OLCAR - Exercise 2
Handout: 21.04.2015
Due: 06.05.2015

ADRL

Introduction

In this exercise you will solve an optimal control problem on two different simple toy systems. Different algorithms will be used to solve the two problems, allowing you to get experience implementing a wide range of Reinforcement Learning methods.

On the course website `http://www.adrl.ethz.ch/doku.php/adrl:education:lecture:fs2015` the software framework for the exercise can be downloaded. Please complete the missing sections and upload your `*_Design.m` files through the appropriate form on the website. Additionally upload a .pdf with the answers to the posed questions (no more than 3 sentences per question, max. 1 A4 page in total). All of this material must be uploaded by **06.05.2015, 11:59 pm**.

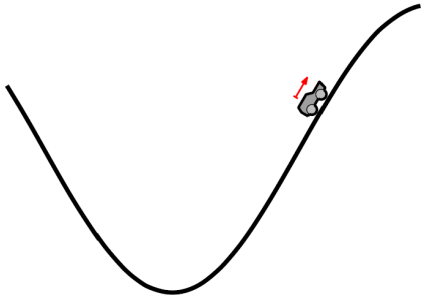
Exercise 2a: Mountain Car

In this problem an under-powered car must go up a hill which is too steep for it to drive up directly. Instead it should bounce back and forth to gain enough velocity to climb up. We will consider a two dimensional state space of position x and velocity v. The range of allowed positions is $[-1.2, +0.5]$ and the velocity is bounded in the range $[-0.07, +0.07]$. The car's actuation is represented as an acceleration, a, in the range $[-1, +1]$. The dynamic equation of the car is given as:

$$v(t+1) = bound\left[v(t) + 0.001a(t) - 0.0025\cos\left(3x(t)\right)\right]$$
$$x(t+1) = bound\left[x(t) + v(t+1)\right],$$

where the function $bound$ maintains the value of x and v within the limits. Whenever the car reaches the position limits, its velocity is set to zero so that it remains there indefinitely. When the car reaches the top of the hill, it gets a reward of $+10$. Otherwise, it gets a reward of -1 for every time step in which it doesn't reach the top.

Step 1: Because the system has continuous states and actions, the first step of this exercise will be to discretize the system so that MDP-based Reinforcement Learning methods can be applied. You will see that discretization is not trivial for this system. To effectively discretize the system, a stochastic discrete model should be created, even though the continuous system is deterministic.



The mountain car problem: An under-powered car must go up a steep hill

Step 2: The second step of the problem will be to implement the Generalized Policy Iteration (GPI) algorithm and use it to find an optimal policy for the discretized system modeled in Step 1. You should then simulate the system using the controller found and analyze its performance. The algorithm should be tested with the algorithm set to perform Policy Iteration and Value Iteration.

Software: All of the code for the Mountain Car problem is in the `/Mountain_Car` directory of the provided code. The code should be run using the `./main_ex2a.m` script. Solutions to the two steps of the problem should be implemented in the provided `./Design_functions/MDP_Design.m` and `./Design_functions/GPI_Design.m` functions. The system dynamics are simulated for a single time-step in the function `./Model/Mountain_Car_Single_Step.p` and over an entire trajectory in `./Model/Mountain_Car_Simulator.p`. Simulation output can be plotted and visualized using `./Visualization/visualizeMountainCar.p`. Working solutions are provided in the functions `./Design_functions/MDP_Design_Solution.p` and `./Design_functions/GPI_Design_Solution.p`. These functions can be run using different tuning parameters by modifying the values set in the corresponding `*_Params` struct in `./main_ex2a.m`.

The following table lists important parameters for both steps of the problem, along with suggested nominal values which have been found to work sufficiently using the solution code. These values are not the best possible values, but are instead intended to serve as a benchmark during algorithm development.

Exercise 2a Questions:

Question 1: Build a probabilistic model of the system. What is the stochastic element in the modeling process and what is its significance? What modeling parameters have the most effect on the quality of the solution?

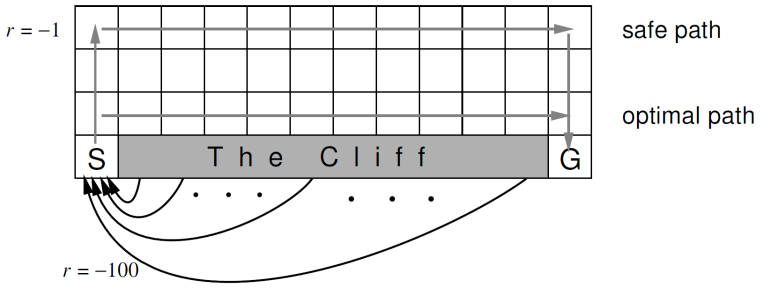
Parameter	Description	Value
`pos_N`	Number of bins used for car position discretization	20
`vel_N`	Number of bins used for car velocity discretization	20
`u_N`	Number of bins used for car action discretization	5
`modeling_iter`	Number of modeling iterations performed for each state-action pair. When set to 1, this builds a deterministic model	50
`maxIter_PE`	Maximum number of iterations allowed during Policy Evaluation	100
`maxIter_PI`	Maximum number of iterations allowed during Policy Improvement	100
`minDelta_V`	Minimum change in the value function allowed before terminating Policy Evaluation	0.01
`minDelta_Policy`	Minimum change in the policy allowed before terminating Policy Improvement	0.1
`alpha`	Decay factor used when calculating the accumulated reward	0.95

Question 2: Implement the Generalized Policy Improvement algorithm introduced in Section 2.8.3. Use appropriate terminal conditions for Policy Evaluation and Policy Improvement processes and implement the Policy Iteration and the Value Iteration algorithms. Think about what the optimal solution to this task should be for the Mountain Car system. Was the learning algorithm able to find this solution? If not, why do you think that is the case?

Question 3: Now build a deterministic discrete state-action space model of the system (i.e. set the number of modeling iterations to 1). Is it possible to find a policy which reaches the goal? What problems are faced when discretizing the system in this way?

Exercise 2b: Cliff World

The Cliff World problem is a grid world problem used for highlighting the difference between on-policy (Monte Carlo) and off-policy (Q-learning) learning methods. A map of the cliff world is shown below. In this task, an agent starts from S and tries to reach G along shortest path without falling off of the cliff. At each time step, the agent can move up, down, left, or right. The reward is -1 for all of the transitions except those which move the agent to regions marked 'cliff', where instead the agent gets a reward of -100 and is sent back to the start. Once the agent reaches G, the task is terminated. The system dynamics are deterministic and agent always moves where it intends. Despite the simplicity of the problem, learning an optimal policy without any prior information about the rewards in the grid world can be cumbersome.



The Cliff Walking problem map

Step 1: The first step of the Cliff World problem is to implement the On-policy Monte Carlo method to find an optimal controller which leads the agent to the goal using the shortest path possible. To do this, you should generate training episodes in which the agent follows some ϵ-greedy policy. After each episode, the policy should be improved based on the observations of that episode using model-free Policy Improvement. For this problem you only need to consider a constant ϵ, though the algorithm will also work with ϵ decreasing over subsequent iterations.

Step 2: Now solve the same optimal control problem, but using the off-policy Q-Learning algorithm. Once again, you should generate training episodes in which the agent follows an ϵ-greedy policy. Learning should now be done during the training episodes, however. The algorithm should be tested for both a constant ϵ and ϵ decreasing over subsequent iterations.

Software: The software structure for exercise 2b is very similar to that of 2a. The code is given in `/Cliff_World`. The code should be run using the `./main_ex2b.m` script. Solutions to the two steps of the problem should be implemented in the provided `./Design_functions/Monte_Carlo.m` and `./Design_functions/Q_Learning.m` functions. The system dynamics are simulated using `./Model/Cliff_World.p`. Learning progress can be plotted after every iteration of each algorithm using `./Visualization/RewardPlotter.p` and the final path followed by a policy executed greedily on the learned Q-table is plotted using `./Visualization/VisualizeCliffWorld.p`. Working algorithms are provided in the functions `./Design_functions/Monte_Carlo_Solution.p` and `./Design_functions/Q_Learning_Solution.p`. These functions can be run using different tuning parameters by modifying the values set in the corresponding `*_Params` struct in `./main_ex2a.m`.

The following table lists important parameters for the two algorithms to be implemented in this exercise. Nominal values are given separately for each algorithm. These values are not the best possible values, but are instead intended to serve as a benchmark during algorithm development. In particular, the Monte Carlo

algorithm may not converge to a valid solution with the parameters given due to the stochasticity in the policy execution.

Parameter	Description	MC Value	QL Value
`epsilon`	Initial value for the 'greediness' of the policy. Set this to 0.0 for a purely greedy policy	0.3	0.1
`k_epsilon`	Gain applied to ϵ after each learning iteration. $\epsilon_{n+1} = \epsilon_n * k_epsilon$	1	0.995
`training_iterations`	Number of training episodes	500	500
`episode_length`	Maximum number of time steps to evaluate in each training episode	500	500
`omega`	Learning rate	0.1	0.1
`alpha`	Decay factor used when calculating the accumulated reward	1	1

Exercise 2a Questions:

Question 4: First implement the Monte Carlo algorithm. Test the algorithm using different values of ϵ. What impact does ϵ have on the solution? Can the algorithm find the optimal greedy policy?

Question 5: Now implement the Q-Learning algorithm. First test the algorithm with decreasing exploration during learning (i.e. `k_epsilon`< 1). Next, test the algorithm using constant exploration during learning (i.e. `k_epsilon`$= 1$). How does `k_epsilon` influence the solution?

Question 6: Compare the computational efficiency and performance of the Monte Carlo and Q-Learning algorithms. What are the fundamental reasons why one algorithm performs better than the other?