TEK5040 Assignment, MDPs, Value Iteration, Policy Iteration and Q-learning

Li Meng and Narada Warakagoda

NOTE: This assignment is optional. Therefore, no submission is required. But you are strongly encouraged to perform the tasks in this assignment

1 Introduction

In this assignment you will explore **Markov Decision Processes**, and algorithms to find the optimal policy within them. You will implement value iteration, policy iteration, and tabular Q-learning and apply these algorithms to simple environments including tabular maze navigation (FrozenLake) and controlling a simple crawler robot. The problems are taken from Deep RL Bootcamp. There are three python files:

`Problem 1.py` Script for Value Iteration
`Problem 2.py` Script for Policy Iteration
`Problem 3.py` Script for Tabular Q-learning

In the **FrozenLake** environment, we know the environment dynamics, and you will use them to solve Problem 1 and 2. For the crawler robot (Problem 3) we only interact with the environment through sampling. As an extra challenge you may also try to solve Problem 1 and Problem 2 them by only sampling from the environment!

2 Setup

You will need the packages `matplotlib`, `numpy`, and `gym` (here you need version 0.9.2):

```
pip3 install matplotlib numpy gym==0.9.2
```

If you use anaconda, you could install what you need with:

```
conda env create -f environment.yml
```

where environment.yml is in the same folder as your assignment.

3 Task

3.1 Problem 1: implement value iteration

This assignment will review the two classic methods for solving Markov Decision Processes (MDPs) with finite state and action spaces.

We will implement value iteration (VI) and policy iteration (PI) for a finite MDP, both of which find the optimal policy in a finite number of iterations.

The experiments here will use the **FrozenLake** environment, a simple gridworld MDP that is taken from 'gym' and slightly modified for this assignment. In this MDP, the agent must navigate from the start state to the goal state on a 4×4 grid, with stochastic transitions.

Winter is here. You and your friends were tossing around a frisbee at the park. When you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water.At this time, there's

an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc.

States
The surface is described using a grid like the following:

S F F F
F H F H
F F F H
H F F G

- S : starting point, safe
- F : frozen surface, safe
- H : hole, fall to your doom
- G : goal, where the frisbee is located

Being at each grid position is a state, i.e. there are 16 states. State are indexed from 0 to 15 from top to bottom and left to right. The episode ends when you reach the goal or fall in a hole.

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

Rewards
You receive a reward of 1 if you reach the goal, and zero otherwise.

Actions and state transitions
At each state, there are 4 actions, move to west, south, east and north. They are indexed as 0,1,2,3. However, the ice is slippery, so you won't always move in the direction you intend. That means that for example, even if you take action 0 (west) from state 1, it is not guaranteed that your end state is 0. It can be state 0,2 or 5 with some probabilities. The gym environment simulates these state transitions.

In this problem, you'll implement value iteration, which has the pseudo-code in Algorithm 1.

Algorithm 1 Value Iteration

Initialize: $V^{(0)}(s) = 0$, for all s
For $i = 0, 1, 2, \cdots$:
  $V^{(i+1)}(s) = \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^{(i)}(s')]$, for all s

We additionally define the sequence of greedy policies $\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(n-1)}$:

$$\pi^{(i)}(s) = \arg\max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^{(i)}(s')]$$

Your code will return two lists: $[V^{(0)}, V^{(1)}, \ldots, V^{(n)}]$ and $[\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(n-1)}]$

To ensure that you get the same policies as the reference solution, choose the lower-index action to break ties in $\arg\max_a$. This is done automatically by np.argmax. This will only affect the "# chg actions" printout in

the code–it won't affect the values computed.

Warning: make a copy of your value function each iteration and use that copy for the update–don't update your value function in place. Updating in-place is also a valid algorithm, sometimes called Gauss-Seidel value iteration or asynchronous value iteration, but it will cause you to get different results than our reference solution (which in turn will mean that the testing commands won't be able to help in verifying your code).

3.2 Problem 2: implement policy iteration

The second task is to implement exact policy iteration (PI), which has the pseudo-code in in Algorithm 2.

Algorithm 2 Policy Iteration

Initialize: π_0
For $n = 0, 1, 2, \cdots$:
  Compute the state-value function V^{π_n}
  Using V^{π_n}, compute the state-action-value function Q^{π_n}
  Compute new policy $\pi_{n+1}(s) = \text{argmax}_a \, Q^{\pi_n}(s, a)$

You'll first write a function called **compute_vpi** that computes the state-value function V^π for an arbitrary policy π.

Recall that V^π satisfies the following linear equation:

$$V^\pi(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

Solving a linear system is rather straightforward in the matrix form. When we have $AX = B$, we can solve for $X = A^{-1}B$. You can solve a linear system in your code simply using an exact solution, e.g., with **np.linalg.solve**.

Next, you'll write a function to compute the state-action value function Q^π in **compute_qpi** , defined as follows:

$$Q^\pi(s, a) = \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^\pi(s')]$$

Then, we will be ready to run the policy iteration using **compute_vpi** and **compute_qpi**.

3.3 Sampling-based Tabular Q-Learning

So far we have implemented Value Iteration and Policy Iteration, both of which require access to an MDP's dynamics model. This requirement can sometimes be restrictive. For example, if the environment is given as a black-box physics simulator, then we won't be able to read off the whole transition model.

We can however use sampling-based Q-Learning to learn from this type of environments. For this exercise, we will learn to control a Crawler robot. First, try some completely random actions to see how the robot moves and familiarize ourselves with Gym environment interface again.

You will see the random controller can sometimes make progress but it won't get very far. Next, implement Tabular Q-Learning with ϵ-greedy exploration to find a better policy piece by piece.

After we observe a transition s, a, s', r, the Q learning update is defined by:

$$\text{target}(s') = R(s, a, s') + \gamma \max_{a'} Q_{\theta_k}(s', a')$$

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha) Q_k(s, a) + \alpha \left[\text{target}(s')\right]$$

In practice, however, the update can also be taken in a simpler manner. Use this instead in your code:

$$Q_{k+1}(s, a) \leftarrow \alpha \left[\text{target}(s') - Q_k(s, a)\right]$$