Koolio: Path planning using reinforcement learning on a real robot platform

Lavi M. Zamstein
University of Florida
dino-man@ieee.org

Dr. A. Antonio Arroyo,
Dr. Eric M. Schwartz
University of Florida

Sara Keen,
Blake C. Sutton,
Gorang Gandhi
University of Florida

Machine Intelligence Laboratory (MIL)
MAEB 325, Building 720
University of Florida
Gainesville, FL 32611-6300
00-352-392-2541

ABSTRACT
In this paper, we describe the learning process of an autonomous delivery robot. We will discuss reinforcement learning and which method is used to reach an optimal policy of actions using only the current sensor inputs.

Keywords
Q-learning, reinforcement learning

1. Introduction
Koolio is part butler, part vending machine, and part delivery service. He stays in the Machine Intelligence Laboratory at the University of Florida, located on the third floor of the Mechanical Engineering Building A. Professors with offices on that floor can access Koolio via the internet and place an order for a drink or snack. Koolio finds his way to their office, identifying it by the number sign outside the door.

Koolio learns his behavior through the reinforcement learning method Q-learning.

2. Koolio
Koolio is controlled by an integrated system consisting of a single board computer along with a pair of smaller processors, used to organize sensor data and to drive the motors [1].

The platform itself consists of a small refrigerator mounted on a circular base. The sonar and bump sensors are located on the base, with a digital compass on top of the refrigerator. Also mounted on the base are a pair of tall poles, which hold up the side-facing cameras, side sonar sensors, and the LCD screen.

3. Reinforcement Learning
Reinforcement learning is a method of learning by way of rewards and punishments. The learning agent will seek those choices that result in high rewards and avoid those actions that result in low rewards or punishments (negative rewards). In this way, an agent will learn to follow the decision path that results in the best possible reward.

Many reinforcement learning processes will only work properly if a Markov process is assumed. A Markov process is any process that uses only current values of inputs to determine the output. All past input values are ignored in Markov processes. This makes Markov processes very useful in the realm of robots, since the output and all calculations will rely only on the current input of the sensors, rather than memory of any past input. Not only does this remove the necessity of memory for past sensor values, but it also reduces the calculations greatly, as there are fewer variables to deal with – only the current sensor values instead of the current and many past sensor values.

One of the fundamental balances in reinforcement learning is that between exploration and exploitation. Since the agent wants to maximize its reward, it will often make the choice that offers the highest reward. However, if this best choice is always made, then many other possible choices are never looked at. It is possible that some choices which yield smaller rewards will, in the end, give higher returns, due to better states following a single sub-optimal choice now. To avoid the skipping of such cases, there must be some exploration of non-optimal states, in addition to exploitation by picking the current best choice.

3.1 Application example
One application of reinforcement learning is the behavior of robots. As an example, the following explanations of reinforcement learning methods will use a robot that is learning to wall-follow while avoiding obstacles and not touching the wall itself, using simple infrared (IR) and bump sensors.

3.2 Reinforcement Learning Algorithms
The primary algorithm chosen in solving this is known as ε-greedy. In this algorithm, ε is some small percentage, with a larger ε (such as 10%) being used for a more exploration-oriented design and a smaller ε (such as 1%) used for a more exploitation-oriented design. Most designs use an ε of 5% by default. The current best choice is chosen 1-ε of the time (exploitation), but during the other ε of choices, a random selection is made (exploration) [2]. This allows for choices which are not currently the best, but which may lead to policies that prove optimal.

There are three major groups of methods in reinforcement learning, though each of the methods may be further divided into more specific methods. These three main methods are differentiated from one another by two main factors: models of the environment and bootstrapping. Bootstrapping is the method of updating estimates based on other estimates. In the realm of

reinforcement learning, bootstrapping is used to update the estimates of state values and action values based on the estimates of the state values and action values of later states.

- **Dynamic Programming** (**DP**) methods use bootstrapping and require a complete and accurate model of the environment.
- **Monte Carlo** (**MC**) methods do not require a model of the environment and do not bootstrap.
- **Temporal Difference** (**TD**) methods do not require a model of the environment but do bootstrap.

3.2.1 Dynamic Programming methods

Because DP methods require an accurate model of the environment, they are not suitable for reinforcement learning with robots. This is because the environment is the real world around the robot, and therefore is very difficult, if not impossible, to model perfectly. In addition, if the environment were to change, the entire model must be changed, which will severely hamper the learning process.

On the other hand, since MC and TD methods do not require an accurate model, but only a general idea of the environment, changing that environment has much less impact on their methods of learning.

Because DP methods are not efficient for use in reinforcement learning for robots, we will favor MC and TD methods over them.

Using the wall-following example, assume the agent has a perfect model of its environment. If the robot is moved to another environment, either by changing the current one or placing it in a different room, the model is no longer valid and the learning process will not work until a new model is made. If the robot were to learn to perform the desired following action in any given room, the policy created from that learning would not work in any other room because of the different environment. Any change in sensor readings will also read as a different state, thus altering the environment. For this reason alone, DP is not suitable for use in real robots, since sensors are not perfect and the values can vary.

3.2.2 Monte Carlo methods

Because MC methods do not require a complex model of the environment, they are much more fit than DP methods for use in robot learning. The environment model for MC need only generate sample state transitions. In other words, the model only needs to define the states and their possible actions, and how the agent can move from one state to another.

This in itself makes MC methods usable for robot learning, as well as minimizing the effects of a changing environment. In the example of wall following and obstacle avoidance, for example, the model must have states defined for the various possible conditions the agent may find itself in. Some of these are *nothing* (where neither the IR nor the bump sensors are activated, showing the robot is in a clear area), *wall* (where the IR sensors detect some obstacle on one side), and *contact* (where the bump sensors indicate the robot has touched an object). For a more complicated model, one could add states for corners, dead ends, and tight spaces. Using the basic states of nothing, wall, and contact, however, a robot can successfully learn to follow walls without bumping into them.

Unlike DP methods, MC and TD methods do not require a model of the environment, and are therefore much more flexible [3]. If a robot were to learn the "following and not bumping" behavior, and was then moved into another room with similar makeup (walls and solid objects), either in the middle of the learning process or after the learning is complete, it would be able to perform with the same policy or with only slight adjustments needed. This also allows for small fluctuations in sensor readouts without altering the decisions made by the learned policy.

However, if the substance of the environment were changed, there would be a problem. For example, if an obstacle was introduced that could be seen with IR sensors but was light enough to be pushed by the robot without triggering the bump sensors, that would represent a fundamental change in the state makeup. Since the "normal" wall/object definition is something that is solid and causes the bump sensors to activate when hit, this new object is completely alien to the agent's knowledge and will serve to make the learning process more difficult, if not impossible. The only way to allow for a change such as this is to add new states, which will then require additional learning to be done to alter the policy to account for the new states.

Because MC methods do not bootstrap and use average sample returns, they are defined only for episodic tasks, since the value estimates and policy are updated only at the termination of each episode. Thus non-episodic tasks must use TD methods for learning.

MC methods use a cycle of evaluation and improvement to implement policy iteration. For each instance of this cycle, the state-action functions are evaluated for the current policy. The best of these actions for each state are selected to make the new policy. This new policy can then be evaluated during the next episode to repeat the cycle. The greedy method is often used in the policy improvement, resulting in a new policy that is either of equal or greater value than the previous policy. Given an infinite number of episodes and an assumption of exploring starts, this method is guaranteed to converge to an optimal policy. Realistically, however, there can never be an infinite number of episodes. Therefore, the evaluation step of each evaluation and improvement cycle may be changed so the value function approaches the state-action value function. In other words, the policy need not immediately jump to the new better value, but may simply proceed toward it. If the next policy is better and has the same state-action pair, then the improvement will approach it further. If not, there is much less backtracking needed to move toward a new choice.

Since a good model of the environment is never assumed for MC methods, state-action values become more important than state values, since they are possible to estimate without values of future events, whereas state values require knowledge of the following states in order to estimate them. However, it is therefore possible for direct policy evaluation to miss some actions in a state. In order to avoid this, exploration must be maintained in a method called **exploring starts**. Exploring starts gives every state-action pair a non-zero probability of being the first step in each iteration.

In order to use exploring starts in our robot example, the robot must be placed in a random location in the environment at the start of each episode. This also means that it must sometimes be placed near a wall to fulfill the state of sensing a wall nearby, as well as touching a wall, to explore the state of bumping into a

wall. From these random starting positions, an action must be chosen randomly from the possible actions available in that state. All this will ensure that exploration is maintained and no state-action pairs are skipped.

This method of ensuring exploring starts, however, is tedious and not likely to fully explore all the possible states and actions as initial steps. There are two other methods used to ensure full exploration: on-policy MC and off-policy MC.

On-policy MC methods use **first-visit** methods (methods which use the average of returns after the first instance of a given state to determine the state-action pair's value function) to estimate the current policy's state-action value functions. Because a full greedy algorithm for improving the policy would miss some state-action pairs without the assumption of exploring starts, ε-greedy is used instead. This ε ensures that policy improvement approaches an optimal policy while still maintaining exploration.

With the robot example, this means that at any state, the robot has a ε chance to select some random action rather than the current best action.

Off-policy MC methods separate the functions of control and policy evaluation into two separate functions: the behavior policy to govern the decisions of the current policy and the estimation policy to be evaluated and improved. Because the policy functions are separated in this way, ε-greedy is not needed, and greedy is sufficient for exploring all possible state-action pairs, since the estimation policy may continue to explore while the behavior policy acts. The tradeoff for not needing ε-greedy, however, is that these methods learn faster for selecting non-greedy actions than for selecting greedy actions, resulting in slow learning for some states, especially early states in long episodes.

With our wall-following robot example, off-policy MC would remove the random aspect of on-policy; however, it would see a decrease in learning speed, especially for longer episodes.

3.2.3 Temporal Difference methods

Like MC methods, TD methods also do not need a model of the environment, and make predictions in the same way, using sample state transitions. Unlike MC methods, TD bootstraps, using estimate to update other estimates. Because of this, TD methods update values after each time step, rather than after each episode. Bootstrapping allows for TD methods to learn faster than MC methods during exploration, since the policy update each time step means the method does not need to wait until the end of an episode to determine if that choice was better or worse than the choice indicated by the previous policy. TD methods can then know whether an explored choice is good or bad right away, rather than waiting until the end of the episode.

The use of bootstrapping is the major difference between TD and MC methods. Many other parts are the same, such as the use of on-policy and off-policy methods, though the results may be different because of the quicker learning under exploration with TD.

TD learning also has several special types of methods, both on and off policy, which are useful in certain cases.

Sarsa, an on-policy TD method, updates after each transition from a non-terminal state. Because of this, sarsa methods always converge to optimal policies so long as all the state-action pairs are visited an infinite number of times and the policy is able to converge to the greedy function.

Unlike MC methods, sarsa and other TD methods are able to determine during an episode whether the policy is good or bad, and change policies if the current one is determined to be bad. This proves very useful in episodes where the current policy may never finish, from an inability to reach the goal.

In the robot example, if the goal was to spend more time along a wall than in the open, some policies may never reach that goal. For example, a policy that turns away from a wall as soon as it is spotted will always spend more time away from the wall than against it, resulting in a never-ending episode. Using sarsa or other TD methods avoid this by determining in the middle of the episode that the choices defined by the policy are not able to reach the goal.

Q-learning, an off-policy TD method, uses learned action value functions to approximate the optimal action value function independent of the current policy. This simplifies policy evaluation and updating while only making the assumption that all state-action pairs continue to be updated. Unlike sarsa, Q-learning does not need to assume state-action pairs are visited an infinite number of times, making it more likely to reach the optimal policy than sarsa given a finite number of episodes [4].

Under the robot example, Q-learning is more likely to be adventurous than other learning methods, since it uses ε-greedy to explore. Because of this, it is likely to quickly go to the wall, but will also bump into the wall more than other methods during the process of learning, as exploration will be more likely to push it in a random direction.

Actor-critic, a group of on-policy TD methods, separates the policy and the value function into independent memory structures. The policy structure, or actor, is used to decide which action to pick in each state. The estimate value function, or critic, determines whether the actions of the actor are good or bad, and whether they should be encouraged or discouraged. These methods are useful because they do not require much computation to select actions, due to the policy and value functions being stored and operated independently. Because of the division between decision making and decision critiquing, actor-critic methods are also useful in modeling psychological or biological behavior, as such behavior functions under the same general separation structure.

R-learning, an off-policy TD method, does not discount past experiences, unlike most other learning methods. R-learning also does not divide experiences into episodes, instead learning from a single, extended task. This shifts the priority for the optimal policy to optimizing each time step, rather than optimizing each episode. R-learning methods use relative values, state-action functions that determine their value based on the average of all other state-action values for the current policy. With relative values, the value of each action chosen is compared to the overall average of all other values in the policy. If the chosen action is better than the average, it is considered good and is incorporated into the policy. If the action returns a value worse than average, it is considered bad and discarded from the policy.

4. Reinforcement Learning Method

Q-Learning was the reinforcement learning method chosen for use in Koolio. This was done for several reasons. Temporal

Difference learning methods are very useful for mobile robots since they allow for a possible change in the environment and they bootstrap. The bootstrapping involves a tradeoff. Bootstrapping methods require more complicated code, but are more efficient in the learning process. Therefore it was decided to use a TD method to allow for optimal learning efficiency. Among TD methods, Q-learning was chosen for ease of use and availability of reference materials. Although Q-learning is not the most efficient learning method for a single robot process, it is a very popular learning method, so there exists lots of documentation to refer to when setting up the code.

5. Learning Process

5.1 Step 1: States and Actions

For a mobile robot, the states are simply the set of all sensor input. Because most of the sensors used on Koolio have analog outputs, there is the potential for a very large number of possible states. Therefore, states must be defined using ranges of sensor values.

Aside from basic informational output such as the LCD screen, the only actions are movement of the wheels. Since the motor drivers operate independently from the learning process, it does not need to put into consideration the specifics of operating the motors directly. The only actions then are the basic movements of forward, backward, turn left, turn right, rotate left, and rotate right.

5.2 Step 2: Simulation

Reinforcement learning in episodic tasks requires a very large number of repetitions to learn. Along with this, many repetitions on the platform can result in wear of parts, and the testing area of a hallway may not always be available since it is used on a daily basis. Because of these factors, the initial parts of learning must be done in simulation.

Learning in simulation has other advantages as well. Data can be easily stored from simulation runs and referred to later [5]. Using simulation data, a graph can be constructed showing the growth of the return as the policy approaches optimal. If data is recorded on the actual robot, graphs can also be constructed. Simulations can also run much faster than a real robot, so many simulations can be run in the time of a single real robot episode.

There are some drawbacks to simulation, however. No matter how good a model of the environment for running test episodes is, it can never be perfect. A learned policy from a simulation may not operate correctly in a real environment because of any number of imperfections in the model.

However, simulation is still an important tool in the learning process. By developing an optimal policy in the simulated environment, much of the time of real robot learning can be done before involving the actual robot. The policy can then be exported to the robot and the learning process can continue.

5.3 Step 3: Real Robot Learning

Once the simulation has reached an optimal policy, it can be brought to the robot to continue the learning. Because a good deal of learning has already taken place, this phase of the learning process is much faster than it would have been if the simulation was skipped and the learning was done solely in the real environment. By the time the policy is ready to be used on the platform, it will have been changed to avoid many time-consuming mistakes such as random wandering.

Despite the shortcuts of using a simulator for the initial learning process, this phase of real robot learning is still the most time-intensive, as episode runs of the robot can take several minutes instead of the accelerated time used in simulation. Because the policy is already refined, however, only a relatively smaller number of episodes is required for reaching a new real environment optimal policy.

6. Future Work

With reinforcement learning, Koolio can be transferred into another environment with similar makeup and learn fairly quickly how to operate optimally in the new environment. For instance, if Koolio was moved into another hallway with the same physical characteristics (such as wall color and room number signs), it could learn to find a room in much less time than it took to initially learn how to navigate a hallway.

Because reinforcement learning develops a policy of action, it is a trivial matter to transfer Koolio's learned policy into another robot. So long as the new robot has the same sensors and motor drivers, it should be able to perform the same operations with little to no additional learning required.

7. Acknowledgements

Our thanks to the University of Florida Machine Intelligence Laboratory for use of resources and to my committee chair Dr. Arroyo. We also thank the rest of the Koolio team for their work on the platform. We would like to recognize Brian Pietrodangelo, Kevin Phillipson, and Dr. E. Schwartz, the original designers of Koolio who made this project possible, as well as Halim Aljibury, who provided the simulation program.

8. References

[1] Sara Keen, Lavi Zamstein, Wenxing Ye, Blake Sutton Gene Shokes, Gorang Gandhi, Eric M. Schwartz, A. Antonio Arroyo. *Koolio: An Autonomous Refrigerator Robot.* Proceedings of the 2006 Florida Conference on Recent Advances in Robotics, May 25-26, 2006, FIU, Miami, Florida.

[2] Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

[3] Malmstrom, K., Munday, L., and Sitte, J. *Reinforcement Learning of a Path-Finding Behaviour by a Mobile Robot*. Australian New Zealand Conference on Intelligent Information Systems, Adelaide, Australia, 1996.

[4] Smart, W. and Kaelbling, L. *Effective Reinforcement Learning for Mobile Robots*. Proceedings of the 2002 IEEE International Conference on Robotics & Automatons, Washington, DC, 2002.

[5] Su, M., Huang, D., Chou, C., and Hsieh, C. *A Reinforcement Learning Approach to Robot Navigation*. Proceedings of the 2004 IEEE International Conference on Networking, Sensing & Control, Taipei, Taiwan, 2004.