Published in Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W),
http://dx.doi.org/10.1109/FAS-W.2019.00047

A Performance Evaluation of Deep Reinforcement Learning for Model-Based Intrusion Response

Stefano Iannucci
Computer Science and Engineering
Mississippi State University
stefano@cse.msstate.edu

Ovidiu Daniel Barba
Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata
ovi.daniel.b@gmail.com

Valeria Cardellini
Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata
cardellini@ing.uniroma2.it

Ioana Banicescu
Computer Science and Engineering
Mississippi State University
ioana@cse.msstate.edu

***Abstract*—Given the always increasing size of computer systems, manually protecting them in case of attacks is infeasible and error-prone. For this reason, several Intrusion Response Systems (IRSs) have been proposed so far, with the purpose of limiting the amount of work of an administrator. However, since the most advanced IRSs adopt a stateful approach, they are subject to what Bellman defined as *the curse of dimensionality*. In this paper, we propose an approach based on deep reinforcement learning which, to the best of our knowledge, has never been used until now for intrusion response. Experimental results show that the proposed approach reduces the time needed for the computation of defense policies by orders of magnitude, while providing near-optimal rewards.**

I. Introduction

Modern computer systems require a flexible architecture to handle sudden and unpredictable variations in application workloads, caused by an increasing number of connected devices. This flexibility is generally achieved by breaking down the system components into smaller, loosely coupled, and separably scalable units. Consequently, the increase in size and complexity of the aforementioned systems makes it difficult for system administrators to keep track of the large number of alerts generated by Intrusion Detection Systems (IDS) and take proper countermeasures to prevent the attacker from inflicting further damage to the system.

To overcome the need for human intervention, various Intrusion Response Systems (IRS) have been developed (e.g., [3], [5], [9], [13], [16]). They are designed to continuously monitor IDS alerts and generate appropriate actions to defend the system from a potential attack. They are classified according to their level of automation, ranging from simple static attack-response mappings (e.g., [21], [22]) to more sophisticated, automated, stateful IRSs, which require an accurate model of the system to produce an optimal long-term defense policy (e.g., [6], [7], [14]).

However, using a stateful model-based planning technique to generate a defense policy could be time prohibitive for large-scale systems, because the size of the state space grows exponentially with the size of the defended system [1]. In order to deal with this issue, several optimal [12], [18] and sub-optimal [8], [10] approaches have been proposed, based either on *planning* or *learning* algorithms. The first is a class of algorithms that requires complete knowledge of the dynamics of the system. The second class involves instead the formulation of a *Multi-Armed Bandit* problem [20], where an agent has to maximize its revenue in presence of alternative and possibly unknown choices, while trying to acquire additional experience that it can leverage for future decisions. Reinforcement Learning (RL) problems are typical instances of Multi-Armed Bandit problem.

In general, approaches based on RL, such as Q-Learning [20], leverage tabular methods to solve the decision-making problem by building an approximation of the optimal action-value function, the function that helps in choosing the best action in a given state. For large-scale systems, the major drawbacks of tabular methods are the high demand in computer memory to store the state-action values for the huge state space, and the lack of generalization of past experiences, making it impossible for the IRS to know how to behave in unseen states.

Deep Reinforcement Learning overcomes the aforementioned limitations because, instead of storing the full action-value table, it only needs to store the parameters of the underlying neural network, and a single forward pass is enough to find the best action to take in a particular state. Deep Learning and in particular Deep Reinforcement Learning have been recently applied to intrusion detection (e.g., [11], [23]). However, to the best of our knowledge, this approach has never been used for intrusion response.

This work intends to explore the feasibility, in terms of performance, of using Deep Q-Learning instead of the traditional Q-Learning for intrusion response. We model a large microservice-based system, and compare the performance of Q-Learning and Deep Q-Learning in terms of steps to convergence, cumulative reward, and execution time. Our preliminary results show that Deep Q-Learning is orders of magnitude faster than Q-Learning in terms of time and number of

episodes needed to converge to an optimal cumulative reward.

This paper is organized as follows: in Section II, the background on the underlying mathematical framework and on the system model is provided. Experimental results are discussed in Section III. Finally, conclusions are drawn in Section IV.

II. Methodology and Model

The proposed IRS leverages a stateful model of the system to plan sequences of defense actions to protect it, upon detection of an attack by an IDS.

The system model is based upon a *Markov Decision Process* (MDP [17]). An introduction to the MDP framework is given in II-A, and different solvers are presented in Section II-B. Finally, in Section II-C, the system model is described.

A. The MDP Framework

An MDP is a mathematical framework adopted to formalize sequential decision making, where actions do not only affect immediate rewards, but also future states and, consequently, future rewards [17].

It is defined with the tuple: $M = \langle S, A, P, R \rangle$, where S is the *state space*, that is the set of all the possible states in which an agent can be at any discrete time step $t = 0, 1, 2 \ldots$. At each t, the agent receives a representation of the current state from the environment, $S_t \in S$, and chooses an action $A_t \in A$ to execute. The outcome of the action is defined by the transition matrix $P : S \times A \times S \to \mathbb{R}$, such that $p(S_t, A_t, S_{t+1})$ is the probability that, by executing action A_t in state S_t, the agent will move to state S_{t+1}. Every time an action is executed, the MDP agent is rewarded with a bonus (or penalized with a cost), according to the reward function $R : S \times A \times S \to \mathbb{R}$. That is, $R_t = R(S_t, A_t, S_{t+1})$ represents the reward that the agent will earn (or the cost the agent will pay) for executing in state S_t the action A_t and being taken to some state S_{t+1}.

The overall behavior of the agent is described by a policy π, that specifies a probability distribution such that $\pi : S \times A \to [0, 1]$. That is, $\pi(A_t|S_t)$ represents the probability that the agent will execute action A_t while in state S_t. Solving an MDP means to find a policy π^* such that the discounted reward $R_t = \sum_{j=0}^{\infty} \gamma^j R_{t+j+1}$ is maximized [20], where γ is the discount factor, which is used to balance the preference of short-term rewards over long-term ones.

In order to avoid the explicit enumeration of the states in the state space, a factored representation of the MDP is adopted, such that each state $s \in S$ is described as a multivariate random variable $s = (a_1, a_2, \ldots, a_n)$, where each variable (or attribute) $a_i \in s$ can take a value in its domain $\mathrm{Dom}(\mathrm{a_i})$. Furthermore, this representation simplifies the model of the actions, which can now be described by a set of difference equations over the state variables associated with each action post-condition. Finally, the factored representation is leveraged to build the termination function $TF : S \to \{\mathrm{true}, \mathrm{false}\}$, used to identify $T \subseteq S = \bigcup_{s \in S} \{s | TF(s) = \mathrm{true}\}$, the set of states in which the system is considered *secure*.

B. MDP Solvers

One of the most commonly used MDP *planners* is *Value Iteration* (VI) [17], [20], because of its simplicity. It is based on the concept of *state-value function* $V_\pi(S_t) = \mathbb{E}_\pi[R_t|S_t]$ that is, the expected achievable reward by the agent starting from state S_t and then following policy π. The base step of the algorithm is to assign an initial random state-value V^0 to all the states, and then to execute the iterative refinement process described in [20]:

$$V^{i+1}(S_t) = \max_{A_t \in A} R(S_t) + \gamma \sum_{S_{t+1} \in S} P(S_t, A_t, S_{t+1}) V^i(S_{t+1}) \tag{1}$$

The sequence of functions V^i converges linearly to the optimal value V^* in the limit, and thus, it provides the expected maximum reward obtainable by following the optimal policy π^* from state S_t.

As opposed to planning algorithms, which require a full knowledge of the system, *learning* algorithms are not aware of P and R. Indeed, they use an approach based on trial-and-error interactions with a dynamic environment to learn its behavior. One of the most commonly used learning algorithms is Q-Learning [20]. It is a Temporal-Difference approach to estimate the action-value function $Q_\pi(S_t, A_t)$ (also known as *q-value function*) of an MDP, which, similarly to the state-value function, denotes the expected return starting from state S_t, taking the action A_t, and thereafter following policy π. State-values can easily be derived from state-action values, and viceversa. The update rule of Q-Learning is defined by:

$$\begin{aligned} Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \\ + \alpha[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{2}$$

The action A_t to perform at time step t can be chosen applying an ϵ-greedy policy, in which $A_t = argmax_a Q(S_{t+1}, a)$ with probability $1 - \epsilon$ (exploitation step), while A_t is randomly chosen from the set of possible actions A with probability ϵ (exploration step). In other words, the ϵ parameter determines the exploration-exploitation trade-off. It has been shown that, by using the update rule described in Equation 2, the Q-Value function Q converges to Q^*, the optimal action-value function.

As already mentioned in Section I, one of the main limitations of both VI and Q-Learning is that they are tabular methods, that is, they store in memory the state-values (in case of VI) or the state-action values (in case of Q-Learning) for every (visited, in case of Q-Learning) state in the state space. For this reason, and since the state space grows exponentially according to the number of attributes used to characterize the states, their application is limited to small-scale systems. Furthermore, another weakness of Q-Learning is the lack of generalization. Indeed, it cannot determine which action to take in an unseen state.

Deep Q-Learning [15], a variation of classic Q-Learning, addresses these problems by using a convolutional neural network as a non-linear function approximator to estimate the action-value function, $Q(S_t, A_t|\Theta) \approx Q^*(S_t, A_t)$. The neural network with parameters Θ is known as *Q-Network*, and it allows generalization of the experience of the learning agent from the interaction with the environment. A Q-Network can be trained by minimizing a sequence of loss functions $L_i(\Theta_i)$ that changes at each iteration i of the algorithm:

$$L_i(\Theta_i) = \mathbb{E}_{S_t, A_t \backsim \rho(.)}[(y_i - Q(S_t, A_t; \Theta_i))^2] \quad (3)$$

where $\rho(S_t, A_t)$ is the behavior distribution over states S_t and actions A_t, and

$$y_i = \mathbb{E}_{S_{t+1}}[R_t + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}; \Theta_{i-1}|S_t, A_t)] \quad (4)$$

is a quantity similar to the target of the update rule of Q-Learning. As opposed to Q-Learning, in Deep Q-Learning the average value of multiple (s, a) tuples observed from the interaction with the environment is used, instead of a single one. The target value y_i depends on the parameters Θ_i of the network at the previous iteration $i - 1$, and it is not changed while optimizing the loss function $L_i(\Theta_i)$. The gradient of the loss function with respect to the weights $\nabla_{\Theta_i} L_i(\Theta_i)$ is used in the Stochastic Gradient Descent algorithm to train the network.

C. The System Model

Given the ever-growing complexity and heterogeneity of modern computer systems, realizing a general modeling framework is an arduous task. In this paper, we mainly focus on microservice-oriented architectures. However, the modeling framework is flexible enough to consider traditional architectures, such as three-tier web applications. An overview of the class diagram used to build the framework is depicted in Figure 1. The core element is `ComponentGroup`, which represents a microservice and is made up of various interchangeable instances (`Component`) offering the same functionality. Every component j can be globally identified in its group i as (i, j). The state of the components is modeled as a list of `StateVariables`, each one identified by a name and, for simplicity, but without loss of generality, all of them are boolean. Furthermore, we also consider that all the components in a group have the same state variables. In general, the number and type of the variables can be different for each component group, but they all share the default ones:

- **Active.** $\mathrm{Act}_{(\mathrm{i,j})}$ defines whether or not the component is currently turned on or off;
- **Updated.** $\mathrm{Upd}_{(\mathrm{i,j})}$ determines if the element is updated to the latest version;
- **New Version Available.** $\mathrm{Nva}_{(\mathrm{i,j})}$ specifies if a new software version can be used to update the current one;
- **Corrupted.** $\mathrm{Corr}_{(\mathrm{i,j})}$ determines whether a component is compromised and it needs corrective actions to take it into a healthy state;
- **Vulnerable.** $\mathrm{Vul}_{(\mathrm{i,j})}$ states whether or not the current software version of the component can be subject to a known attack. If it is vulnerable, upgrading to a new version, if available, can reduce the probability of being exposed to that particular threat.

The state variables are intentionally not application or attack-specific, because our goal is to simulate complex large scale systems and it can be a tedious task to enumerate specific variables. State variable k of instance j in group i can be identified by (i, j, k), where $k \in \{\mathrm{Act}, \mathrm{Upd}, \mathrm{Nva}, \mathrm{Corr}, \mathrm{Vul}\}$ is the name of the variable.

The connection between a requesting component group that uses services exposed by a providing group can be represented with a `Connector`. The latter also specifies what protocol the two microservices use to exchange data. Connectors are contained in `ConnectorGroups`, an abstract class representative of a communication environment or component, e.g., a network, a load balancer or a proxy. The connector group can connect two microservices by instantiating a specific `Connector`, adding it to the outgoing connectors list of the requester service and to the incoming connectors list of the provider service. For example, if on the same network we have multiple NodeJS servers and a MongoDB cluster, with the former using the latter as a persistence component, the network's `ConnectorGroup` instance will: (i) create a `Connector` between the two; (ii) add the connector to the outgoing list of the component group NodeJS (since it is the requester); (iii) add the connector to the incoming list of MongoDB component group (because it is the provider of the service). Figure 2 provides a high-level overview on how to map actual system components to our modeling framework.

The `APIGateway` is defined as a particular `ComponentGroup`, since it has the same default state variables (active, corrupted, vulnerable, etc) and it can connect to other component groups over a connector group (network, load balancer) to route incoming requests. The main differences are that it has a public IP address and can add a public-facing firewall as a defense mechanism. `Firewall` instances can be also added to connector groups and have the ability to block malicious IPs. The entire composition of the system is contained in `SystemModel`, which is used by the translation process to convert it into the underlying MDP model, using the same approach described in [7].

The reward function is characterized as a penalty score on the considered actions, and it is defined as:

$$R(S_t, a, S_{t+1}) = -w_t \frac{T(a)}{T_{max}} - w_c \frac{C(a)}{C_{max}} \quad (5)$$

where the weights $w_t, w_c \in [0, 1]$ represent the importance of, respectively, the execution time $T(a)$ and cost $C(a)$ of action $a \in A$. Time and cost values are normalized using their respective maximum value, T_{max} and C_{max}.

The set of actions A consists of 6 actions that can be applied to any component and can potentially modify its state. Each action is identified with a name and the ID of the component on which it can be executed, and is characterized by a certain execution time $T(a)$ and cost $C(a)$. Furthermore, each action is characterized by: a boolean *pre-condition* function

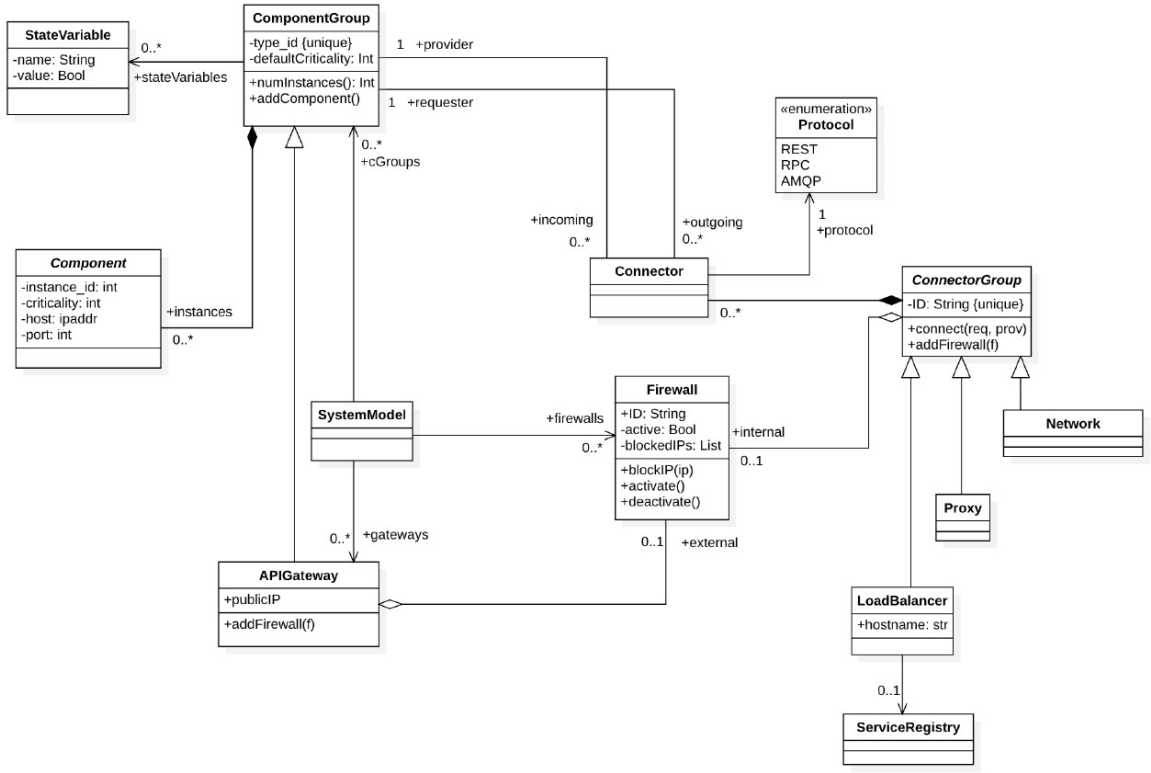

Fig. 1. System modeling framework class diagram

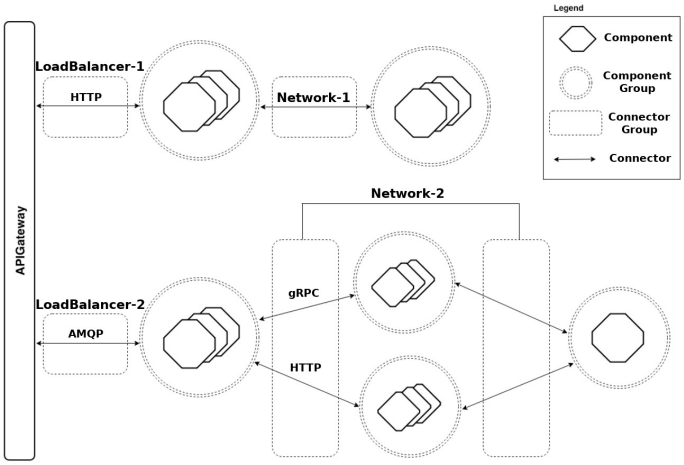

Fig. 2. Example of microservice-based system modeled with the elements of the framework

$PC : S \times A \rightarrow \{\text{true}, \text{false}\}$, which identifies the subset of the state space in which the action is executable; and by a *post-condition* function, which defines a probability distribution over the possible next states, after the execution of the action. Table I summarizes the pre-conditions, post-conditions, and reward parameters of the actions.

III. Experimental Evaluation

The objective of this work is to show that, in the domain of intrusion response, Deep Q-Learning is a valid and more effective alternative than standard planning and learning techniques, such as VI and Q-Learning. Although learning algorithms mainly target the control of a system at runtime, where they can automatically learn a model from the environment, by design they approximate the optimal q-value function. Therefore, an agent based on either Q-Learning or Deep Q-Learning asymptotically behaves like an agent that follows a policy planned with VI. As a consequence, if a model of the system can be built to plan with VI, then a simulator of the system for use with Q-Learning and Deep Q-Learning can also be built, and the agent can run either of them to reach an asymptotically optimal q-value function. The obtained result can then be used to plan optimal defense policies, in the same way it is done with VI.

In this section, we discuss two types of experiments aimed at testing both, the effectiveness and the performance of VI in comparison to Q-Learning and Deep Q-Learning. All the experiments have been executed on a Dell PowerEdge c8220 with 40 cores running at 2.20Ghz and 256 GB of RAM. The realized prototype leverages the BURLAP library [2] for the design of the domain, and for the implementation of VI and Q-Learning; our implementation of Deep Q-Learning is instead based on DL4J [4].

A. Effectiveness and Performance Evaluation

The experimental validation of the effectiveness has been performed as follows: system models with an increasing number of components, from 4 to 7 (corresponding to a number

TABLE I
ACTIONS CHARACTERIZATION

Action Name	Pre-Condition	Post-Condition	T	C
$StartComponent_{(i,j)}$	$Act_{(i,j)} = false$	$P = 1 \rightarrow Act_{i,j} = true$	200	20
$RestartComponent_{(i,j)}$	$Act_{(i,j)} = true \wedge Corr_{(i,j)} = true$	$P = 1 \rightarrow Corr_{i,j} = false$	300	50
$StartFirewall_{(i)}$	$FwAct_i = False$	$P = 1 \rightarrow FwAct_{(i)} = true$	100	5
$UpdateComponent_{(i,j)}$	$Act_{(i,j)} = true \wedge Nva_{(i,j)} = true \wedge Upd_{(i,j)} = false$	$P = 1 \rightarrow Nva_{(i,j)} = false, Upd_{(i,j)} = true$	1000	150
$HealComponent_{(i,j)}$	$Act_{(i,j)} = true \wedge Corr_{(i,j)} = true$	$P = 1 \rightarrow Corr_{(i,j)} = false$	1200	200
$FixVulnerability_{(i,j)}$	$Act_{(i,j)} = true \wedge Vul_{(i,j)} = true$	$P = 1 \rightarrow Vul_{(i,j)} = false$	700	100

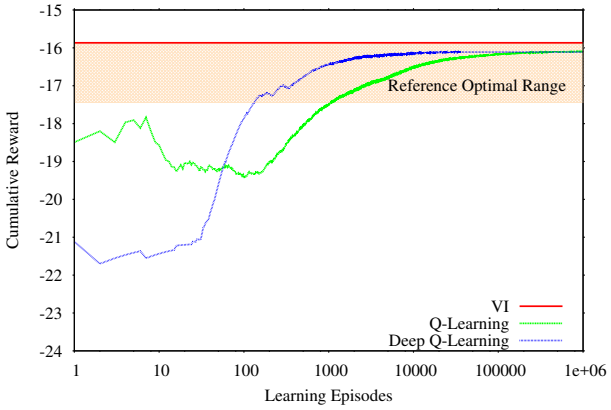

Fig. 3. Effectiveness comparison with 4 microservices (20 attributes). Log-scale is used on the x-axis.

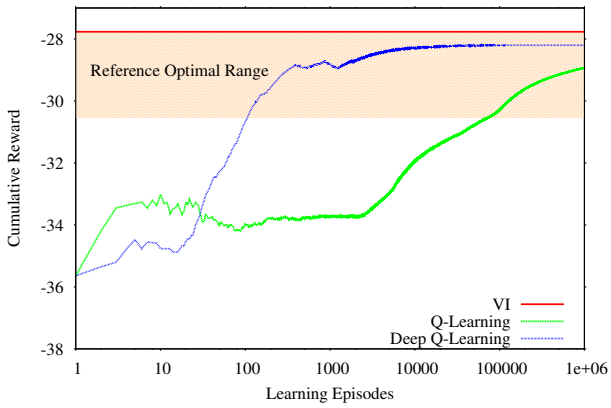

Fig. 4. Effectiveness comparison with 7 microservices (35 attributes). Log-scale is used on the x-axis.

of attributes that ranges from 20 to 35) have been generated, according to the microservices architecture described in Section II-C. VI, Q-Learning, and Deep Q-Learning have been executed on the generated models. For all the experiments, we used $\gamma = 0.9$; Q-Learning and Deep Q-Learning have been executed with the learning parameter $\alpha = 0.9$. For Deep Q-Learning, we rely on two neural networks: a current network and a stale network, as described in [19]. Both of them have 3 hidden layers with size equal to the input layer, and corresponding to the amount of attributes in the system. Finally, the size of the output layer corresponds to $|A|$, that is, the amount of executable actions.

Figures 3-4 show a comparison of the cumulative reward obtained by the three algorithms, according to the number of learning episodes. VI provides the optimal reference value, and the reference optimal range is given by the optimal value subtracted by 10%. The rationale is that, since we use a static ϵ-greedy policy for the learning algorithms, with $\epsilon = 0.9$, the minimum average cumulative reward that is possible to obtain after convergence is 90% of the optimal reward. It is possible to see that, although in every experiment both Q-Learning and Deep Q-Learning eventually converge to the optimal solution (the small gap with the optimal solution is due to the ϵ-greedy policy), Deep Q-Learning reaches the optimal region in a number of learning episodes from 1 to 3 order of magnitude smaller than Q-Learning, depending on the size of the system. In general, Deep Q-Learning is able to reach a near-optimal solution after only 100 learning episodes.

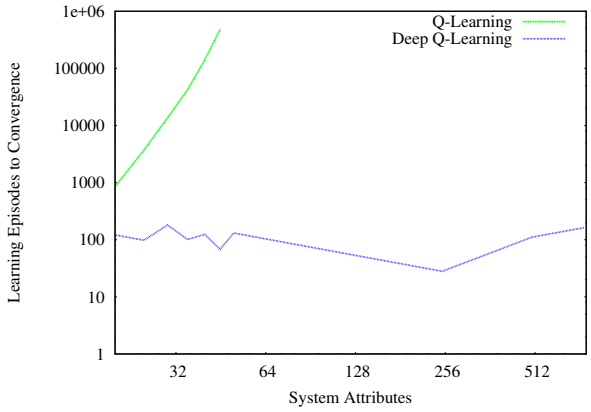

Fig. 5. Comparison of the learning steps needed to convergence. Log-scale is used on the x-axis and on the y-axis.

Furthermore, it is worth noting that, while the number of steps needed for Q-Learning to converge grows exponentially with the amount of system attributes, it exhibits a constant behavior with Deep Q-Learning, as shown in Figure 5.

Another aspect that must be considered is the memory utilization, which grows exponentially for VI and Q-Learning, and linearly for Deep Q-Learning. For this reason, we could not run any experiment with a system larger than 35 attributes with VI, which used almost 100GB of memory. The same amount of RAM was used by Q-Learning for a system with 45 attributes, due to the partial exploration of the state space.

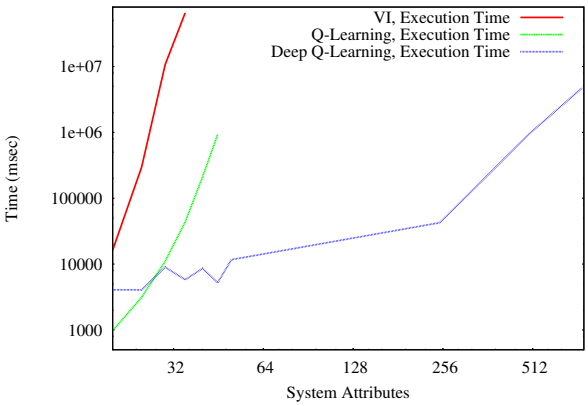

Fig. 6. Performance comparison with varying system size. Log-scale is used on the x-axis and on the y-axis.

In Figure 6, a comparison is shown between the planning time of VI and the time-to-convergence of Q-Learning and Deep Q-Learning. VI and Q-Learning exhibit the expected exponential behavior; Deep Q-Learning, instead, has a constant behavior at the beginning, followed by an exponential behavior as the size of the problem grows. This is due to the inherent scalability of the algorithm, which is capable of using all the available CPUs of the machine used for the experiments. Hence, at the beginning of the experiment, the number of used CPUs grows according to the size of the problem, until it reaches the amount of available CPUs, thus amortizing the exponential execution time. Afterwards, when all the CPUs are used, the convergence time exhibits the usual exponential behavior. However, given the extremely limited amount of learning episodes needed to reach convergence, Deep Q-Learning provides an advantage of several orders of magnitude over VI and Q-Learning.

IV. Conclusion and Future Works

The problem of automatically controlling a system to achieve self-protection exhibits an exponential complexity in time, if addressed with a stateful approach. This is one of the main limitations in the development of efficient and effective intrusion response systems. In this paper, we compared the performance and the effectiveness of standard planning and learning techniques with an approach based on deep reinforcement learning, in terms of execution time, learning episodes, and cumulative reward. We have shown that, using a single compute node with 40 cores, it is possible to efficiently compute defense policies for systems characterized by up to 750 attributes. In [7], we proposed a change of paradigm, according to which the computational complexity does not depend anymore on the size of the system, but on the scope of the attack. These two results, combined together, allow us to state that the proposed approach can be used to protect large-scale systems against large-scale attacks.

As a future work, we intend to explore the usage of Deep Reinforcement Learning on General Purpose Graphical Processing Units (GPGPUs), and on clusters of standard compute nodes, with the objective of further increasing the processing speed. Furthermore, we intend to extend the application of this approach to multi-agent systems, where multiple defenders compete against multiple attackers.

Acknowledgment

All the experiments have been executed on the NSF-sponsored CloudLab platform.

References

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
[2] Brown-UMBC Reinforcement Learning and Planning (BURLAP). http://burlap.cs.brown.edu/.
[3] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang. Nice: Network intrusion detection and countermeasure selection in virtual network systems. *IEEE Trans. Dependable Secure Comput.*, 10(4), 2013.
[4] Deep Learning for Java. https://deeplearning4j.org/.
[5] B. Foo, Y.-S. Wu, Y.-C. Mao, S. Bagchi, and E. Spafford. Adepts: adaptive intrusion response using attack graphs in an e-commerce environment. In *Proc. of Int'l Conf. on Dependable Systems and Networks (DSN '05)*, pages 508–517. IEEE, 2005.
[6] S. Iannucci and S. Abdelwahed. Model-based response planning strategies for autonomic intrusion protection. *ACM Trans. Auton. Adapt. Syst.*, 13(1):4, 2018.
[7] S. Iannucci, S. Abdelwahed, A. Montemaggio, M. Hannis, L. Leonard, J. King, and J. Hamilton. A model-integrated approach to designing self-protecting systems. *IEEE Trans. Softw. Eng.*, 2018.
[8] M. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.
[9] H. A. Kholidy, A. Erradi, S. Abdelwahed, and F. Baiardi. A risk mitigation approach for autonomous cloud intrusion response system. *Computing*, 98(11):1111–1135, 2016.
[10] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.
[11] D. Kwon, H. Kim, J. Kim, S. C. Suh, I. Kim, and K. J. Kim. A survey of deep learning-based network anomaly detection. *Cluster Computing*, pages 1–13, 2017.
[12] L. Li and M. L. Littman. Prioritized sweeping converges to the optimal value function. Technical report, Tech. Rep. DCS-TR-631, Rutgers University, 2008.
[13] E. Miehling, M. Rasouli, and D. Teneketzis. Optimal defense policies for partially observable spreading processes on Bayesian attack graphs. In *Proc. of 2nd ACM Workshop on Moving Target Defense*, 2015.
[14] E. Miehling, M. Rasouli, and D. Teneketzis. A POMDP approach to the dynamic defense of large-scale cyber networks. *IEEE Trans. Inf. Forensics Security*, 13(10):2490–2505, 2018.
[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
[16] C. Mu and Y. Li. An intrusion response decision-making model based on hierarchical task network planning. *Expert Systems with Applications*, 37(3):2465–2472, 2010.
[17] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
[18] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11):1127–1137, 1978.
[19] M. Roderick, J. MacGlashan, and S. Tellex. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*, 2017.
[20] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 2 edition, 2018.
[21] S. Tanachaiwiwat, K. Hwang, and Y. Chen. Adaptive intrusion response to minimize risk over multiple network attacks, 2002.
[22] T. Toth and C. Kruegel. Evaluating the impact of automated intrusion response mechanisms. In *Proc. of 18th Ann. Computer Security Applications Conf.*, pages 301–310. IEEE, 2002.
[23] C. Zhang, P. Patras, and H. Haddadi. Deep learning in mobile and wireless networking: A survey. *arXiv preprint arXiv:1803.04311*, 2018.