A .NET REINFORCEMENT LEARNING PLATFORM FOR MULTIAGENT SYSTEMS

BY

ALEXANDRA CONSTANDACHE and FLORIN LEON∗

“Gheorghe Asachi” Technical University of Iaşi, Faculty of Automatic Control and Computer Engineering

Received: December 15, 2011
Accepted for publication: February 7, 2012

Abstract. Reinforcement learning is a convenient way of allowing the agents to autonomously explore and learn the best action sequences that maximize their overall value, based on successive rewards received from the environment. Among other similar libraries and platforms, the reinforcement platform presented here is especially designed to be used with the .NET framework and provides a general support for developing solutions for reinforcement learning problems.

Key words: reinforcement learning, reinforcement learning platform, intelligent agents, benchmark problems.


1. Introduction

Reinforcement learning is a subfield of machine learning inspired by behaviorist psychology, concerned with the ways in which an agent can incrementally learn an efficient decision policy in an unknown environment by trial and error, such that it should maximize its long-term utility. The only
feedback the agent receives from the environment is a numeric value known as
the reward. During the interaction with the environment, the agent learns the
utility of the states based on statistical techniques and methods of dynamic
programming.

The main characteristics of reinforcement learning are: searching and
testing different actions as well as getting the results determined by selecting the
actions. The results can be immediate or can be more obvious in the future after
many other steps.

The reinforcement learning model is composed of the environment
states \( s \in S \), the actions of the agent \( a \in A \), the rewards \( r \in \mathbb{R} \) and a
stochastic transition model \( T \) of the environment states.

The agent is connected with the surrounding environment through its
perception and action as presented in Fig. 1. At every step, the agent receives
information about the current state from the environment. Based on the
information received, the agent chooses an action \( a \). By executing the chosen
action, the current environment state is modified and the utility of this transition
is returned to the agent as the reward. Usually, the agent has to choose between
exploration, where it selects a random action, and exploitation, where it selects
the best action known so far.

A reinforcement learning problem has three essential components
(Harmon & Harmon 1999):

a) The environment – represented by states;

b) The reinforcement function – the function of future reinforcements
that the agent tries to maximize. This function is used to define the goal of the
reinforcement learning agent;

c) The value (utility) function – determines how the reinforcement
agent learns to choose the right action. A policy is the mapping from states to
actions. The state value is the sum of time-discounted reinforcements received
on the path from the state following a specific policy to a terminal state.
The agent has to discover a certain policy which maximizes the long term utility. This policy can be discovered through trial and error interactions and it indicates what action can be executed and when. The agent’s optimal behaviour is defined by the optimal policy that produces the maximum value.

Reinforcement learning is a convenient way to allow the agents to autonomously explore and learn the best action sequences that maximize their overall value, based on successive rewards received from the environment.

A multiagent reinforcement problem adds an additional level of complexity. Since classic algorithms estimate the values for each possible discrete state-action pair, each agent causes an exponential increase of the size of the state-action space. Another challenge is the implicit or explicit need for coordination, since the effect of an agent’s action also depends on the actions of the other agents. If their actions are not consistent, the overall goal may be impeded.

The environment in such a setting is no longer static, but dynamic (Russel & Norvig, 2002), because it is not only the actions of a particular agent that determine the next state of the environment, but the actions of all the other agents. Therefore, it is not enough for an agent to learn how to react best to the environment. It must also adapt to the models of the other agents. In this respect, the dynamism of the problem makes it similar in a way to the moving target learning problem: the best policy changes as the other agents’ policies change (Buşoniu et al., 2010).


The reinforcement learning concept presented in this paper is part of the BEPALIA (“Behavioural Patterns Library for Intelligent Agents Used in Engineering and Management”) project (Leon et al., 2011) whose aim is to construct a collection of algorithms suitable to represent the core of intelligent behaviour of agents in many situations. Following this approach, the developer of multiagent applications is offered an integrated library of behavioural patterns which can be reused as needed. This methodology is consistent with the recent results from artificial intelligence and cognitive psychology, which state that the human cognitive model itself is not unitary, but has a wide range of strategies, continuously adapted for handling the current tasks.

We organize our paper as follows. Section 2 describes some of the most commonly used reinforcement learning algorithms. Section 3 provides an overview of existing reinforcement learning libraries and framework, publicly available. Section 4 presents the architecture of the reinforcement learning platform. Section 5 contains several case studies, while section 6 concludes the article.
2. Reinforcement Learning Algorithms

2.1. Q-Learning

One of the most important discoveries in the domain of reinforcement learning is the development of a control algorithm through temporal difference, called Q-Learning (Watkins, 1989), in which the agent learns the best strategy even when the actions are selected according to an exploration method.

After every interaction with its surrounding environment, the agent receives a reward, changes its current state and updates the value associated to each action-state pair.

The Q-Learning updating rule is defined by the following equation (Sutton & Barto, 1998):

\[
Q(s_t, a_t) = Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]
\] (1)

In this case, the \( Q \) function will approximate the optimal action value \( (Q^*) \) without considering a specific policy used in the process of decision making. The policy determines which state-action pairs are visited and updated. The parameters used in the process of updating the \( Q \) values are the learning rate \( \alpha \) and the updating factor \( \gamma \).

The value of the learning rate can be in the interval \([0, 1]\). If the learning rate is 0, the \( Q \) values will not be updated thus the agent will not be able to learn anything. The agent increases its learning capabilities as the learning rate value is closer to 1.

The updating rate value can also be anywhere between 0 and 1. It influences the current value of future rewards. If \( \gamma = 0 \) then the agent is near-sighted and is interested only in maximizing its immediate rewards. In this case the main goal is learning which action \( a_t \) determines the highest reward \( r_{t+1} \).

2.2. SARSA

SARSA (Rummery & Niranjan, 1994) is a reinforcement learning algorithm which combines the advantages of temporal difference learning with the Monte Carlo learning methods. Its main advantage is that the agent learns during each episode. The Monte Carlo learning advantage also present in the SARSA algorithm is the possibility of using all the rewards received in each previous state to update the values of the previous state-action pair.

The name of SARSA is an acronym of the quintuple \( Q(s, a, r, s', a') \) used in the updating process where \( s \) and \( a \) represent the current state and the chosen action, \( r \) is the reward received as a response to the \( a \) action, \( s' \) and \( a' \) represent the next state and the next action chosen. Unlike Q-Learning, SARSA doesn’t use the maximum value of the next state to update the \( Q \) values, but a
new action and implicitly a new reward is selected using the policy that led to the current state.

The main idea behind this algorithm is learning the function value of an action, not the function value of a state (Sutton & Barto, 1998). The equation used in the updating process of the state-action pairs is the following:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$  \hspace{1cm} (2)

2.3. Optimization Methods

During the learning process, a large amount of data is accumulated. In practical applications this causes the learning process to be very slow. Eligibility traces are one of the basic reinforcement learning mechanisms used to accelerate learning. An eligibility trace is defined as a temporary record of the occurrence of an event. An event can be visiting a state or executing an action.

When the Q-Learning and SARSA algorithms are improved with the use of eligibility traces, they become the $Q(\lambda)$ and SARSA($\lambda$) algorithms, where $\lambda$ symbolizes the number of steps after which the Q value will be saved.

![Types of eligibility traces](image.png)

Fig. 2 – Types of eligibility traces.

An eligibility trace $e_t$ is associated to each state-action pair as a variable. The trace for each pair decays at each step with $\gamma \lambda$ ($\gamma$ is the updating factor). The trace for the state visited in the current step is modified depending on the implementation type chosen. The two implementation types are: trace accumulating, where the eligibility trace of the visited state is incremented, and trace replacing where it becomes 1, as shown in Fig. 2. In the end, the eligibility traces keep a record of the visited state-action pairs and their degree of suitability to go through learning changes.

Reinforcement learning has proved to be effective when the $Q$ values are represented in a tabular data structure and each state-action pair is visited an adequate number of times. In many practical applications, the states as well as
the actions are defined by continuous parameters, e.g. distance or velocity. In fact, memory requirements to store and access tables quickly become infeasible; moreover, visiting adequately each state-action pair is dramatically time consuming and very often impossible. These problems raise the issue of generalization: how to represent the $Q$ value function $Q(s, a)$ in a compact way and possibly reuse the collected experience in areas of the problem space scarcely or even never visited. As a result, for the case when the state space $S$ and the action space $A$ are (almost) infinite, the process of learning the $Q$ value function requires some form of function approximation.

Neural networks are a particular case of such function approximations used in reinforcement learning. One of the major successes using this approach is TD-Gammon (Tesauro, 1995), a backgammon program which reached world-class level by repeatedly playing against itself.

A successful function approximation technique used in many reinforcement learning systems is tile coding (Sutton & Barto, 1998). It combines linear approximation with an input mapping function $\phi(s)$ that translates the state $s$ into a vector of $n$ binary features ($\phi_1(s),..., \phi_n(s)$). Accordingly, the value of $Q(s, a)$ is computed as $\phi(s) \cdot \theta_a$, where $\theta_a$ is a vector of $n$ parameters associated to action $a$ (Loiacono & Lanzi, 2008).

In tile coding (Sherstov & Stone, 2004), the variable space is partitioned into tiles. Any such partition is called a tiling. The method uses several overlapping tilings and for each tiling it maintains the weights of its tiles. Typically, the tilings are all partitioned in the same way but are slightly offset from each other. Each element of the tilings called a tile is a binary feature activated if the given state/action (which is approximated) falls in the region delineated by that tile. Fig. 3 illustrates a tile coding scheme with two tilings.

![Tile coding example](image-url)
The approximate value function of a given point is found by summing the weights of the tiles, one per tiling, in which it is contained:

\[ V(s) = \sum_{i=1}^{n} b_i(s)w_i \]  

(3)

In Eq. (3) \( n \) is the total number of tiles, \( b_i(s) \) is the value (1 or 0) of the \( i \)th tile given state \( s \), and \( w_i \) is the weight of that tile.

In practice, it is not necessary to sum over all \( n \) tiles since only one tile in each tiling is activated for a given state. Given \( m \) tilings, we can simply compute the indices of the \( m \) active tiles and sum their associated weights.

In a multiagent setting, state attractors (Leon, 2011) can prove to be valuable, since agents compute their actions based on the proximity of their current state to the nearest state attractor. This representation can be used as a compact way to define individual or joint policies. The classification methods can also be used to include a learning phase into the plans of an agent, so that the agent can dynamically recognize the preconditions of an action when the states are not fully determined, and even directly choose its actions based on learning results (Leon, 2010).

3. Publicly Available Reinforcement Learning Platforms

Even though machine learning through reinforcement learning is very popular, there are not many platforms available. A typical reinforcement learning platform must provide a framework for different environments and agents.

One of the most popular is the RL-Glue (Tanner & White, 2009), which provides a standard interface that allows connecting agents, environments and experiment programs. The RL-Library (Tanner et al., 2011) creates a centralized place for the reinforcement learning community to share RL-Glue compatible software projects. It currently offers a static set of 10 environments and 3 agents. The RL-Library was created mainly for educational and research purposes. The library provides a variety of learning algorithms and problems for empirical research. The development of the RL-Glue started in 2005 and it has been constantly updated ever since.

Another framework available is the Reinforcement Learning Toolbox. It is a C++ based, open source, OS-independent framework for many reinforcement learning algorithms. This framework is the result of a Master thesis (Neumann, 2005) finished in June 2005 and it has never been updated since November 2006.

LibPGrl (Aberdeen & Buffet, 2007) is another reinforcement learning library implemented in C++. It was intended to be a high-performance policy-gradient reinforcement learning library, but it was extended to a number of
value-based reinforcement learning algorithms. It does not provide value iteration algorithm or real-time dynamic programming. Some of the main reinforcement algorithms implemented are: SARSA, Q-Learning, vanilla policy-gradient and natural actor-critic.

*PyBrain* (Schaul & Bayer, 2010) is a modular machine learning library built for Python-based applications, specialised in reinforcement learning. Among other machine learning algorithms, it contains a large number of reinforcement learning algorithms, including SARSA, Q-Learning algorithm (with or without the eligibility traces), neural fitted Q-iterations and a series of exploration methods like epsilon-greedy or Boltzmann exploration.

4. The Reinforcement Learning Platform

4.1. The General Architecture

The platform presented in this paper offers a framework for building the main components of a reinforcement learning problem: the environment, the utility, the reward function and the agent. The main packages are presented in Fig. 4.

![Fig. 4 – The general architecture of the reinforcement learning platform.](image)

Unlike the other reinforcement learning libraries mentioned in the previous section, this platform was developed for the .NET framework and offers a support for developing general solution to different problems in need of a reinforcement learning approach. The *Agent*, *Environment* and the transition model are generic elements and can be adapted to different solutions depending on the approach considered by the developer.
The architecture includes the Agent, Action and Environment packages and an optional package called FunctionApproximation. They are built based upon the general interaction model of the reinforcement learning problems.

The Environment package is used to describe the state space and to define the reward function used for calculating the utility. This package provides an interface specialised in handling the interactions between the agent and the environment.

The Action package offers an interface for handling the actions available to the agent and it is necessary for identifying the utility, depending on the algorithm used.

The Agent package defines the agent itself and provides a data structure used for storing the values of the utility function. As the agent continuously interacts with the environment either by exploring or exploiting the utility values, also known as the Q values, it must have a way of interacting with the environment. This explains the dependence between these two packages, Environment and Agent.

The FunctionApproximation package is useful in case of a large state space because it provides methods of approximating the state space.

4.2. Component Design

The ReinforcementLearningLibrary offers a support for developing various solutions for reinforcement learning problems. The Agent package available in this library offers a reinforcement learning agent developing pattern through the abstract class AbstractAgentExploreExploit, which provides a start function Run, exploration and exploitation methods, and an implementation model AbstractAgent ideal for sequential use through DoAction and Start methods. The user can choose which function to use and in what moment. The AbstractAgentExploreExploit class also provides the QLearningFunction method responsible for the computation of the new Q value depending on the specific parameters.

In the Agent package, the representation models of the Q matrix are also available. The interfaces for defining this matrix are: IQMatrixExplicitMultipleAgent, IQMatrixMultipleAgent and IQMatrixSingleAgent. The IQMatrixMultipleAgent and IQMatrixSingleAgent interfaces are generic and require the specification of the state and action representation type. The classes that define the state and action types must implement the IState and IAction interfaces. The initialization of an object derived from IQMatrixExplicitMultipleAgent requires only the specification of the reward type. The state is encoded as a long integer value, and the action as an integer value. The interfaces IQMatrixMultipleAgent and IQMatrixExplicitMultipleAgent offer a structure for using a single Q value matrix for several agents. The main operations of these interfaces are the appropriate accessor methods:
GetCellValue, SetCellValue and MaxValue for determining the maximum value of the supposed state as well as the best action. The IState interface present in the Environment package alongside the IAction interface from the Action package are used in the representation of state and action, and their encoded values used in identifying the corresponding $Q$ value are a result of the associated encoding function.

The Environment package offers a representation model of the actual environment in which the reinforcement learning agent is placed. The only limitation in using the corresponding model is that the IModel interface must be implemented with the specification of the state and action type.

The FunctionApproximator package so far provides a TileCoder class used in tile coding.

5. Case Studies

Using the reinforcement library mentioned, the solutions to three popular reinforcement learning problems were implemented. The first problem is the classic robot in a maze, where without any prior knowledge of the environment the agent has to find its docking station. The second reinforcement learning problem is elevator dispatching for multiple elevators in a building. The third case study is the Mountain Car problem (Moore, 1990; Sutton & Barto, 1998), where a driver is stuck in a valley with an underpowered car and cannot accelerate up a steep mountain road.

5.1. The Learning Process. Problem Details

The learning process uses the Q-Learning and SARSA algorithms. In the Robot in a Maze case study, the agent is able to move north, south, east, west, one cell at a time and the state space is represented by the cells on the map. Each cell can be an obstacle or it can be a clear cell. $Q$ values are stored in a matrix where the number of rows is equal with the size of the state space and the number of columns is the same as the number of actions available to the agent. The $Q$ values are calculated based on the reward received by the agent from the environment while interacting with it. The rewards returned are:

- $-5$ if the agent hits a wall or an obstacle;
b) −10 if the agent hits one of the exterior walls of the maze;  
c) 100 marks the goal state;  
d) 0 is the default reward.

As mentioned, the agent learns the appropriate policy by combining the exploration with exploitation.

For this case, the agent first explores the state space by choosing a random action until it reaches the goal state or it is stuck and cannot learn anything valuable. After the exploration stage has ended the agent begins to choose the best action based on the $Q$ values known. During these two stages the $Q$ values are constantly updated using the formula that characterizes the Q-Learning algorithm. These two stages repeat themselves until the agent reaches the goal state in the exploration stage.

Elevator Dispatching is also a classic reinforcement learning problem. Barto and Crites (1996) suggested a solution to a similar problem, in which all the traffic was down-peak. Unlike the problem solved by Barto and Crites, in this Elevator Dispatching case study the agent must learn to serve the requests in a timely manner heavy down-peak, up-peak and inter-floor traffic. There are two types of requests:

a) Internal requests that represent the requests present in the elevator car at a specific time and are characterized by the destination floor;

b) External requests that represent the requests from a certain floor and are characterized by the number of people waiting at that floor and the direction of the request. They can be:
   - Going down requests;
   - Going up requests;
   - Going both ways requests.

The actions of the agent used in learning are going up and going down. Also, the agent can park the car in its default position and serve a request.

The state space is complex and very large. A state has three components:

a) The list of external requests from the building;  
b) The current floor of each elevator car;  
c) The current travel direction of each elevator.

This case study involves an agent which tries to move efficiently two elevator cars in a seven-storey building. There were two types of reinforcement functions used in the agents learning process. One type of reinforcement function is request-based and its formula is presented in Fig. 5. The second type of function used for calculating the reward at a certain moment is time-based and the formula is represented in Fig. 6, where $T$ is the constant inter-floor travel time.
reward = 0
for each person waiting
    reward += (-1)
for each elevator shaft
    if (MaximumLoad was reached)
        reward += maximumLoadCapacity

Fig. 5 − Request-based reinforcement function.

reward = 0
for each person waiting
    if (T_{\text{waiting}} > T \cdot \text{Height})
        reward += (-20)
for each internal request
    if (T_{\text{waiting}} > T \cdot \text{Height})
        reward += (-50)

Fig. 6 − Time-based reinforcement function.

The general learning process is similar to the one in the first case study. The only difference is the way in which the agent chooses between exploration and exploitation. In this case the agent chooses to explore with a probability ε that diminishes at every step but it will never be 0. The following constraints help in simulating a real elevator system:

a) An elevator cannot skip a floor with an internal request present;

b) The elevator shaft will stop only at floors where an internal or external request exists;

c) The elevator shaft cannot stop at the same floor as another elevator shaft unless there is an internal request;

d) The elevator shaft will not stop to service an external request if the maximum load has been reached;

e) The elevator shaft cannot change the servicing direction unless it is empty.

The last problem implemented using the reinforcement learning platform is the Mountain Car problem where a driver is stuck in a valley with an underpowered car. It is assumed that the gravity is stronger than the engine of the car, and even at full throttle the car cannot accelerate up the steep mountain road. Thus the only solution is to move away from the goal at first, up the opposite slope on the left. Then by applying full throttle the car can build up enough inertia to carry it up the steep slope.

The set of possible actions is the simple list \([-1, 0, 1]\), of which the elements stand for a full throttle reverse, zero throttle and full throttle forward, respectively.
The agents’ actions available are: moving forward and backward at full throttle and stop. The state space is very large and each state variable consists of the agents’ position and velocity. The transition model is a function mapping a state $s$ and action $a$ to a next state. It returns the next state according to the following equations:

$$x_{t+1} = x_t + v_{t+1}$$  \hspace{1cm} (4) \\
$$v_{t+1} = v_t + 0.001 a_t - 0.0025 \cos(3x_t)$$  \hspace{1cm} (5) \\
$$-1.2 \leq x_{t+1} \leq 0.5 \text{ and } -0.07 \leq v_{t+1} \leq 0.07$$  \hspace{1cm} (6)

The reward is $-1$ for each step until the agent reaches the goal state when the learning stops.

The learning process is based upon the SARSA algorithm presented and the agent chooses to explore with a constant $\varepsilon$ probability.

The states and actions are defined by continuous parameters such as position and velocity. As a result, the state space is very large or infinite, and learning the value function requires some form of function approximation. The function approximation used in discretizing the state space is tile coding.

5.2. Case Study 1: Robot in a Maze. Results and Discussions

The values are updated according to Eq. (1). As mentioned, the $\alpha$ and $\gamma$ parameters influence the learning process. In the next paragraph we will study how the learning process progressed with different parameters. Three mazes will be considered: a 6x6 maze, a 10x10 maze and a 15x15 maze. The learning rate as well as the updating factor received values situated in the (0, 1] interval. The 0 value was avoided because the agent will not learn anything. After all the parameters were initialized as mentioned, the progress is monitored by counting the learning steps as well as the total number of steps executed during the exploitation stage until the agent reaches the goal state all by itself. A sample of the data collected is presented in Table 1.

Although in case of simpler and smaller mazes, the results are good if the learning factor is approximately 0.5, the use of a smaller value for $\alpha$ is recommended. According to the Q-Learning equation, if the learning rate is too high, the value of the chosen action will be determined only by the reward and the $Q$ value of the next state.
### Table 1

The Evaluation of the Learning Process for Different Maze Sizes

<table>
<thead>
<tr>
<th>Maze size</th>
<th>Value of the learning rate $\alpha$</th>
<th>Value of the updating factor $\gamma$</th>
<th>Total number of steps</th>
<th>The number of learning episodes</th>
<th>Total number of steps executed in the exploitation phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (6x6)</td>
<td>0.1</td>
<td>0.1</td>
<td>6209</td>
<td>5</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>2765</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>4061</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>0.1</td>
<td>4940</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>3913</td>
<td>6</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>3525</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>0.1</td>
<td>3322</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>4536</td>
<td>7</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>2639</td>
<td>3</td>
<td>45</td>
</tr>
<tr>
<td>Medium (10x10)</td>
<td>0.1</td>
<td>0.1</td>
<td>6216</td>
<td>5</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>9141</td>
<td>6</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>20049</td>
<td>18</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>0.1</td>
<td>15075</td>
<td>9</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>10120</td>
<td>5</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>5731</td>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>0.1</td>
<td>5521</td>
<td>5</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>10327</td>
<td>6</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>3581</td>
<td>5</td>
<td>64</td>
</tr>
<tr>
<td>Large (15x15)</td>
<td>0.1</td>
<td>0.1</td>
<td>26505</td>
<td>9</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>21524</td>
<td>10</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>31801</td>
<td>14</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>0.1</td>
<td>26500</td>
<td>5</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>14329</td>
<td>8</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>21015</td>
<td>12</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>0.1</td>
<td>15536</td>
<td>6</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>25548</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9</td>
<td>19839</td>
<td>6</td>
<td>111</td>
</tr>
</tbody>
</table>

As it can be noticed in Table 1, in case of the smaller maze better results were found by using a medium learning rate (0.5) and a high updating factor (0.9), as suggested by the fact that the total number of steps during the exploitation stage and the total number of learning episodes have a smaller value. But according to the data, this is not true in case of a larger maze, because better results were found for an updating factor equal to 0.1.
5.3. Case Study 2: Elevator Dispatching. Results and Discussions

In order to test the performance of the reinforcement learning solution for the elevator dispatching problem, the results were compared with the results of a classic elevator dispatching solution with the first floor priority and the highest floor priority met in most of the building. The system is characterized by the presence of 2 elevator shafts associated with a single external request servicing direction. The dispatching with priority to the first/highest floor consists in servicing first the requests situated closer to the more important floors (determined by the priority setting). Once an external request is serviced, the shaft will move in its characteristic direction stopping only at the destination floors of the internal requests. When there are no more internal requests, the elevator will be directed to serve the external request with more priority.

For the problem of two elevators, it was considered that the first elevator will service only the “going up” external requests and the second will service only the “going down” external requests.

The waiting time used in Fig. 8 is the time passed from the moment an external request was instantiated until it was serviced, turning the external request into an internal request. The servicing time is the time from the moment the external request was initiated and the moment the appropriate destination floor was reached. It can be said that the servicing time is the sum of the waiting
time and travel time (the time an internal request reaches its appropriate destination floor). The number of served requests is represented by the total of external requests that reached their appropriate destination. And the number of waiting requests is the number of external requests and internal requests that have not reached their destination yet.

The reinforcement learning dispatching system provided an average servicing time of maximum 38 units, with a waiting time lower than 20 units, as it can be noticed in Fig. 8. For the classic elevator dispatching system, the minimum servicing time is 50 time units and an average waiting time of 30 time units. In conclusion, in case of the classic elevator dispatching, the requests wait for at least 30 time units more before reaching their destination. That also explains the fact that the same request data are served in 80% more time than the reinforcement learning dispatching system.

Fig. 8 – Comparison between the classic and reinforcement learning elevator control system.

5.4. Case Study 3: Mountain Car. Results and Discussions

The path of the car is shown in Fig. 9, where the state of the agent is represented for each step in the episode. The line colour represents the velocity of the agent at that specific moment, and because the time step is very small, only the moving direction will be represented: the dark grey is for going forward and the light grey is for going backward. The value on the ordinate axis represents the position of the agent, where 0 represents goal position. As it can be easily seen in Fig. 9, the agent managed to learn without any prior knowledge that it has to use the inertia and first it has to move as far away as possible from the goal state to accumulate enough power and velocity to reach the goal position.

The evolution of the learning process with different values $\alpha$ and $\lambda$ for the parameters is displayed in Fig. 10. The ordinate marks the total number of
steps executed after running 30 agents which learn for a maximum of 20 episodes each. It can be easily noticed that lower values for the two parameters determine a faster learning process.

Fig. 9 – Agent position representation.

Fig. 10 – The evolution of learning with different learning rates and updating factors.

Fig. 11 represents the tile coding used in this problem. In the solution, 10 tilings were used, each characterized by the two state features: position and velocity. For calculating the active features for the selected state, the position and the velocity were divided in 9 sections with a width of 0.1(8) and 0.01(5), respectively.

Fig. 11 – Tile coding representation for the Mountain Car problem.
The tile coding function used in determining the active features is found in the *TileCoder* class included in the *FunctionApproximation* package. The implementation of this function is inspired by the tile coding with hashing *C* function (Sutton & Barto, 1988).

4. Conclusions

Although it is only a prototype, our platform provides a rather general support for developing solutions for reinforcement learning problems, unlike the publicly available reinforcement learning libraries. The generality can be noticed in the different implemented case studies.

As expected, the reinforcement learning simulations developed using this platform demonstrated that on the long run the reinforcement learning systems are better, shown for example by the comparison between the classic elevator dispatching system and the reinforcement learning one. The only downfall is that the system must be trained appropriately.

In the Mountain Car and in the Robot in a Maze case study, the system behaved as expected but better performance can be reached only by trying various values for the reinforcement learning parameters and choosing the most suitable ones, as shown by the obtained results.

REFERENCES


PLATFORMĂ .NET DE ÎNVĂȚARE CU ÎNTĂRIRE PENTRU SISTEME MULTI-AGENT

(Rezumat)

Învățarea cu întărire este o modalitate convenabilă de a permite agenților să își exploreze în mod autonom mediul de execuție și să învețe secvențele de acțiuni optime care să le maximizeze utilitatea totală, pe baza recompenserilor succesive primite din mediul său. Spre deosebire de alte platforme și biblioteci similare, platforma de învățare cu întărire prezentată în acest articol este special proiectată pentru platforma .NET și oferă posibilitatea dezvoltării de soluții generale pentru probleme de învățare cu întărire.

Multe dintre problemele complexe cu care se confruntă oamenii sunt aparent fără soluție și din cauza faptului că este foarte dificilă anticiparea acțiunilor necesare pentru rezolvarea lor. În aceste situații, pentru a putea furniza soluția necesară, aplicația ar trebui să învețe singură pe baza interacțiunilor cu mediul înconjurător prin încercări și erori. Faptul că experiența este uneori mai importantă decât inteligența nativă a fost postulat de mulți psihologi, filoșofi și acum este demonstrat și de către cercetătorii în inteligență artificială ce preferă învățarea cu întărire în defavoarea altor abordări.

Avantajul principal al acestui tip de învățare este capacitatea agentului de a reacționa în cazuri neobișnuite după cum se observă în cadrul studiului de caz Mountain Car, agentul fiind capabil să descopeze de unul singur că pentru a depăși panta trebuie să acceleereze înapoi, deci să se depărteze aparent de scop. Pe lângă această capacitate de adaptare, învățarea cu întărire poate oferi și îmbunătățiri ale preformanțelor în comparație cu sistemele clasice. Și nu în ultimul rând, învățarea cu întărire este naturală, fiind inspirată din comportamentul ființelor superioare.