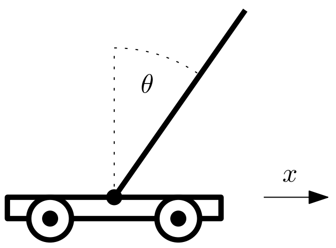
CS 4758/6758: Robot Learning: Homework 6 preview

Due: May 6th

1 Reinforcement Learning: The Inverted Pendulum (50 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 written a simple Matlab simulator for this problem. The simulation proceeds in discrete time cycles (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 to the underlying dynamics, and provides a new discretized 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 reinitialized 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 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 to show a display and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation [2]

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 value function. Initially, each state has estimated reward zero, and the estimated transition probabilities

[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

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 probabilities 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 estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks 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 parameter NO_LEARNING_THRESHOLD) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline for 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 $\gamma = .995$.

Implement the reinforcement learning algorithm as specified, and run it. How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged?

Hand in your implementation of `control.m`, and the plot it produces.

2 Reinforcement Learning MDP

In This problem, we show that MDP is gaarentted to find the optimal policy. Consider an MDP with finite state and action spaces, and discount factor . Let B be the Bellman update operator with V a vector of values for each state. I.e., if $V = B(V)$, then

$$V'(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V(s')$$

(a) Prove that if $V_1(s) \leq V_2(s)$ for all $s \in S$, then $B(V_1)(s) \leq B(v_2)(s)$ for all $s \in S$

(b) Prove that for any V,

$$||B^{\pi}(V) - V^{\pi}||_{\infty} \leq \gamma||V - V^{\pi}||_{\infty}$$
$$where \ \ ||V||_{\infty} = \max_{s \in S} |V(s)|$$

Intuitively, this means that applying the Bellman operator B^{π} to any value function V , brings that value function closer to the value function for π, V^{π}. This also means that applying B^{π} repeatedly (an infinite number of times)

$$B^{\pi}(B^{\pi}(B^{\pi} \cdots B^{\pi}(V) \cdots))$$

will result in the value function V^{π} (a little bit more is needed to make this completely formal, but we will not worry about that here). Use the fact that for any α, $x \in R^n$, if $\sum_i a_i = 1$ and $a_i \geq 0$, then $\sum_i \alpha_i x_i \leq max_i x_i$

(c) We say that V is a fixed point of B if $B(V) = V$. Using the fact that the Bellman update operator is a γ-contraction in the max-norm, prove that B has at most one fixed point -i.e., that there is at most one solution to the Bellman equations. You may assume that B has at least one fixed point.

(d) Now suppose that we have some policy π, and use Policy Iteration to choose a new policy π' according to

$$\pi'(s) = arg\max_{\alpha \in A} \sum_{s' \in S} P_{sa}(s')V^{\pi}(s')$$

Show that this policy will never perform worse that the previous one i.e., show that for all $s \in S, V^{\pi}(s) \leq V^{\pi'}(s)$. In order to show it, first, show that $V^{\pi}(s) \leq B^{\pi'}(V^{\pi})(s)$, then use the a) and b) to show that $B^{\pi'}(V^{\pi})(s) \leq V^{\pi'}(s)$.