

Project3: Reinforcement Learning

CS3630 Spring 2007

Due Beginning of Class March 1

20th February 2007

Introduction

Reinforcement learning is a learning-control technique concerned with how an agent should take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to find policies that choose optimal actions. In this project, you will get a more comprehensive understanding on reinforcement learning by implementing it based on the *Processing* environment.

1 Getting Started (No Deliverable)

Install the *Processing* environment on your machine. You can download it from <http://processing.org/download/>. Double click the 'Processing' application, and a simple editor-based environment is launched in which you can create, load, and run applets.

Testing your Processing Installation

Get the 'Sample' applet distribution from <http://borg.cc.gatech.edu/ipr/files/Sample.zip>. Unzip it and load 'Sample.pde'. This applet shows how to sample probabilities from a cumulative probability function, which you will find very helpful in Section 2. The 'setup()' function creates 300 equivalent probabilities and builds cumulative probabilities according to them. The 'draw()' function calls 'sample()' repeatedly to sample from the cumulative function and draw a histogram to show how many times each probability is sampled.

Getting to know the RL applet

Get the assignment's applet distribution from http://borg.cc.gatech.edu/ipr/files/RL2_EMPTY.zip. Depending on which platform you are on, you might have to uncompress the file differently, e.g., on Linux do it with the command 'unzip RL2_EMPTY.zip'.

You should now have a directory called RL2_EMPTY, which contains an applet source file named 'RL2_EMPTY.pde'. Load this file by choosing: 'File -> Sketchbook -> Open ->', and run it by choosing: 'Sketch -> Run'.

The agent starts out with a random policy and starts executing it immediately. In each cell, the agent can take three actions: up, stay, or down. The agent's preferences for each of these actions (Q values) are indicated by the color of the cell: red=up, green=stay, blue=down. To start getting a feel for how the applet works, click in the window and try the following: press 'r' to add some rewards (strawberries!), press 'q' to start executing again, press 'l' to start learning a policy that behaves optimally given the rewards.

Keep pressing 'I' until the Q-values converge. More details about the agent interface and reward editing mode are available on <http://www-static.cc.gatech.edu/~dellaert/applets/RL/index.html> or at the beginning of 'RL2_EMPTY.pde'.

Implementation Details

In the applet you downloaded the world is deterministic. The 'transition(s,a)' function implements the deterministic state transition, which is used by 'simulate(s,a)'. Learning through value iteration is implemented by the function 'iterate_values()': it first executes one step of value iteration, i.e., for each state s and action a the Q-values are updated as follows

$$Q(s, a) = R(s, a) + \gamma V(\delta(s, a))$$

where δ is the 'transition' function, and then calls 'calculate_values()' which re-computes the value function as the maximum Q-value at each state, i.e.,

$$V(s) = \max_a Q(s, a)$$

When you run the program, keep pressing 'I' so that 'iterate_values()' is called repeatedly. After the Q-values converge, you can see that the agent executes the policy learned by value iteration and successfully reaps the rewards you have put in the world.

2 Making the World Non-Deterministic

In this section, you will introduce non-determinism into the world and see what will happen in this case. In Section 1 the world was deterministic world, but in a real application an agent's actions can sometimes yield unpredictable results, i.e., the agent (or world) will not always transit to the correct state. For example, if you ask the agent to take action "up", it might "stay" or go "down" resulting in a different state than expected.

To model this, you need to introduce a state transition probability function

$$T : state \times action \rightarrow state$$

To implement this, fill in the 'T(s,a,s\'' function in the file 'Part2_EMPTY.pde'. While we put in a default implementation (what does it do?) it is not realistic: please change it to implement realistic agent behavior, i.e., the actions transition occasionally to a different state. Then, modify the 'simulate(s,a)' function to use the transition probability function 'T()'. You will need to use the 'sample()' function introduced in Section 1 to sample from the probability distribution over possible next states.

After this, run the program again, including learning, and you will see the agent execute a policy in a non-deterministic world, but one that was learned using the old, deterministic value iteration.

Deliverable

1. Briefly describe your implementation and the choices you made.
2. Save your code as 'Section2.pde'.
3. Discuss how the performance of the agent changes after the world becomes non-deterministic. How is the policy now non-optimal ?

3 Value Iteration in an Uncertain World

You have introduced the non-determinism into the world in Section 2 and seen its effect on the agent's behavior. The value iteration makes an incorrect assumption of a deterministic world and as a result a sub-optimal policy is obtained (what does it mean to be sub-optimal?). Therefore, in this section you will implement non-deterministic value iteration so as to make the agent learn the correct, optimal policy.

In particular, you will have to modify the 'iterate_values()' function in the applet 'Section2.pde' from Section 2 to update the Q-values as follows:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s')$$

Then, learn the policy again till the Q values converge and see whether the agent behaves more optimally.

Deliverable

1. Briefly describe your implementation and the choices you made.
2. Save your code as 'Section3.pde'
3. Provide a screenshot of the Q-values after convergence plus a screenshot of the corresponding policy
4. Discuss how non-determinism changed the Q-values, value function, and the optimal policy with respect to the deterministic case.

4 Q-Learning

We have known how to deal with non-deterministic world in Section 3. Now, suppose that we do not have access to the 'T()' function, i.e., the agent knows nothing about the transition probabilities. As discussed in class, we can still learn an optimal policy by *Q-learning*, which you will implement here.

Modify the function 'learnQ()' in the applet 'Section3.pde' from above to execute one step of Q-learning, i.e., implement the following training rule

$$Q_n(s, a) = (1 - \alpha_n) Q_{n-1}(s, a) + \alpha_n [r + \gamma \max_{a'} Q_{n-1}(s', a')]$$

given an experience tuple (s, a, r, s') , and where α_n is the *learning rate*. You should comment out the code followed by "FOR VALUE ITERATION" and uncomment the code followed by "FOR Q LEARNING" in function 'draw()'. Note that Q-learning will always be performed in the whole running process. Something that you will find very useful is that, by ressing the 'e' key, the agent will *explore* by taking random actions.

Deliverable

1. Briefly describe your implementation and the choices you made.
2. Save your code as 'Section4.pde'
3. Submit two screenshots of the Q-values while learning, one near convergence
4. Discuss how Q-learning behaves with respect to value iteration
5. Discuss why exploration is useful