intuitively, the RL agent should learn how to improve resource utilization at the expense of a high application response time (i.e., that exceeds $R_{\max}$). Table 1 and Fig. 3 show that, in general, DQL performs better than QL in terms of resource usage. DQL registers 89.18% of resource



(a) Deep Q-learning (b) Q-learning

Figure 3: Application performance using the weights $w_{\text{res}} = 1$ and $w_{\text{perf}} = 0$.



(a) Deep Q-learning (b) Q-learning

Figure 4: Application performance using the weights $w_{\text{res}} = 0.5$ and $w_{\text{perf}} = 0.5$.

utilization running with 1.58 application replicas. This is very close to the lowest amount of resources assignable to the application. Run-time adaptations are also avoided. As a consequence, the application is overloaded and the resulting median response time is unbounded. Conversely, the QL solution struggles to find a stable configuration. It identifies an adaptation policy that runs the application using, on average, 3.90 instances. On average, its resource usage is lower than in DQL (65.56% and 89.18%, respectively), as also the percentage of the target application response time $R_{\max}$ violations. Besides the weight configurations at the opposite ends, we can obtain a wide set of adaptation strategies that differ by the relative importance of the two deployment goals. In Table 1 we propose a simple case, where we set $w_{\text{perf}} = w_{\text{res}} = 0.50$. To visualize the update of the application deployment by the two RL policies, we report in Fig. 4 the application behavior during the whole experiment, when we want to optimize the performance avoiding resource wastage ($w_{\text{perf}} = w_{\text{res}} = 0.5$). Intuitively, the RL agent has to find a trade-off between the resource usage and the number of $R_{\max}$ violations. The neural network allows to approximate the Q-function using a non-linear function; in such a way, DQL can explore a reduced number of system configurations before learning a good adaptation policy. On average, it runs

the application with 3.58 replicas, registering a 66.64% of CPU utilization. The median application response time is 11.11 ms, with about 39% of $R_{\max}$ violations. Although we could pre-train the Q-network to further improve the DQL learned policy (mitigating also the initial exploration phase), we observe that the obtained results are already remarkable, considering that DQL is model-free. Conversely, when QL updates the application deployment, it continuously performs scaling actions, meaning that QL is still exploring the best actions to perform. This behavior is also reflected on the number of application instances, whose average value is greater than those used in the $w_{\text{perf}} = 1$ configuration. Being model-free and storing experience without approximation, QL cannot quickly learn a suitable adaptation strategy for intermediate cost weight configurations during the experiment.

Discussion. In this paper, we evaluated QL and DQL to adapt the application deployment at run-time. First, we showed the flexibility provided by a RL-based solution for updating the application deployment. By correctly defining the relative importance of the deployment objectives through the cost function weights in Eq. 1, the RL agent can accordingly learn a suitable application deployment strategy. Very different application behavior can be obtained when we aim to optimize the application response time, resource saving, or a combination thereof. Second, we showed that a DQL approach takes advantage of the approximate system knowledge and outperforms QL, especially when we pre-train the Q-network. We observe that, although we do not need to define the system model as in a model-based approach, DQL introduces the effort of defining a suitable Q-network architecture. However, this is an empirical process that may require a large number of preliminary experiments and trial-and-error repetitions.

5 Conclusion

Most policies for scaling applications resort on threshold-based heuristics that require to express how specific goals should be achieved. In this paper, aiming to design more flexible solution, we have proposed Q-learning and Deep Q-learning policies for controlling the application elasticity. Relying on a simulation-based evaluation, we have shown the benefits of the proposed RL-based approaches. Deep Q-learning exploits deep neural networks to approximate the system dynamics, estimated through system interactions. The deep neural network speeds up the learning phase, improving the application performance; however, modeling the neural network architecture can be challenging.

As future work, we plan to further investigate RL approaches for elasticity. We will investigate more sophisticated techniques for improving the convergence speed of the learning process (e.g., by leveraging Bayesian Decision Trees, Function Approximation). Moreover, we plan to extend our model by explicitly considering multiple system-oriented metrics within the adaptation policies.

References

1. Alam, M.G.R., Hassan, M.M., Uddin, M.Z., Almogren, A., Fortino, G.: Autonomic computation offloading in mobile edge for IoT applications. Future Gener. Comput. Syst. **90**, 149 – 157 (2019)
2. Arabnejad, H., Pahl, C., Jamshidi, P., Estrada, G.: A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In: Proc. of IEEE/ACM CCGrid '17. pp. 64–73 (2017)
3. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proc. of ACM SIGSOFT FSE '16. pp. 217–228. ACM (2016)
4. Gias, A.U., Casale, G., Woodside, M.: Atom: Model-driven autoscaling for microservices. In: Proc. of IEEE ICDCS '19. pp. 1994–2004 (2019)
5. Horovitz, S., Arian, Y.: Efficient cloud auto-scaling with SLA objective using Q-learning. In: Proc. of IEEE FiCloud '18. pp. 85–92 (2018)
6. Jerzak, Z., Ziekow, H.: The debs 2015 grand challenge. In: Proc. ACM DEBS'15. pp. 266–268 (2015)
7. Kwan, A., Wong, J., Jacobsen, H., Muthusamy, V.: Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In: Proc. of IEEE ICDCS '19. pp. 80–90 (2019)
8. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., et al.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
9. Nouri, S.M.R., Li, H., Venugopal, S., Guo, W., He, M., Tian, W.: Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. Future Gener. Comput. Syst. **94**, 765–780 (2019)
10. Rossi, F., Cardellini, V., Lo Presti, F.: Hierarchical scaling of microservices in Kubernetes. In: Proc. of IEEE ACSOS '20. pp. 28–37 (2020)
11. Rossi, F., Cardellini, V., Lo Presti, F.: Self-adaptive threshold-based policy for microservices elasticity. In: In Proc. of IEEE MASCOTS '20. pp. 1–8 (2020)
12. Rossi, F.: Auto-scaling policies to adapt the application deployment in Kubernetes. In: CEUR Workshop Proc. 2020. vol. 2575, pp. 30–38 (2020)
13. Rossi, F., Cardellini, V., Lo Presti, F., Nardelli, M.: Geo-distributed efficient deployment of containers with Kubernetes. Comput. Commun **159**, 161 – 174 (2020)
14. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using Reinforcement Learning. In: Proc. of IEEE CLOUD '19. pp. 329–338 (2019)
15. Shevtsov, S., Weyns, D., Maggio, M.: Self-adaptation of software using automatically generated control-theoretical solutions. Engineering Adaptive Software Systems pp. 35–55 (2019)
16. Srirama, S.N., Adhikari, M., Paul, S.: Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. **160** (2020)
17. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 2 edn. (2018)
18. Tang, Z., Zhou, X., Zhang, F., Jia, W., Zhao, W.: Migration modeling and learning algorithms for containers in fog computing. IEEE Trans. Serv. Comput. **12**(5), 712–725 (2019)
19. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A hybrid Reinforcement Learning approach to autonomic resource allocation. In: Proc. of IEEE ICAC '06. pp. 65–73 (2006)

20. Xavier, G., Yoshua, B.: Understanding the difficulty of training deep feedforward neural networks. In: AISTATS'10. vol. 9, pp. 249–256 (2010)
21. Zhao, D., Mohamed, M., Ludwig, H.: Locality-aware scheduling for containers in cloud computing. IEEE Trans. Cloud Comput. **8**(2), 635–646 (2018)