CS 4758/6758 Robot Learning: Homework 5

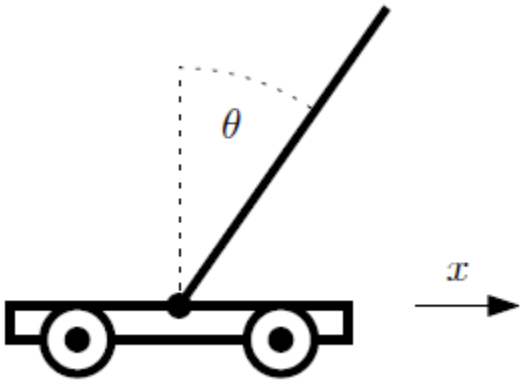
Deadline not yet set

April 17, 2011

1 Reinforcement Learning: The Inverted Pendulum. (70 pts)

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[1]



Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.

We have provided a simple Matlab simulator for this problem. The simulation proceeds in discrete time steps. The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it'd be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 1 to NUM_STATES. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no do-nothing action.) These are represented as actions 1 and 2 respectively in the code. When the action choice is made, the simulator updates the state parameters according ot the underlying dynamics, and provides a new discreteized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitilialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

The files for this problem are in `hw6p1.zip`. Most of the code has already been written for you, and you need to make changes only to `control.m` in the places specified. This file can be run in Matlab, which shows a display and to plot a learning curve at the end (trial # vs logarithm of number of steps to failure). Read the comments at the top of the file for more details on the working of the simulation.[2]

[1] The dynamics are adapted from http://www-anw.cs.umass.edu/rlr/domains.html

[2] Note that the routine for drawing the cart does not work in Octave.Setting min_trial_length_to_start_display to a very large number disables it

To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

Briefly, you will maintain a current model of the MDP and a current estimate of the vaue function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabiliites and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estiamted MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion hecks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several *consecutive* attempts (defined by the paramter NO_LEARNING_THRESHOLD) to sole Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline in this problem is already in `control.m`, and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

Start by running the code as given, which attempts to keep the pole up with a simple, non-learning rule set. As you see, it is not very effective. Print the "learning" curve that this test produces and include it in your submission (you might want to set min_trial_length_to_start_display to a very large number to make it go faster). Once you have written your code, run the script again, print the learning curve that results from that, and include it in your code. Print out your code and include that.

Answer the following questions:

1. Did you direclty utilize knowledge of physics or pendula in solving this problem?

2. Roughly, what is the average logarithm of the number of steps to failure for the initial script run (the one without any learning)?

3. Roughly, what is the average logarithm of the number of steps to failure after learning has occurred (the right side of the second plot you printed)?

4. What would be the advantages and disadvantages of increasing the resolution with which we discretize the states used in this simulatiuon?

5. After running the learning algorithm, take a look at the number of times you were in each state. How many of the NUM_STATES states did you ever actually hit? What does this tell you about the optimality of the solution that the reinforcement learning algorithm found?