Scientific Bulletin of the „Petru Maior" University of Tîrgu Mureş
Vol. 9 (XXVI) no. 1, 2012, ISSN 2285 – 438X (Online), ISSN–L 1841 – 9267

ROBOT CONTROL USING Q-LEARNING

SZÁNTÓ Zoltán

Department of Industrial Engineering and Management
Faculty of Engineering
„Petru Maior" University of Tîrgu Mureş, Romania
Nicolae Iorga Street, no. 1, 540088, Mureş County

szanto_z_zoltan@yahoo.com

ABSTRACT

This paper focuses on machine learning, where an agent learns how to solve a specific problem. The learning process will take place in a simulated environment, so the effectiveness can be measured without any potential damage to real the robot. Q-Learning is a temporal-difference learning method, that maps the effectiveness of an action in a given situation. Our learning agents use this method to solve a simple "catch and escape" scenario in a 2D world.

Keywords: robot control, q-learning, reinforcement learning, simulation, ε-greedy algorithm

1. Introduction

Our modern, scientifically advanced world is heavily based on robotics. At first, robots were introduced to the assembly line to execute specific tasks over and over again. Due to their versatility, we are using robots everywhere, especially in environments that are too dangerous for humans. Typical applications for these industrial robots include painting, assembly, welding, packing, testing.

In the last two decades, robotics has taken a step further with the extra intelligence added: the robots are capable off decision making in certain situations. The modern robot tends to mimic the behavior of humans or animals. The authors of [6] have created robot that learns to act as a pet dog. Other project focus on insect, or spider like behavior, see [2] for details. Thus, the term "intelligent robotics" is used. Even competitions like RoboCup where created to test these robots.

The ultimate goal is to achieve a fully autonomous, intelligent, mobile robot that can solve problems in real the world. Just to name a few areas: outer space exploration, mining, deep sea constructions, disaster areas, or places to small for humans, or canines to fit in etc. Also there are everyday issues that can be done by intelligent robots: imagine a robot that cleans the house while you are away.

The use of reinforcement learning (RL) algorithms in solving everyday problems is a relatively new approach. In theoretical problems, such as Maze World, or Cliff World [3], these algorithms perform well, great results can be achieved. However, in real life the problems are dynamic, usually in a noisy environment, therefore we wish to create our own theoretical world and test reinforcement learning methods here.

Our goal is to implement a "catch and escape" scenario, in which our trained agent will try to locate and catch the escapee in a 2D terrain that has obstacles in it. To accomplish this we will use a RL algorithm, Q-learning, that will guide the agent through an unfamiliar terrain and to the target.

2. Reinforcement Learning (RL)

A reinforcement learning agent learns the impact of a certain action that it preforms in a given environment, thus the environment is providing the feedback. Unlike in many other machine learning algorithms, here "the learning agent is not told which to take, it has to discover, by trial and error", which action gives the most reward.

Richard S. Sutton and Andrew G. Barto, see [3], note that: "These two characteristics--trial-and-error search and delayed reward--are the two most important distinguishing features of reinforcement learning". Temporal-difference methods (TD) "require no model and are fully incremental, but are

more complex to analyze". TDs are a combination of DP and Monte Carlo methods. TD methods learn directly from raw experience without a model of the environment's dynamics, and "they bootstrap": "update estimates based on previous estimates".

Q-learning is an off-policy reinforcement learning algorithm created by Watkins in 1989, see [5]. Q-learning is a member of the Tds methods family. The algorithm learns an "action-utility representation instead of learning utilities". The Q function calculates the "Quality of a state-action combination" [4]. If S is a set of states and A a set of actions, then the simplest form, called "*one-step Q-learning"* [3], is:

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

Where *Q(s,a)*, denotes the value of taking action *a* in state *s*. Often a two dimensional table is used, the Q table, to store the data. The main parameters are:

- α – learning rate: meaning how much of the new information will override the old information. A factor of 0 means that the agent will not learn, while a factor of 1 means that our agent considers only the most recent information
- γ – discount factor :is used to weight the rewards: The closer to 1 the greater the weight of future rewards. A factor of 0 will make the agent "opportunistic", giving much more interest to current rewards.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Fig. 1: Q-learning Algorithm

The Q-learning method, and TD methods in general do not specify what actions should the agent take in each step. The agent can take any action, from a list of possible actions. This means that the agent at each step can chose an action that is either exploratory or the best know until this point. The ε-greedy method specifies this: the closer ε is to 0, the agent will take the best action, whereas if ε → 1 the agent will simply explore the state, meaning that it will take a random possible action.

In our learning scenarios, the agent uses the algorithm which is presented by the authors of [3]. This algorithm can be seen on fig 1.

At the beginning of the training the γ is set to a high value thus forcing the agent to explore. After a while we can lower this value and choose exploitation instead.

One of the greatest advantages of the Q-learning algorithm is its simplicity. Th major drawback of the algorithm is the finite number of states and actions – in the real world this is not always possible. If the number of states, or actions increases the Q-table gets big, resulting in a poor performance. The problem can be solved by using discrete the inputs.

Another disadvantage is the local maximum problem – selecting the best rated action can take the agent in a local maximum.

3. Proposed framework

A simulated world filters many noises that are common in the real world. For example, in the real world the ground might be slippery in some parts, a robot's wheel might slip, thus causing false inputs. Also, in the real world collisions can have bad consequences, like partially destroying the robot. In the simulator collisions are only theoretical.

Q-learning is a "simple" algorithm, defined by a few key inputs and outputs. *States* are the way of defining how the agent "interprets current state of the environment". In every given state the agent can choose from a series of actions. These *Actions* are: move (forward), rotate right, rotate left. After executing the selected action, a reward is given. A *Reward* representing the amount theoretical bonus for reaching that state.

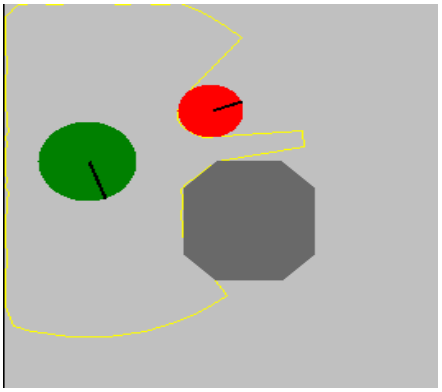

Fig. 2 – Simulation

As seen on fig. 2, the agent, shown here in green, searches for the target, here in red. The agent has only local knowledge this information is gathered from sensors. To simulate this we added a "visibility circle" around the agent, shown here in yellow.

Working with only local knowledge means that fewer information needs to be processed.

Because we are not focusing on gathering and processing sensor information, everything inside this circle in considered known to the agent. We tried to make this visibility as realistic as possible: the agent can not see through objects. An object can be either the target, an unmovable obstacle (dark gray), or a wall (no color).

4. Environment without obstacles

At the first approach we put our agent in an environment that had no obstacles. Catching a target in an unknown environment, requires the following steps: wander around until target is visible, rotate the direction marker towards the target, move forward, see fig. 2 for details.

Every state is composed of two descriptors: *distance* and *angle*. The *distance*: the euclidean distance between the targets and robots center in sorted in a few distance categories: "category0" means that the target is right near the agent.

The *angle* is always between [0, 360), no negative values are allowed. In our case an angle of just 1° is not significant. This means that we use angle categories instead: "category0" points directly in front of the robot and represents any angle from [+5°, -5°]. Angle categories are visible on the fig. 3, because having to much lines will make confusion, they are not displayed. The authors of [2] and [6] also use discrete inputs since they are also working with continuous values.

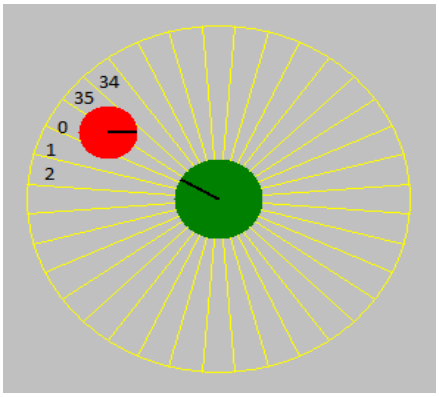

Fig. 3 – Angle categories

5. Environment with obstacles

The current state descriptors do not provide a way handle obstacles because they only contain information about the target. Unfortunately, almost every real environment has some sort of obstacles. In our simulated world we added only stationary obstacles, meaning that they do not change their position in time.

Catching the target still requires the previously mentioned steps, but when an obstacle blocks the path to the target the agent needs to figure out how to avoid a collision. A basic solution to an obstacle would be to use a modified BUG1 algorithm where the agent simply moves along the contour of obstacle until it is no longer blocking the path to the target. See [1] for details on Bug1.

To do this the agent needs to store the obstacles relative position to its own. The agent is equipped with bumper sensors that inform the algorithm about the objects which lies directly in front of the agent. These bumpers are not visible by default, but the can be seen on fig. 4.

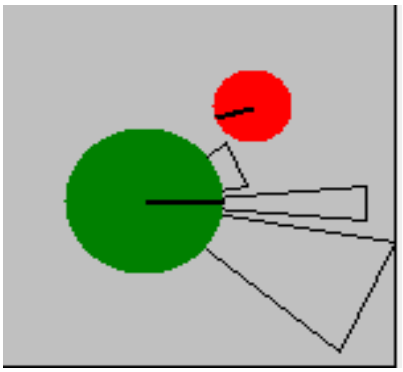
Fig. 4 – Bumper sensors

The state is described similarly as in the previous case, but with three additional sensors. These are: left, front, right bumper. A bumper holds extra information: what lies there, and how far is it. On fig. 3 the left bumper sees the target, the front bumper has nothing, and the right bumper points to the wall. The agent needs three bumpers because it must know were the obstacle is located. If the "left bumper" is showing that we are about to hit an obstacle, we can enforce the agent to take the appropriate action, in this case "rotate right".

If the target is currently not visible the agent's main task is to avoid obstacles. Otherwise follow the target and avoid obstacles if necessary.

6. Experimental results

For testing we used the following algorithm:

1. Generate a new target
2. Start the agent → wandering process
3. Check if the target is reached - in other words the target in front of the agent with minimum distance between them.
4. If true go to 1.
5. Otherwise go to 2.

We repeat this process until the agent learns the problem. We consider the learning process successful when our agent upon seeing the target starts moving directly towards it successfully avoiding any obstacles.

We repeat this process for five times and calculate the average learning runs. This will give us information about how fast the learns in a given test

scenario. These scenarios are created based on the rewarding method.

6.1 Agent, no obstacles

Choosing the wright reward set is a very difficult process. First we experimented with rewarding only when the target is reached. This approach proved not to be a successful one: it needs a large amount of trials to finally move towards the target. The average here was 1648 runs.

Rather than giving a "big reward" for reaching the target, we introduced a new concept: sub goals. If the agent is "getting closer to the target" give a positive reward, otherwise negative. This is calculated based on the current angle and distance categories and the previous angle and distance categories. If the distance and the angle is getting smaller, then the reward increases. We used a higher value for angle because it is more important to turn towards the target and then to start moving forward.

To further enhance the algorithm, we added 'Losing Target' reward which means that if the agent loses the target it will get a big negative reward. This produced almost the same result, but with half of time need to learn the task. Also we experimented with other sub goals like: "is the distance getting smaller?", giving a small but positive reinforcement to the agent. Similarly, we applied this to the angle input as well. In test case #3 the agent learns the problem in approximately 300 successful learning iterations.

Table 1: No obstacles - summary

#	Avg	Rewarding
1	1648	Reach the Target → +100
2	972	Added 'Losing Target' rewards: current distance>older distance → -100
3	308	Added 'Sub Goals' rewards: Angle: "from good to worse" → -3 Angle: "from bad, improvement" → +2 Distance: "from good to worse" → -2 Distance: "from bad, improvement" → +1

6.2 Agent and obstacles

As described earlier this agent is equipped with more sensors (bumpers) that serve as inputs for the agent. The drawback of adding more state descriptors is that the learning will take longer, but the obvious advantage is that "the agent knows more information about the environment".

The same testing scheme presented earlier is applied here to after all this agent is an upgraded version of the previous one, meaning that the reward rules still apply here as well. Only difference is that once the bumpers sense that the agent is about to hit some object (other than the target), they will force the agent to turn away first. They also help to solve a very interesting problem: the local maximum, shown on fig. 5. The problem here is that if the agent chooses the "left" action it will hit the obstacle. Choosing the "right" action turns away from the target.

Here, the algorithm is overriding the just for brief moment the goal of reaching the target, with avoiding collisions. In this case the reward is greater than 100 (the reward normally given for reaching the target). We also experimented with a smaller reward, 90, but in this case the results were worse then before.

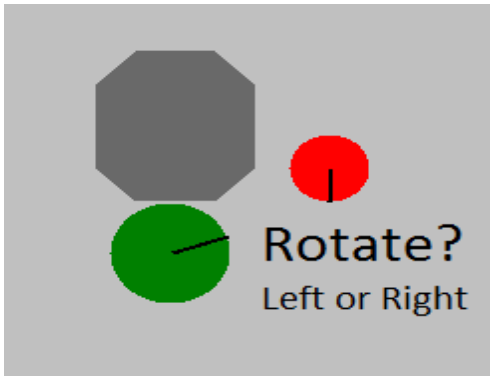

Fig. 5 – Local maximum

This agent works the same as the previous one in an environment containing no obstacles. However, when these exist evasive action is taken. First we experimented with "not forward" reward: meaning that if an obstacle is blocking the path force a rotating action. This setup does not hit any obstacle but it might take the wrong angle (ex. on fig. 5 rotate left all the way). The success rate is low, in many case the agent fails to learn the problem.

Then we introduced the "good angle reward". If the right bumper is activated and the other is not, the agent turns the other way. Same applies in the other case.

This setup sometimes gets caught in an infinite loop due to the learning process. Look at the situation presented in fig. 5. The agent chooses the correct action (rotate right), however after executing this action, it immediately chooses the opposite action (rotate left) which takes us back to the initial problem.

To solve this in test case #3 check the previous action as well. The rewarding is the same as in test case #2 but the algorithm also checks the previous action thus avoiding the infinite loop.

Table 2: With obstacles - summary

#	Avg.	Rewarding
1	14821	Added "Not forward" → +10
2	5138	Added "Good angle" → +110
3	2387	Added "previous action check"

7. Conclusions

The Q-learning algorithm can be applied even in a dynamic environment. Selecting the proper state space can be difficult even at such a simple task. Also the research shows how the rewards affect the learning.

We store the data in a multidimensional table, the Q table. The dimensionality of Q is given by multiplying the state descriptors: #angle x #dimensions... x #actions. In the current the Q table there are unused areas. If the target is not visible, then the angle and distance do variables do not count, they are set to predefined number. Although not used theoretically, it occupies space in the memory. One way to fix this issue is to create a list of states, and every time the agent finds a new undocumented state, it simply adds it to the back of the list. We experimented with this as well. Problem here is a new state found only in the testing phase (does not have a Q value). The agent can take a potentially bad move.

The main benefit of using discrete input values is that less categories are manageable. This way we only handle significant changes, that clearly have an impact, but if we do not select a proper sampling, we might be loosing on these details.

Another potential problem might be the usage of ε-greedy algorithm. This approach allows the agent to select between exploration and exploitation. In case of exploitation, what happens if we have multiple best actions? The answer is that always the first, or last action will be selected. Using a soft-max algorithm [3], this problem can be solved. The soft-max algorithm distributes more equally among all actions that give the same reward. The soft-max algorithm provide a better distribution among equal Q valued actions. The main problem here is that the agent always chooses the first (or last depending on the search implementation) best value, We did not include this algorithm but in the near future we wish to do so. For more details see [2].

8. Future

In the near future, we wish to further extend this research, implementing a system in which many agents compete against one another, or perhaps working together to achieve a common goal.

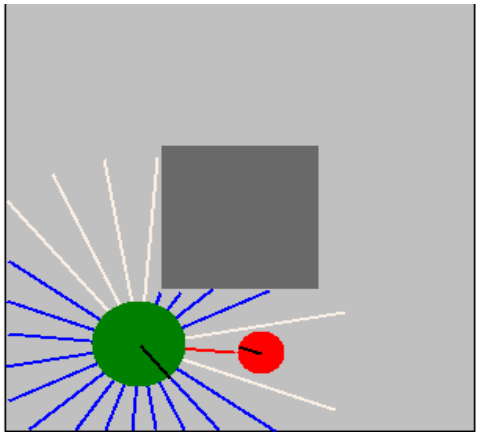

Fig. 6 - Potential field

Another idea is to change the state descriptors completely. This new approach is illustrated on fig. 6. The agent here is equipped with sensors that go all around. Each sensor marks how desirable that point is: *red* attracts, *blue* repels and *white* means that there is nothing there.

This new method is called potential field navigation. Currently this development is in testing phase.

9. References

[1] Annamária R. Várkonyi-Kóczy: *A Complexity Reduced Hybrid Autonomous Navigation Method for In-Door Robots*, IPSI BgD Journal 2010

[2] Q-Learning Hexapod (May 2009), Matt R. Bunting, Member, IEEE, and John Rogers

[3] Richard S. Sutton, Andrew G. Barto (1998), *Reinforcement Learning: An Introduction*, The MIT Press Cambridge, MA USA, ISBN: 0262193981

[4] Russell, Stuart J.; Norvig, Peter (2009), *Artificial Intelligence: A Modern Approach* (3rd ed.), Prentice Hall, ISBN: ISBN 0-13-604259-7

[5] PhD thesis "*Learning from Delayed Rewards*", Cambridge, 1989 by Christopher John Cornish Hellaby Watkins.

[6] Måns Ullerstam's Master's thesis: *Teaching robots behavior patterns by using reinforcement learning. How to raise pet robots with a remote control*, NADA, Kungliga Tekniska Högskolan, within Computer Science and Engineering, 2004

[7] Yadira Qui˜nonez, Javier de Lope, and Dario Maravall (2009) *Cooperative and Competitive Behaviors in a Multi-robot System for Surveillance Tasks.* Eurocast 2009, 12th International Conference on Computer Aided System Theory

[8] Luiz A. Celiberto Jr., Carlos H.C. Ribeiro, Anna H.R. Costa , and Reinaldo A.C. Bianchi: *Heuristic Reinforcement Learning Applied to RoboCup Simulation Agents*, RoboCup journal 2007, Springer