Implementation of reinforcement learning algorithms in a cyber security simulation

Nicolò Romandini
Alma Mater Studiorum - University Of Bologna
Bologna, Italy
`nicolo.romandini@studio.unibo.it`

Abstract

The goal of this work is to implement reinforcement learning algorithms in the context of a cyber security simulation. The simulation under consideration is represented by a simple game described by Elderman et al. [1]. In a network, an attacker tries to make his way through the nodes to reach the one in which sensitive data are stored. Meanwhile, a defender tries to raise the security level of the network to prevent breaches. For both players the following algorithms have been implemented: *Monte Carlo learning*, with *ε-greedy* and *Softmax* as exploration strategies, *Q-Learning*, with *ε-greedy* as exploration strategy. Furthermore, for the defender only, a *DQN* has been implemented, able to fully exploit all the information available to the player.

1 Introduction

A Clark School study estimates that a hacker attack occurs every 39 seconds [2]. This number is certainly destined to grow over time, as information technology is increasingly at the center of our world. The Covid-19 pandemic we are experiencing is a proof of this. As OpenVault reports, in the first quarter of 2020 network usage increased by 47% compared to the same period last year [3]. These data show that the Internet is becoming the main means of communication globally every year that passes. The increase in use, as expected, also leads to an intensified activity of violations. It is estimated that by 2022 the amount of data exchanged on the network will reach 396 exabytes, with approximately 4.8 billion active users [4]. Part of this information will then be stored by the companies in order to operate their services. It is precisely these data mediums that are most targeted by hackers, who try to get hold of the sensitive information present in them. This work aims to present a scenario that happens every day around the world and to exploit its context to implement different reinforcement learning algorithms.

2 The Game

As already mentioned, the game aims to simulate a now very frequent situation. An attacker has to breach a series of nodes in a network to reach the one where sensitive information are stored. A defender must try to prevent the theft. As will be shown later, the game is a stochastic game, because at each time step there is a non-zero probability that the game will end, due to the detection probability of the attacker's attacks. Moreover, the game is a zero-sum game, due to the fact that one agent's gain is equivalent to other's loss. Three main components emerge: the network and two agents that interact with it, the attacker and the defender. They are briefly described below. For any more detailed information see [1].

2.1 Network

It represents the environment in which the simulation takes place. Since two different agents coexist in it, it means that we are in the context of a multi-agent environment. There are three types of different nodes:

- The first node, *START*: it is the attacker's starting note, it can be seen as his personal computer.
- Intermediate nodes: they are nodes without sensitive data and must be hacked by the attacker to reach the final node.
- The end node, *DATA*: it is the node of the network in which sensitive data are stored. It is the attacker's target.

Each node is described by a series of values that represent the types of attack that the attacker can put into practice, the respective countermeasures of the defender and a value that represents the detection probability of an attack. To be more precise, an attacker node is defined as $n(a_1,a_2,...,a_{10})$, where each a is the attack value of an attack type on the node. A node for the defender is defined as $n(d_1,d_2,...,d_{10},det)$ where each d is the defence value of an attack type on the node and *det* is the detection value on the node. Each node consists of 10 different attack types, 10 different defence types and a detection value. Each value in a node has a maximal value of 10. The attack and defence values are paired, each pair represents the attack strength and security level of a particular hacking strategy. The detection value represents the chance that, after a successfully blocked attack, the hacker can be detected and caught. The environment is partially observable by the two agents, due to the fact that the attacker knows only the attack values and the node it is currently in, while the defender knows only the defence values and the detection values. The standard topology consists in four nodes, the *START* node, two intermediate nodes and the *DATA* node. See Figure 1. Even a small network like this creates a huge number of possible environmental states. The state of the network can be described as the combination of all attack types, defence types and detection values, so, for each node, 21 different features with 11 possible values each, from 0 to 10. With N nodes, there are 11^{N*21} possible states.

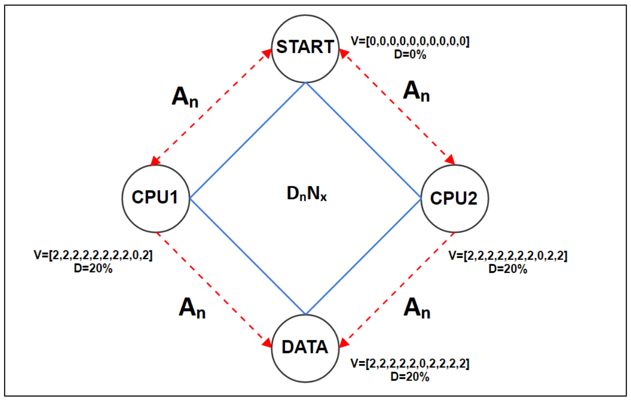

Figure 1: The standard network.

2.2 Attacker

The attacker's goal is to hack the nodes that separate him from the *DATA* node. If the latter is reached, the game would end and the attacker would win, as he would have reached sensitive data. One action for the attacker is to attack a node m from a node n with a specific attack type a_i. By doing this, the respective attack value in node n is increased by one. At each turn, the attacker can take only one action.

2.3 Defender

The defender's goal is to find the attacker. In case of detection, the game would end and the defender would win. One action for the defender is to choose a node *n* and a defence type d_i or the detection value *det*. By doing this, the corresponding defence value or detection value in node *n* is increased by one. As it is for the attacker, only one action can be done at each turn.

3 Execution

At the beginning of each episode of the game, the network is initialized, i.e. the attack values, the defence values and the detection values in all the nodes are initialized as shown in Table 1.

Table 1: Initial Values. The majority of defence types have an initial security level > 0 for most of the attack types, but there are some vulnerabilities. For example, the node CPU1 is already vulnerable to an attack of type 0.

Node	Attack Values	Defence and Detection Values
START	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
CPU1	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	[0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
CPU2	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	[2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 1]
DATA	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	[2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 1]

The attacker is in the start node. Each game step, both agents choose an action from their set of possible actions and perform them in the network. The node on which the attack value of an attack type was incremented determines if the attack was successful. If the attack value for the attack type is higher than the defence value for that attack type, then the attack overpowers the defence and the attack was successful. In this case, the attacker moves to the attacked node. If this node is the *DATA* node, the game ends and the attacker wins. When the attack value of the attack type is lower than or equal to the defence value, the attack is blocked. In this case, the attack is detected with some probability given by the detection value, a number between 0 and 10, of the node that was attacked. The chance to be detected is (detection value * 10)%. If the attack was in fact detected, the game ends and the defender wins. If the attack was not detected, another game step is played. At the end of each episode of the game, the winner gets a reward with the value 100 and the loser gets a reward with the value -100.

4 Reinforcement Learning Algorithms

The two agents involved in the simulation have to learn the best strategy to play. To do so, different reinforcement learning algorithms have been implemented: *Monte Carlo learning*, *Q-Learning* and, only for the defender, *Deep Q-learning*. All of these algorithms use also an exploration strategy chosen between *ε-greedy* and *Softmax*.

4.1 Learning Algorithms

Monte Carlo Learning In the first reinforcement learning technique, agents learn using Monte Carlo methods (Sutton and Barto [5]). These approaches are ways of solving the reinforcement learning problems based on averaging the sample returns. To ensure that well-defined returns are available, Monte Carlo methods are defined only for episodic tasks. Only at the end of an episode values are estimated and policies changed. Monte Carlo methods are used in tasks with small, finite state sets, in which it is possible to represent the state value function (or state-action value function) as arrays or tables with one entry for each state (or state–action pair). In fact Monte Carlo methods are part of the so-called tabular methods. After each game the agents update the estimated reward values of the state-action pairs that were selected during the game. Due to the fact that in this game there is only one reward given at the end, Monte Carlo methods update each state the agent visited with the same reward value, using an incremental implementation:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(R - Q_t(s,a))$$

where α is the learning rate, a value ranging between 0 and 1 that represents how much the agent should learn from a new observation. *R* is the final reward, 100 or -100. The *Q-Values* are the estimated reward values, i.e. how much the agent expects to get after performing an action. The *s* is the current state and *a* is a specific action.

Q-Learning Q-Learning is another tabular method, so all the considerations made for the Monte Carlo methods apply. The main difference between Q-Leaning and Monte Carlo methods is that the first is an "Online method", this means that it is not necessary to wait the end of the episode to be able to update the Q-Values. Thanks to this, Q-Learning can be used also in sequential environments. In our case, due to the fact that the reward is given at the end of the episode, only at that moment will it be possible to update the table. To do this, as recommended by Elderman et al. [1], an alternative version of Q-Learning is used, called *Backward Q-Learning* [6], since the update is done by starting from the last state visited first and then proceeding backwards. The last state-action pair is updated as follows:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(R - Q_t(s,a))$$

For every other state-action pair but the last one the learning update is given by:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(\gamma \max_b Q_{t+1}(s',b) - Q_t(s,a))$$

In the original Q-Learning algorithm, some reward is also used in this second formula, but, as already said, the reward is given only at the end, therefore there are not intermediate rewards. *s'* is the next state.

Deep Q-learning The last learning algorithm that has been implemented is Deep Q-learning, that is an application of Q-learning to deep learning using a deep network, called Deep Q-Network (DQN). This method can be used only by the defender, to take full advantage of the information known by the agent. In fact, the tabular methods for the defender cannot exploit the whole environmental state due to dimensionality issues. The neural network uses stochastic gradient descent back-propagation to train its weights:

$$w_j = w_j - \eta \frac{\delta J}{\delta w_j}$$

where w_j is the jth weight, η is the learning rate and *J* is the loss function. To improve efficiency and stability, the rewards are normalized to 1 for winning and -1 for losing and, for training, *experience replay* method is used. This means that the update of the network is done sampling random experiences from a buffer where they were stored, to speed up convergence. The structure of the network is described in the Section 5.3.

4.2 Exploration Strategies

ε-greedy This strategy selects the best action with probability 1 - ε and in the other cases it selects a random action out of the set of possible actions. ε is here a value between 0 and 1, determining the amount of exploration.

Softmax This strategy gives every action in the set of possible actions a chance to be chosen, based on the estimated reward value of the action. As the name suggest, the Softmax function is used in this algorithm to calculate the action-selection probabilities:

$$P_t(a) = \frac{e^{\frac{Q_t(s,a)}{\tau}}}{\sum_{i=1}^{K} e^{\frac{Q_t(s,i)}{\tau}}}$$

where $P_t(a)$ is the chance that action a will be chosen. *K* is the total number of possible actions. τ is the temperature parameter, which indicates the amount of exploration.

5 Implementation

In this section it will be shown how the game was implemented. Python was chosen as the programming language, since it is by now the de facto standard in the field of machine learning. The implementation is divided on four files:

- *simulation.py*: the main file, that controls the execution flow of the simulation.
- *environment.py*: the file in which the environment, i.e. the network, is described.
- *attacker.py*: the attacker implementation.
- *defender.py*: the defender implementation.

They will be described briefly in the next subsections. To go deeper on the implementation details, the full Python code can be found at

```
https://github.com/nickromandini/reinforcement-learning-cybersecurity/
```

5.1 simulation.py

The aim of this file is to control the simulation. In it is implemented the concept of *episode*. An episode, as described earlier, consists in a sequence of time steps, that terminates when one of the *termination conditions* are reached. At each time step, the defender and the attacker choose an action. Theoretically the two actions should be performed simultaneously. Since, for obvious reasons, this is not possible in practice, it was decided to perform the action of the defender first and then the one of the attacker. Subsequently, the data of the current and next state and of the possible rewards are stored in different arrays. This is done to update the *Q-values* or *weights* used by the various reinforcement learning algorithms at the end of the episode. There are three possible *termination conditions*: attacker detection, *DATA* node hacked or a tie. In fact, there is a small probability that the attacker will never be detected and, at the same time, never manage to hack the *DATA* node. In this condition, all attack, defence and detection values would be 10, so no action could be chosen by either agent. In case of a tie, the game must be restarted. After *N* episodes, the simulation ends by printing some statistics, such as the win percentage of both agents.

5.2 environment.py

It represents the network in which the game takes place. It maintains all the environment information, like the attack, defence and detection values, the indication if the attacker has been detected or if the *DATA* node has been hacked, the topology of the network, through the use of a matrix that indicates the neighborhood relations between the nodes (*neighbours_matrix*) and variables that stores the information about the number of nodes and the number of attack/defence values per node. It also exposes two main functions that are used to execute the defender's and the attacker's moves. When the latter is applied, a reward is returned. If the game is not over, i.e. if the attacker has not already been detected or if it has not reached the *DATA* node yet, the reward for either the attacker and the defender is 0. Instead, in case of one of the *termination conditions* occurs, the winner receives a reward of 100 and the loser a reward of -100. The reward is returned as a tuple *(attacker_reward, defender_reward)*.

5.3 attacker.py and defender.py

Since the attacker and the defender are implemented very similarly, they will be presented together. Each agent has a data structure containing the parameter values of each algorithm used. The choice of parameters was made following some of the indications given in the reference paper [1]. In it, in fact, it is specified the combination of parameters to obtain the best performance from each agent against an opponent trained using Monte Carlo method with ε-greedy strategy. The latter, instead, is optimized against an opponent that use random choice. The table 2 shows the agents' parameters.

Table 2: Optimal parameters for each algorithm.

	Attacker		Defender	
Learning Technique	Learning Rate	Parameter(s)	Learning Rate	Parameter(s)
MC ε-greedy	$\alpha = 0.05$	$\varepsilon = 0.05$	$\alpha = 0.05$	$\varepsilon = 0.1$
MC Softmax	$\alpha = 0.05$	$\tau = 5$	$\alpha = 0.05$	$\tau = 4$
QL ε-greedy	$\alpha = 0.1$	$\varepsilon = 0.04$, $\gamma = 0.95$	$\alpha = 0.05$	$\varepsilon = 0.07$, $\gamma = 0.91$
DQN			$\eta = 0.001$	$\varepsilon = 0.05$, $\gamma = 0.95$, hidden neurons = 6, batch size = 15

There are two main functions performed, the selection of an action and the update of the learning parameters. Both depend on the learning algorithm and exploration strategy chosen during agents' initialization. To simplify the use of Python classes, there are only two general methods which are *select_action* and *QValuesUpdate*, which in turn call specific functions depending on the information passed during initialization. There is only one exception, which is when using a DQN. In this case, instead of calling *QValuesUpdate*, it is called *dqn_training_step* to update the DQN's weights.

DQN Structure The Deep Q-Network is modelled as a network 3 layers: the first is a *Flatten* layer, used to flatten the *defence_values* matrix, a *Dense* hidden layer and another *Dense* output layer, with 44 output neurons, one for each possible action, i.e. increase a specific defence or detection value. In this way, the index of the highest value given in output by the network represents the nth defence/detection value that has to be increased. The only hidden layer has 6 neurons and use the *sigmoid* as activation function, as suggested in [1].

State Representation As has already been shown, all learning algorithms use some sort of state representation to work. In this case, however, the representation changes according to the method used or the agent using it. All possible representations of *current state* are shown below.

- Attacker (all methods): the state is the node it is currently in.
- Defender (tabular methods): the state is always 0. This is due to the fact that, as already said, the number of possible states is too big to be stored in a table. See the end of Section 2.1.
- Defender (DQN): the state is the sequence of all defence and detection values.

6 Results

In order to compare the results with those present in the reference paper, the same experimental parameters were adopted. In fact, 10 simulations were performed for each pair of algorithms, each of them comprising 20000 episodes. Eventually, the average of the various observations was performed. The results obtained are shown in the Table 3 below which shows the various percentages of victory/defeat of the two players with the various algorithms.

Table 3: Victory/defeat percentages. Attacker's algorithms are indicated in the first row. Defender's algorithms are indicated in the first column.

D $\downarrow$ A $\rightarrow$	Random	MC ε-greedy	MC Softmax	QL ε-greedy
Random	A: 8.5% D: 91.5%	A: 93.2% D: 6.8%	A: 96.6% D: 3.4%	A: 97.7% D: 2.3%
MC ε-greedy	A: 7.1% D: 92.9%	A: 67.1% D: 32.9%	A: 75.1% D: 24.9%	A: 48.2% D: 51.8%
MC Softmax	A: 5.2% D: 94.8%	A: 70.7% D: 29.3%	A: 79.6% D: 20.4%	A: 65.8% D: 34.2%
QL ε-greedy	A: 8.6% D: 91.4%	A: 62.2% D: 37.8%	A: 67.4% D: 32.6%	A: 42.5% D: 57.5%
DQN	A: 14.4% D: 85.6%	A: 79.8% D: 20.2%	A: 78.9% D: 21.1%	A: 97.4% D: 2.6%

The results show that in general the attacker is the best player, except for a few cases. A possible explanation for this is that the defender's victory, very often, is not directly due to actions performed by it but due to the randomness linked to the detection probability. This can be seen by analyzing the vistory/defeat probabilities when both agents are random. Many times, in fact, the defender manages to win even if he performs random actions, leading him to believe that those are good actions when they are not. In general these results are in line with those obtained in the paper by Elderman et al. [1].

Possible Optimization As can be seen from the results, the worst choice for the defender is by far the use of the DQN. The problem is that Deep Neural Networks struggle in environments with sparse rewards. In fact, since the reward is given only at the end of the episode, most of the state transitions have a reward equal to 0, so, sampling experiences from the replay buffer, it becomes very frequent to select experiences with a reward equal to 0. A possible optimization in this case could be the use of *Prioritized Experience Replay* [7], sampling more frequently the experiences that are more significant, for example, the ones that lead to a reward.

7 Conclusion

The intention of this work was to link the field of cyber security with the one of reinforcement learning in a very simple way. In reality, as seen, the central argument was precisely the latter, since we the former was not dealt in depth, using it more as a pretext to create a context. The reference for this work was the paper written by Elderman et al. [1] in which the implemented game is presented. The results obtained in this work almost confirm those described in the reference paper, but the aim was not to reach the same conclusions, but to put reinforcement learning techniques into practice in still partially unexplored areas, like cyber security.

References

[1] R. Elderman, L. Pater, A. Thie, M. Drugan, and M. Wiering. Adversarial reinforcement learning in a cyber security simulation. *International Conference on Agents and Artificial Intelligence (ICAART)*, pages 559–566, 2017. URL `https://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/CyberSec_ICAART.pdf`.

[2] Michel Cukier. Study: Hackers attack every 39 seconds, 2007. URL `https://eng.umd.edu/news/story/study-hackers-attack-every-39-seconds`.

[3] OpenVault. Openvault broadband insights report (ovbi), 2020. URL `https://openvault.com/complimentary-report-Q120/`.

[4] Cisco. Cisco predicts more ip traffic in the next five years than in the history of the internet, 2018. URL `https://newsroom.cisco.com/press-release-content?articleId=1955935`.

[5] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 2018, 2018.

[6] Y. Wang, T. S. Li, and C. Lin. Backward q-learning: The combination of sarsa algorithm and q-learnings. *Engineering Applications of Artificial Intelligence*, 2013.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silve. Prioritized experience replay. 2015.