LEGO© MINDSTORMS NXT AND Q-LEARNING: A TEACHING APPROACH FOR ROBOTICS IN ENGINEERING

A. Martínez-Tenor, J. A. Fernández-Madrigal, A. Cruz-Martín

Systems Engineering and Automation Dpt., Universidad de Málaga–Andalucía Tech (SPAIN)

Abstract

Robotics has become a common subject in many engineering degrees and postgraduate programs. Although at undergraduate levels the students are introduced to basic theoretical concepts and tools, at postgraduate courses more complex topics have to be covered. One of those advanced subjects is Cognitive Robotics, which covers aspects like automatic symbolic reasoning, decision-making, task planning or machine learning. In particular, Reinforcement Learning (RL) is a machine learning and decision-making methodology that does not require a model of the environment where the robot operates, overcoming this limitation by making observations. In order to get the greatest educational benefit, RL theory should be complemented with some hands-on RL task that uses a real robot, so students get a complete vision of the learning problem, as well as of the issues that arise when dealing with a physical robotic platform. There are several RL techniques that can be studied in such a subject; we have chosen Q-learning, since is a simple, effective and well-known RL algorithm.

In this paper we present a minimalist implementation of the Q-learning method for a Lego© Mindstorms NXT mobile robot, focused on simplicity and applicability, and flexible enough to be adapted to several tasks. Starting from a simple wandering problem, we first design an off-line model of the learning process in which the Q-learning parameters are studied. After that, we implement the algorithm on the robot, gradually enlarging the number of states-actions of the problem. The final result of this work is a teaching framework for developing practical activities regarding Q-learning in our Robotics subjects, which will improve our teaching.

Keywords: Reinforcement Learning, Q-learning, Robotics, Lego© Mindstorms NXT.

1 INTRODUCTION

Nowadays, robotic applications are moving from specific realms, like industry, to everyday situations, like house cleaning; in a not too distant future, Robotics is expected to be a pervasive discipline. Today students must be prepared to face such perspectives, and thus, many engineering degrees and postgraduate programs now include Robotics as a part of their curricula.

Robotics encompasses several subfields. One of these more specialized areas is Cognitive Robotics, which tries to provide the robot with some high-level human-like abilities, like decision-making, which involves mechanisms that represent cognitive processes capable of selecting a sequence of actions leading near-optimally to a specific outcome. Reinforcement Learning (RL) is a decision-making methodology well suited to be part of a Cognitive Robotics subject. From a teaching point of view, RL theory should be combined with practices, so the students could see the nuts and bolts of such methodologies. This could be done by programming RL algorithms in some language or software, like Matlab©, but, since we are in the robotics realm, the problem to be addressed should include a real robot. In this way, students get a complete vision of the robotic learning problem, as well as a first-hand experience of the issues that arise when dealing with a physical robotic platform.

Since 2008 we have been using the Lego© Mindstorms NXT [1] robot as an educational tool in several engineering subjects, both at undergraduate and graduate levels. Elsewhere [2], we have claimed that these robots properly perform in teaching tasks, and that they offer many more advantages (software/firmware robustness, variety of sensors, PC and robot-to-robot communications, diverse programming languages...) than drawbacks (weak mechanic structure, lack of sensory precision, computational limitations...) Gathering RL techniques and Lego© robots would be a proper way of helping the alumni to understand some core topics in real robotics applications.

There are several RL methodologies that could fit a Cognitive Robotics syllabus. Q-learning is a simple, effective and well-known RL algorithm. However, it has a major difficulty: the learning process

of this algorithm is often incomplete or fall into local maxima if its parameters are not well tuned; in a real scenario, this problem is worsen due to real-time constraints. Setting the Q-learning parameters in a real application is usually quite different from simulation tuning.

In this paper we present a minimalist implementation of a Q-learning method for the Lego© NXT robot, developed as a Master Thesis for obtaining the Master Degree in Mechatronics Engineering at Universidad of Málaga [3]. Our implementation is focused on simplicity and clearness, and it is flexible enough to be adapted to learning complex tasks; its natural next step is to be used for teaching practices at class. Several benefits can be derived from this NXT implementation:

• The Q-learning theory explained at lectures is fully connected to the designed practice environment.

• Starting from a simple wandering problem, we have designed a simulated off-line Octave/Matlab© model of the learning process in which the Q-learning parameters can be thoroughly studied; the role and influence of each one, as well as their interactions, can be analyzed by the students.

• We have also implemented this solution on the robot. Students can find three Q-learning wandering problems coded in the programming language of the NXT, as well as some useful code that eases the data logging for posterior analysis. Since this Lego© educational platform has several computing limitations, we have studied the effects that those constraints have on the Q-learning initial algorithm.

The paper is organised as follows. Section 2 presents some relevant works concerning Q-learning and Lego Mindstorms NXT©. Section 3 covers the basics of the Q-learning algorithm, so the work presented in the rest of the paper can be understandable. Section 4 is devoted to the off-line simulation that leads to the set of parameters used for our Q-learning implementation. Section 5 deals with implementation details, namely the limitations of the CPU of the robot, and the coding of three wandering learning processes. Conclusions and future work are commented on Section 6.

2 RELATED WORKS

We provide in the following a review of relevant scientific publications on Q-learning implementations on Lego© Mindstorms NXT robots:

• Obstacle avoidance task learning [4]. This paper presents the design of a learning task with 36 encoding states and 3 actions employing 1 ultrasonic and 2 contact sensors. A comparative study shows better results with traditional Q-learning than Q-learning with Self Organizing Map.

• Line follower task learning [5]. Q-learning algorithm implemented in Matlab© using USB communication with the robot. The problem statement includes 4 states encoding 2 light sensors and 3 available actions. This design includes an ultrasonic sensor for reactive behaviour only.

• Walking and line follower tasks learning [6]. They present a SARSA algorithm on the NXT using 4 legs moved by 2 servomotors and 1 light sensor directed to a grey-gradient floor for a walking task; for the line follower task, a 4 wheels vehicle setup with 2 light sensors is used.

• Phototaxis behaviour [7]. They implement three different layers on the NXT robot: low level navigation, exploration heuristic and Q-learning layer. Almost every function of the robot is executed in an external computer communicated with the robot by Bluetooth. This setup includes 11 encoding states based on 2 light and 2 contact sensors, and 8 actions representing wheel power levels.

Also, there is a growing interest in extending and combining these techniques, resulting in new methods such as hierarchical Q-learning, neural Q-learning, RL and partially observable Markov decision process (POMDP), Peng and Naive (McGovern) Q-learning, policy gradient RL, emotionally motivated RL, quantum Inspired RL, quantum parallelized hierarchical, Monte Carlo and TD methods combination, curiosity and RL, and so on.

3 Q-LEARNING BASICS

A very short introduction to the reinforcement learning theory and the Q-learning algorithm makes up this section; we refer the reader to [3], [8] for more details.

3.1 Model-based decision-making processes

To begin with, it is necessary to introduce the concept of Markov Decision Process (MDP), defined as a Dynamic Bayesian Network (DBN) [9] whose nodes define system states, and arcs define actions along with their associated rewards. Formally, an MDP is a tuple *(S, A, T, R)*, where *S* is a finite set of states, *A* a finite set of actions, *T* the transition function, and *R* the reward function defined as $R:(S\times A\times S')\rightarrow\mathbb{R}$.

An important concept in machine learning with MDPs is the one of *policy*. We call policy to a function $\pi:S\rightarrow A$ defining the action *a* to execute when the agent is in the state *s*. Thus, a policy is the core of the decision-making process. The term *policy value* or V_π indicates the goodness of a policy measured through the expected rewards when it is executed. We will use as a policy value the total expected discounted reward for our reinforcement learning problem, in which the policy value is the expected reward gathered after infinite steps of the decision process, decreasing the importance of future rewards as we move forward in time. It can be formally expressed as:

$$V_\pi(s_0)=E[R(s_0,\pi(s_0),s_1)]+\gamma\cdot E[R(s_1,\pi(s_1),s_2)]+\gamma^2... \quad (1)$$

being $\gamma\in(0,1)$. We can also define Q_π as the expected total reward from taking a given action in a given state, using *succ* to refer to the set of states that can be reached from another state:

$$Q_\pi(s,a)=\sum_{s'\in succ(s)}T(s,a,s')\cdot E[R(s,a,s')]+\sum_{s'\in succ(s)}\gamma\cdot T(s,a,s')\cdot V_\pi(s') \qquad (2)$$

It can be demonstrated that at least one optimal policy π^* exists that has the greatest value V_π^*. The following expressions will serve to calculate both the optimal policy value and the optimal policy:

$$V_\pi^*(s_0)=\max_a Q_\pi^*(s_0,a) \quad (3)$$

$$\pi^*(s_0)=\arg\max_a Q_\pi^*(s_0,a) \quad (4)$$

These expressions, called the Bellman equations, can be used recursively for improving an arbitrary initial policy. Practical implementations involve an intermediate definition: the expected reward of executing action a when the agent is in state *s*:

$$R(s,a)=E[R(s,a,s')]=\sum_{s'\in succ(s)}T(s,a,s')\cdot[R(s,a,s')] \qquad (5)$$

A classic algorithm for finding optimal policies using the Bellman equations is the *Value iteration* algorithm. This is the algorithm we have chosen for our implementation; in every step, it computes $Q_k(s,a)$ and $V_k(s)$.

3.2 Model-free decision-making processes. The Q-learning algorithm

In this subsection we will derive the Q-learning algorithm we have implemented in our robots, which is based on the MDP formulation of section 3.1 but without knowing *T*. We start expressing (5) as:

$$R(s,a)=E_{s'\in succ(s)}[R(s,a)]\equiv E_{s'}[R(s'\mid s,a)] \qquad (6)$$

Hence, $Q_k(s,a)$ has the same behavior as an expectation:

$$Q_k(s,a)=E_{s'}[R(s'\mid s,a)]+\gamma E_{s'}[V_\pi(s')] \qquad (7)$$

We will use this expression for deciding what to do when the model is not available, that is, when there is no information about the transition function *T(s,a,s')*, but we can estimate it from the real system; in other words, the lack of a model is compensated by making observations:

$$\hat{Q}_k(s,a)=\frac{\sum O_i}{n}, O_i=R(s'\mid s,a)+\gamma\cdot\hat{V}_{k-1}(s',a) \qquad (8)$$

This is the basis of the reinforcement learning concept. Expression (8) is similar to a batch (off-line) sample average of observations gathered from the real system. However, a RL problem is an on-line process evolving from an initial policy to a pseudo-optimal policy by executing sequentially actions, making observations, and updating its policy values. Therefore, we need a recursive estimation of the average. We use the one explained in [3], which leads us to:

$$\hat{Q}_k(s,a) = (1-\alpha_k)\cdot\hat{Q}_{k-1}(s,a) + \alpha_k[R(s,a,s') + \gamma\cdot V_{k-1}(s')] \qquad (9)$$

being $V_{k-1}(s') = \max_{a'}\hat{Q}_{k-1}(s',a')$ according to Bellman equations, and $\alpha_k = \frac{1}{k}$.

Equation (9) represents the general form of the Q-learning algorithm, and Fig. 1 shows the same algorithm expressed as pseudo-code.

It should be noticed that the success of a RL process depends on the accurate choice of its parameters, summarized in the following; a more detailed explanation can be found in [10].

• *Reward*. The task we want an agent to learn in a RL problem is defined by assigning proper rewards as a function of the current state, the action executed, and the new reached state.

• *Exploitation/exploration strategy*. It decides whether the agent should exploit its current learned policy or experiment unknown actions at each learning step.

• *Learning rate (α)*. It establishes how new knowledge influences the global learning process. When α=1, only brand new knowledge is taken into account (the one of the current step). On the contrary, when α=0, new actions do not affect the learning process at all (no learning). Both extreme cases mean no evolution in the learning process.

• *Discount rate (γ)*. It regulates the look-ahead of the learning process by determining how much the current reward will influence the elements of *Q(s,a)*, usually called the *Q-matrix*, in the next steps. When γ=0, only immediate rewards are taken into account, hence our agent will only be able to learn strategies with a sequence of one step. On the other hand, when γ values are close to 1, the learning process will allow strategies with larger sequences of steps, increasing learning time.

```
% Q-learning algorithm parameters
    N_STATES, N_ACTIONS, INITIAL_POLICY
% Experiment parameters
    N_STEPS, STEP_TIME

% Variables
    s,a,sp   % (s,a,s')
    R        % Obtained Reward
    alpha    % Learning rate parameter
    GAMMA    % Discount rate parameter
    Policy   % Current Policy
    V        % Value function
    Q        % Q-matrix
    step     % Current step

% Initial state
    V = 0, Q = 0, Policy = INITIAL_POLICY
    s = Observe_state() % sensors-based observation

    for step = 1:N_STEPS  %---------------------  Main loop --------------------

        a = Exploitation_exploration_strategy()

        robot_execute_action(a), wait(STEP_TIME)

        sp = observe_state(), R = obtain_Reward()

        Q(s,a) = (1-alpha)*Q(s,a) + alpha*(R+GAMMA*V(sp))

        V(s)_update()
        Policy(s)_update()
        alpha_update()
        s = sp  % update current state
    end  %-------------------------------------- Main loop end-----------------
```

Fig. 1: Q-learning algorithm

4 OFF-LINE ANALYSIS

This section addresses the study of the Q-learning algorithm in a simulated scenario for a simple obstacle-avoidance wandering task, from now on called *wander-1*. It introduces the physical configuration of the robot, the MDP model of the system, and an analysis of the parameters of the Q-learning algorithm. The numerical computation platforms Octave and Matlab© were employed for simulations in this stage of the work.

4.1 Mobile robot configuration

We have used a standard differential drive base assembly for our robot. This setup includes two driving wheels separated 11cm, a caster wheel, an ultrasonic sensor, and two contact sensors located on the front side (Fig. 2).

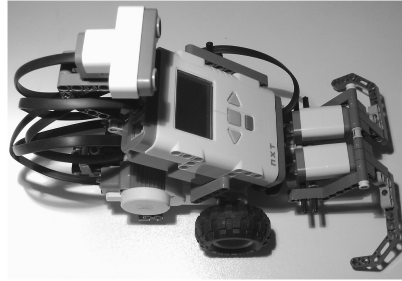

Fig. 2: NXT setup for our experiments

The following software was employed when working with the robot:

• NBC 1.2.1.r4 and NeXTTool 1.2.1.3. under Ubuntu 12.10.

• Bricx Command Center v3.3 (Build 3.3.8.9) [11] under Windows XP (just for verifying that the tests presented in this work can be reproduced in other platforms with the same source code).

• The code for the robot has been written in Not eXactly C (NXC), a programming language invented by John Hansen strongly based on C [12].

4.2 *Wander-1* task definition: 4 states+4 actions

With 2 frontal contact sensors (the sonar is not needed here), four states and four actions (see Table 1), the *wander-1* task seeks that the robot learn how to wander dodging obstacles after touching them. We expect that in a state with a left front contact activated, the robot learns that turning to the right is the action that will lead to the optimal policy. Regarding the environment, a 70x70 cm enclosure has been used.

Table 1: *Wander-1* states and actions

States		***Actions***	
s1	No bumper pressed	a1	Stop
s2	Left bumper pressed	a2	Left-turn
s3	Right bumper pressed	a3	Right-turn
s4	Both bumpers pressed	a4	Move forward

4.3 Models

An estimation of the rewards and the transition function is necessary so as to simulate a realistic RL process. Both are described below, and have been filled from tests performed on the real robot.

4.3.1 Reward function

In each step, we will measure the relative increase of the encoder of the wheel of the Lego© robot, assigning larger rewards when both wheels rotate forward a greater angle; the lack of forward motion when the robot collides is interpreted as an absence of positive reward, leading the learning process to select another action. However, preliminary tests showed that once the robot collided with obstacles fixed on the ground, the chance of keep rotating both wheels forward, sliding on the floor, while receiving positive reward, was very high. Thus we added a soft penalization when one bumper is activated and a large penalization in case both contacts are on. In those situations, any positive reward due to the servomotor encoders measurements is rejected.

4.3.2 Transition function and system model

The off-line model has been implemented as a function that simulates the execution of the selected action *a* from the state *s*, resulting in a new reached state *s'*, corresponding to the transition function *T(s,a,s')*. Some NXC code was written for collecting the necessary data for designing a realistic frequentist model of the robot in its environment. For that, at each step an action was selected according to a discrete uniform distribution. After compiling and running the code, we obtained a file containing the statistical information. Then, a simple algorithm was used to generate the function from these data.

4.4 Simulation: Q-learning parameters tuning

For implementing the simulation, we have to translate the Q-learning pseudo-code in Fig. 1 to a script in our simulation language. Since any Q-learning parameter can affect the behaviour of the rest, we have settled the reward values and the exploitation-exploration strategy before tuning the learning rate α and discount rate γ. This allows us to obtain the optimal values of α and γ by performing experiments under the same conditions; a suitable set of rewards that satisfy the requirements discussed in subsection 4.4.1 have been settled.

4.4.1 Reward tuning

The rewards assigned to the *wander-1* task were normalized in the range [0,1]. Since the goal of the task is defined by the reward function, we must be careful in the choice of these rewards to avoid falling into local maxima [10]. Thus, the highest reward, obtained when the robot moves forward without colliding, is ten times higher than the reward assigned when the robot turns. After a number of simulations, the reward values of Table 2 were chosen as the best ones.

Table 2: Adjustment of rewards

R	*Event*
1	Move forward (no collision)
0.1	Turn (no collision)
-0.2	One bumper collides
-0.7	Both bumpers collide

4.4.2 Exploitation-exploration strategy

We have selected a fixed ϵ-greedy strategy, meaning that the agent will usually exploit the best policy learned at each step but with a probability ϵ of exploring any action randomly. The selected value ϵ=30% has allowed the robot to explore all possible combinations of states and actions in all the tests.

4.4.3 Learning rate and discount rate tuning

The number of steps needed to learn a suitable policy has a strong dependence on the first states that are explored, sometimes resulting in the inability of learning a simple policy even after hours. Therefore we have chosen a decreasing profile for α_k. From simulations and tests [3], we detected two possible combinations of α and γ: [α=0.02, γ=0.8] and [α=0.02, γ=0.9]. As we are interested in learning more complex tasks, we have preferred to work with higher discount rates, which allow the robot to

have a longer look-ahead horizon in its strategies. To sum up, we have chosen the parameters of Table 3.

Table 3: Best Q-learning parameters selected after simulation tests

Parameter	***Value***
Rewards	{1, 0.1. -0.2, -0.7}
Exploitation/exploration	Є-greedy, 30% exploration
Learning Rate α	0.02
Discount Rate γ	0.9

5 IMPLEMENTATION IN THE ROBOT

The present section describes the transition between the simulation of the Q-learning algorithm and its implementation on the real robot. It begins by describing the limitations of the platform; then, the results of three wandering learning tasks in the robot (*wander-1*, *wander-2*, and *wander-3* tasks) are shown. For this part of the work, several NXC libraries and programs have been prepared.

5.1 Q-learning algorithm with CPU limitations

5.1.1 Fixed-point notation

The NXC programming language supports real numbers, which are necessary for the learning algorithm; however, we have found that sometimes this feature is not very stable. This is why the expression from the theoretical equation (9) used in Octave/Matlab© simulation was translated to:

$$Q[s][a]_{FP} = \frac{(FP-\alpha_{FP})\cdot Q[s][a]_{FP}}{FP} + \frac{\alpha_{FP}\cdot (R_{FP} + \frac{\gamma_{FP}\cdot V[s]_{FP}}{FP})}{FP} \quad (10)$$

where *FP* is a constant used to emulate real numbers with integers in fixed-point notation. After a thorough analysis, we confirmed that *FP*=10000 is valid to perform our experiments. Since *R*, α, and γ are real, all of them must be expressed as integers multiplied by *FP*, as well as *Q[s,a]* and *V[s]*.

5.1.2 Overflow analysis

The NXT robot has a 32-bit CPU, therefore NXC signed long variables can store values up to $2^{31}-1$=2.147.483.647=*MaxLongNXC*. This integer capacity will be limited in our notation system so as the integer part of any cell in Q must be ≤*MaxLongNXC*/FP, because fixed point variables are already multiplied by FP. The results of this analysis are summarized in Table 4, where $Q_{max}/(FP\cdot R)$ represents the value to which the Q-matrix converges; different values of parameter α for a fixed γ resulted in the same value of convergence, thus α has been left out of the Table 4.

Table 4: Limits in the Q-matrix values to avoid overflow when using a fixed-point notation system.

γ	0.5	0.6	0.7	0.8	0.9	0.95	0.99
$Q_{max}/(FP\cdot R)$	2	2.5	3.33	5	10	20	100

The data obtained in Table 4 along with the restriction Q ≤ *MaxLongNXC*/FP allow us to determine whether our system will cause overflows for any set of γ, *R* and *FP*.

5.1.3 Precision Analysis

After comparing two simulation data sets [3], one on Octave with 64-bit floating-point, and one performed in the NXT with 32-bit fixed-point, we detected that the precision differences were so small that did not affect the learning process at all.

5.1.4 Memory limits

Taking into account the nature of the problem and the robot, our conclusion is that the amount of states and actions will not represent any problem in this work: the most complex task we have implemented has 16 states and 9 actions, which is less than 2% of the total memory available.

5.2 Implementation results

At this point, we have obtained all the information about the parameters that best suit the robot, as well as their restrictions due to its computational limitations. This section explains what happens when implementing three RL tasks on the robot; the interested reader may consult the video in [13], where the robot performs these learning tasks in real scenarios.

5.2.1 Q-learning algorithm in NXC

The Q-learning pseudo-code shown in Fig. 1 has been translated to the NXC language. Any Q-learning implementation developed in the robot should follow that template, regardless of the task to be learned. This offers the students the possibility of analysing not only wandering, but any other ideas they may be interested in, just changing two constants (*N_STATES* and *N_ACTI*ONS) and two functions *robot_execute_action()* and *observe_state()*.

5.2.2 *Implementation results of task* wander-1

Task *wander-1* was explained on section 4 for simulation purposes; Table 5 collects its parameters and results on the robot.

Table 5: Parameters of the *wander-1* task implementation and learned policy

Parameter	***Value***
Robot Speed	80 (in a range [0,100])
Number of steps	2000
Exploitation/exploration	Є-greedy, 30% exploration
Learning Rate α	0.02
Discount Rate γ	0.9
FP	10000 (4 decimals in fixed-point notation)
Q-matrix cell size	4 bytes (long)

State	***Learned Action***
s1 (no obstacle)	a4 (move forward)
s2 (left contact)	a3 (right turn)
s3 (right contact)	a2(left turn)
s4 (front contact)	a2 or a3

5.2.3 *Implementation results of task* wander-2

Another wandering task has been proposed for taking advantage of the ultrasonic sensor. As in the *wander-1* task, we expect that in a state with a right-front contact activated, the robot learns that turning to the left will lead to its optimal policy, but now, if a wall or obstacle is detected close to the robot, a turning action will be most likely the best decision to achieve the optimal policy. Neither rewards nor actions change in this new setting, just a fifth state has been added that is reached when the sonar measures a distance above a given threshold with no collisions.

After a few tests, we detected some accuracy problems from the ultrasonic sensor that may affect the learning process (real maximum length detected by the sonar, obstacle angle detection), which were addressed properly [3]. The parameters employed in *wander-2* were the same as in *wander-1*, except for the speed, that was decreased to 50; the learned policy is shown in Table 6.

Table 6: Policy learned by w*ander-2* task

State	***Learned Action***
s1 (near obstacle)	a2 or a3 (left or right turn)
s2 (left contact)	a3 (right turn)
s3 (right contact)	a2(left turn)
s4 (front contact)	a2 or a3 (left or right turn)
s5 (far obstacle)	a4 (move forward)

The present task has been tested in two scenarios:

• Scenario A: The same 70x70cm square enclosure for *wander-1*.

• Scenario B: 105x70cm enclosure with a 35x8cm obstacle.

The learning process of the *wander-2* task in scenario A reached the optimal policy in an average number of steps of 531 (133 seconds) ranging from 128 (32s) to 1154 steps (289s). The learning process, although successful in the experiments, needed almost three times more iterations than the learning process of *wander-1*; besides, the dispersion of the results found in *wander-2* was larger. However, the number of states and actions used so far is so small that the maneuvers of the robot lacked precision, especially when avoiding certain corners. The learning process cannot improve this behaviour unless we increase the number of states and actions.

5.2.4 Implementation results of task wander-3

The final task, *wander-3*, addresses a more complex situation with 16 states (4 distance ranges from the ultrasonic sensor, combined with the 4 possible combinations of the contact sensors) and 9 actions (each wheel is able to move forward, move backward or remain stopped independently of the other wheel), which involves a large increment in the size of the Q-matrix up to 144 elements. Our goal in this task is that the robot learns to wander avoiding obstacles with smoother and more precise movements than in previous tasks. New distance ranges of the sonar have been used; the rest of the parameters remain the same as in the previous *wander-2* task. We executed the tests for 2000 and 15000 steps (> 1 hour in real-time) of learning. It has been necessary to change the exploitation/exploration strategy so as to avoid that some actions remained unexplored after the learning process; now, when exploring, there is a 60% probability of selecting the least explored action for the current state.

The experiments were placed on three different scenarios:

• Scenarios A and B: Same as in both *wander-1* and/or *wander-2*.

• Scenario C: A 200x125cm enclosure with four 135º corners, two 90º corners, and 2 small obstacles.

No single policy was learned for this task, but some remarkable conclusions can be drawn:

• Tests performed in scenario A showed that the time needed for learning an accurate policy was much longer than in the two previous tasks. This behaviour was expected, since the number of states and actions is now higher.

• States where both frontal bumpers collide while no near obstacle was detected did not learn an accurate action, which is reasonable.

• The learning processes of 2000 steps in scenario C generally gave better results than the previous tests in scenario A with more than 9000 steps. We thus can state that larger and more complex scenarios offer better models of learning for the *wander-3* task than a minimalist simple scenario. Scenario A had many redundant states in the *wander-3* setup, making the learning inefficient.

• Although a basis for a correct policy was reached in 2000 steps, when passing from 2000 to 15000 steps the selected actions evolved into a more accurate policy.

• Every test achieved sequences of movements (i.e., strategies), including turns and backwards displacements that prevented the robot from getting stuck.

• Moving forward usually resulted in anything but a straight path: the robot made small turns to the right; the lower the battery, the higher the drift. This can justify why in most of the experiments of *wander-2* and *wander-3* the learned policies circle the scenarios clockwise.

6 CONCLUSIONS AND FUTURE WORK

In this work we have presented a framework that adapts Q-learning, a well-known RL algorithm, to a teaching environment for Robotics subjects like Cognitive Robotics. Its utility is twofold: on one hand, it provides a simulation environment that allows a thorough analysis of the different parameters related to the Q-learning methodology, so the students can understand its meaning and role; on the other hand, it offers a set of programs written specifically for the Lego© Mindstorms NXT robot, and thus adapted to its computing limitations, where the robot learns how to wander in a closed environment as it avoids obstacles using the Q-learning algorithm. The result is a complete framework that helps the students to fill the gap between theory and practice, since they can test the theoretical aspects of Q-learning to a real robotic platform.

Up to this moment, we have designed and implemented the framework. Our next step is to use it in our classes and evaluate quantitative and qualitatively the benefits (and, if it is the case, the disadvantages) for our students.

REFERENCES

[1] Lego© Mindstorms (2014). http://mindstorms.lego.com

[2] Cruz-Martín, A., Fernández-Madrigal, J. A., Galindo, C, González-Jiménez, J., Stockmans-Daou, C., Blanco, J. L. (2012). A Lego Mindstorms NXT approach for teaching at data acquisition, control systems engineering and Real-time systems undergraduate courses. Computers and Education, Vol.5, issue 3, pp. 974-988. Elsevier.

[3] Martínez-Tenor, A. (2013). Reinforcement Learning on the Lego Mindstorms NXT Robot. Analysis and Implementation. Master Thesis. Systems Engineering and Automation Department. Universidad de Málaga.

[4] Ferrer, G.J. (2010). Encoding robotic sensor states for q-learning using the self-organizing map. J. Comput. Sci. Coll., 25(5):133–139.

[5] Cruz-Álvarez, V.R., Hidalgo-Peña, E., Acosta-Mesa, H. G. (2012). A line follower robot implementation using Lego's Mindstorms kit and Q-learning. Acta Universitaria, 22(0).

[6] Vamplew, P. (2004). Lego Mindstorms robots as a platform for teaching reinforcement learning. International Conference on Artificial Intelligence in Science and Technology, pp. 21–25.

[7] Kroustis, C. A., Casey, M. C. (2008). Combining heuristics and q-learning in an adaptive light seeking robot. Technical Report Rep. CS-08-01, 2008, University of Surrey, Department of Computing.

[8] Sutton R. S., Barto, A.G. (1998). Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning Series. Mit Press.

[9] Murphy, K. P. (2002). Dynamic bayesian networks. Probabilistic Graphical Models, M. Jordan.

[10] Millington, I. (2006). Artificial Intelligence for Games, chapter 7. Adaptive Computation and Machine Learning Series. Elsevier Science.

[11] Bricx Command Center (2014). http://bricxcc.sourceforge.net

[12] Hansen, J. (2007). Not eXactly C (NXC) Programmer's Guide. Version 1.0.1 b33.

[13] Martínez-Tenor, A. (2014) Reinforcement Learning on the Lego Mindstorms NXT Robot. http://www.youtube.com/watch?v=WF9QWc_lxfM

Nombre de archivo: qlearning_template.doc
Directorio: Z:
Plantilla: C:\Documents and Settings\anita\Datos de programa\Microsoft\Plantillas\Normal.dot
Título: IATED PAPER Template
Asunto:
Autor:
Palabras clave:
Comentarios:
Fecha de creación: 02/02/2009 17:30:00
Cambio número: 290
Guardado el: 30/09/2014 18:24:00
Guardado por: instalador
Tiempo de edición: 1.102 minutos
Impreso el: 01/10/2014 10:36:00
Última impresión completa
Número de páginas: 10
Número de palabras: 4.976 (aprox.)
Número de caracteres: 27.374 (aprox.)