XCS229ii Problem Set 2 — Inverted Pendulum

Due Sunday, 1 November at 11:59pm PT.

Guidelines

1. These questions require thought, but do not require long answers. Please be as concise as possible.
2. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs229ii-scpd.slack.com/
3. Familiarize yourself with the collaboration and honor code policy before starting work.
4. For the coding problems, you may not use any libraries except those defined in the provided started code. In particular, ML-specific libraries such as `scikit-learn` are not permitted.

Submission Instructions

Coding Submission: All assignment code is in the `src/` subirectory. You will submit only the `src/submission.py` file. Please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` will be used to autograde your submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
python grader.py
```

There are two types of unit tests used by our autograders:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases.
- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. In the student version of `src/grader.py`, only the setup and inputs to these unit tests are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder.

For debugging purposes, a single unit test can be run locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
python grader.py 3a-0-basic
```

Before beginning this course, we highly recommend you walk through our Anaconda Setup for XCS Courses to familiarize yourself with our coding environment. Please use the env defined in `src/environment.yml` to run your code. This is the same environment used by our autograder.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

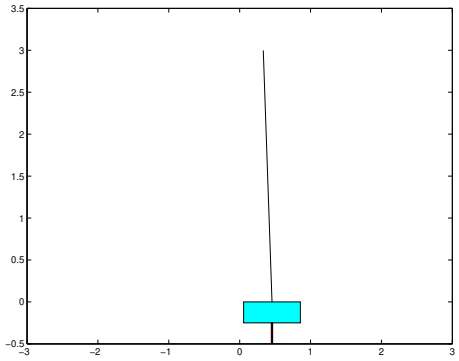
1. **[40 points (Coding)] Reinforcement Learning: The inverted pendulum**

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 Environment

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 or right.



We have written a simple 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 would 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 0 to `NUM_STATES-1`. 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 0 and 1 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.

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.

Learning Algorithm

The files for this problem are in the `src/` directory. Most of the the code has already been written for you in `src/cartpole.py` and `src/env.py`. Like all coding assignments in this course, you need only make changes only to `src/submission.py`. The other files will import the functions that you have implemented. The comments at the top of `src/cartpole.py` provide more details on the structure of the simulation.

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 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

[1] The dynamics are adapted from https://en.wikipedia.org/wiki/Inverted_pendulum

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.

The code outline for this problem is already in `src/cartpole.py`. There are several details (convergence criteria etc.) that are explained inside this file. Use a discount factor of $\gamma = 0.995$. We will assume that the reward $R(s)$ is a function of the current state only.

Implement the reinforcement learning algorithm as specified, and run it with the following command form within the `src/` directory.

```
(CS229ii) $ python cartpole.py
```

The Python starter code will plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. As a sanity check, your resulting plot should like similar to the one shown below, although it can vary significantly. A correct algorithm should have very little trouble balancing the pole for at least 200 time steps.