UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ROBOTICS
AND COGNITIVE SYSTEMS

Course: Reinforcement Learning (SS2020)
Dr. Christoph Metzner & Honghu Xue, M.Sc.
Assignment Report Nr. I
Date: 5th May 2021

Team Nr. 1
Rakesh Reddy Kondeti
Christopher Schmale
Tim-Henrik Traving
Contact: timhenrik.traving@student.uni-luebeck.de

1 Basics of Reinforcement Learning

1.1 Markov Decision Process

A Markov Decision Process is a Tuple (S, A, P_a, R_a, γ), where

- S, a set of states, is the *state space*,
- A, a set of actions, is the *action space*,
- $P_a(s, s') = Prob(s_{t+x} = s'|s_t = s, a_t = a)$, the probability that an action a taken in state s at time t will lead to state s' at $t + 1$. Here, P is called the state transition probability matrix.
- $R_a(s, s')$ is the reward for a transition from state s to s' by action a. $R_a(s) = E[R_{t+1}|S_t = s, A_t = a]$.
- γ is called the discount factor, $\gamma \in [0, 1)$. It determines how much importance is to be given to the immediate reward and future rewards.

If all states in the state space satisfy the *first-order Markovian property*, then $P(S_{t+x}|S_t) = P(S_{t+1}|S_t, S_{t-1}, ..., S_1)$. This means that the future is independent of the past given the present.

1.2 Goal of RL

The goal of the Reinforcement Learning problem is to maximize the expected cumulative rewards, thereby allowing the agent to learn the optimal policy.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \Sigma_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{1}$$

- A *reward* R_t is a scalar feedback signal that indicates how well an agent is performing at time point t.
- The *return* G_t is the expected total rewards from time step t into the future,
- The parameter $\gamma \in [0, 1)$ is a *discount factor* to avoid infinite returns. When $\gamma = 0$, the total return G_t only depends on immediate reward R_{t+1}.

The *goal* of a reinforcement learning problem is the maximization of the return,

$$\max(G_t). \tag{2}$$

In other words: The goal of an RL problem is that an agent shall perform as well as possible.

1.3 Policies

A *policy*(π) is an agent's behaviour. A *policy* is a mapping from state to actions. Simply speaking, a *policy* defines what actions to perform in a particular state s.

$$\pi : S \to A \tag{3}$$

Two different kinds of policies exists:

- *Deterministic* policy: Given the state $s \in S$, the agent always takes the same action with probability 1, unless the policy changes
$$\pi(s) = a \tag{4}$$
- *Stochastic* policy: A stochastic policy is conditional probability distributions,$\pi(a|s)$, from the set of states, $s \in S$, to the set of actions, $a \in A$
$$\pi(a|s) = P(A_t = a|S_t = s) \tag{5}$$

1.4 Continuing vs. episodic tasks

Episodic tasks are those tasks that have a terminal state(end state). Here we have finite number of states. These episodes are considered as agent-environment interactions from initial to final states, and whenever an agent reaches the goal, it starts over again and again until reaching certain number of loops.It maximizes the continuous reward.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T, \tag{6}$$

where R_T is the final time step.
For example, you play a game of snake. Every single game is executed as a single task. The reward will be given at the end of each game or during the game after certain actions. The reward is measured at the score reached in the game for example.

Continuous tasks continuing task is one where there is no terminal state, and the agent goes on forever in the environment setting.It maximizes the continuous reward and discount rate.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{7}$$

where R_T is the final time step.
For example, you want to teach the system how to code. The system searches the internet continuously without an end. For archiving new abilities, the system gets rewards during the learning process.

2 Dynamic Programming

2.1 Formal Definitions

2.1.1 State- & Action-Value Function

Given a policy π and a state s, the *state value function*

$$v_\pi(s) = E_\pi\left[G_t|S_t = s\right] = E_\pi\left[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t\right] \tag{8}$$

describes the expected total return G, at time step t starting from the state s at time t and then following the policy π. In other words, it describes how good or bad it is to be in a particular state s according to the return G when following a policy π.

The *action-value function* is slightly different:

$$q_\pi(s, a) = E_\pi\left[G_t|S_t = s, A_t = a\right] = E_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, a_{t+1})|S_t = s, A_t = a\right]. \tag{9}$$

It defines the value of taking action a in state s under a policy π, as the expected return G starting from s, taking the action a and thereafter following policy π. The Q-functions defines how good it is, to do action a when it is in state s.

2.1.2 Optimal State Value and Action Value Functions

The policy, which maximizes the total cumulative reward is called the optimal policy $\pi*$.
A policy π' is defined to be better than or equal to a policy π if and only if $V_{\pi'}(s) \geq V_\pi(s)$ for all states s.
An optimal policy $\pi*$ satisfies $\pi* \geq \pi$ for all policies π. The definition of an optimal policy defines a partial ordering over policies:

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s \tag{10}$$

Following this definition the state-value function $v_{\pi_*}(s)$ over the optimal policy $\pi_* \geq \pi, \forall \pi$, yields at least the same return, if not better, as it does for any other policy:

$$v_{\pi_*}(s) \geq v_{\pi}(s), \forall \pi \tag{11}$$

The optimal value function is one which yields maximum value compared to all other value function (following using other policies).

$$V_{\pi *}(s) = \max_{\pi} V_{\pi}(s), \forall s \tag{12}$$

In the above formula, $v_*(s)$ tells us what is the maximum reward for state s we can get from the system.

Similarly, optimal state-action value function indicates the maximum reward we are going to get if we are in state s and taking action a from there on-wards

$$Q_{\pi *}(s,a) \geq Q_{\pi}(s,a), \forall s,a \tag{13}$$

$$Q_{\pi *}(s,a) = \max_{\pi} Q_{\pi}(s,a) \tag{14}$$

2.1.3 One or many optimal Policies?

An optimal policy π^* is a policy such that $\pi^* \geq \pi, \forall \pi$. This does not imply that there is only one optimal policy. There might be infinitely many optimal policies. These policies only have to achieve the optimal value function $v_{\pi^*} = v_*$ and action-value function $q_{\pi^*} = q_*$. The value $q_{\pi^*}(s, a^*), a^* = \pi^*(s)$ is the maximum for all possible actions, $q_{\pi^*}(s, a^*) = \max_{a' \in A} q_{\pi^*}(s, a')$.

2.1.4 Estimation

No, action a does not need to follow the the current policy distribution π (π^*) when estimating $Q_\pi(s,a)$ ($Q_{\pi^*}(s,a)$). However, the actions taken to estimate the value from the state s', the goal state when starting in state s and taking action a, on have to follow the policy distribution π (π^*).

2.2 Bellman Equation

The *Bellman Equation* decomposes the value function into two parts: Immediate reward and discounted feature values. This simplifies the computation of a value function, such that summing up multiple time steps. We can find the optimal solution of a complex problem by breaking it down into simpler recursive subproblems and finding their optimal solutions.
The formula for the bellman equation is:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \tag{15}$$

The bellman expectation equation for the state-value function is:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right) \tag{16}$$

The bellman expectation equation for the action-value function is:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s'|a') \tag{17}$$

2.3 Generalized Policy Iteration

The *generalized policy iteration* is used to describe the abstract idea behind the interaction of its two parts: the policy *evaluation* and the policy *improvement*. The policy evaluation uses some form of evaluation algorithm in order to estimate the state-value function v_π. Based on the results, some form of policy improvement algorithm is used to derive a new policy $\pi' \geq \pi$. Without loss of generality, this improvement algorithm can assumed to be greedy, as the value-function is directly dependent on the policy. This cycle repeats until both processes stabilize. In this case it can be reasonably assumed that an optimal policy π^* and an optimal state-value function V^* have been found.

3 Programming Part on Dynamic Programming

3.1 Transition Dynamics

In the `FrozenLake-v0`-environment an agent has to move from a starting position, across a frozen lake, to a goal. Besides the start and goal, the water at a position on the lake can either be frozen or not, which means there is a hole in the ice. In order to navigate the lake (ideally without drowning/freezing to death in a hole) an agent can perform four distinct actions:

- Move *up*
- Move *down*
- Move *left*
- Move *right*

When moving into a direction, the agent is guaranteed to not move into the opposite direction. However, the agent has a probability of $\frac{1}{3}$ to move into any of the other three directions (probably because the ice is so slippery). As example, if an agent tries to move right, he/she will surely not move left. However, the agent has only a chance of $\frac{1}{3}$ of actually moving to the position on the right. He/She might just as easily move a position up or down.
If an action would cause the agent to leave the defined space (if the agent would move over the „edge" of the charted space), the agent will remain in its current state.

3.2 Value Iteration

The value iteration is implemented in the "`value-iteration-frozen-lake.py`" file,
SHA-256 Hash: `e8c5537b21e2174f75b28d8a14c3a3915cbe07df6e7f41a4b605555361184488`.
The final learned policy as well as the state values for each step are shown in table 1. Running this policy over 100 episodes we took average of 36 steps to get to the goal and we fell in a hole 27% of the times.

0.82352941	0.82352941	0.82352941	0.82352941
0.82352941	0	0.52941176	0
0.82352941	0.82352941	0.76470588	0
0	0.88235294	0.94117647	0

(a) State Values

←	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

(b) Policy

Table 1 Value Iteration Results

3.3 Policy Iteration

The policy iteration is implemented in the ”`policy-iteration-frozen-lake.py`” file,
SHA-256 Hash: `02889e967bbed607c6c184d05bca2f37344e958f3eb49f847ffc00ab3df7e2fd`.
The final learned policy as well as the state values for each step are shown in table 2 Running this policy over 100 episodes we took an average of 40 steps to get to the goal and fell in a hole 23% of the times.

0.82335966	0.82330397	0.82326527	0.82324557
0.82337674	0	0.52929602	0
0.82340348	0.82343777	0.76462558	0
0	0.88228925	0.94114406	0

(a) State Values

←	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

(b) Policy

Table 2 Policy Iteration Results

3.4 Comparison

Method	Steps	Execution Time
Value Iteration	1630	2.17s
Policy Iteration	1552	0.97s

Table 3 Comparison of the two methods.

It can be seen, that the derived optimal policy(π^*) is the same for both the value- and the policy-iteration. The calculated state values($V_{\pi^*}(s)$) have a small difference, which could be neglected. These differences might origin from the different initial policies, which are initialized at random. Also, the value iteration has fewer steps that the policy iteration, but this too might change with different initial policies.
The final policies were tested 100 times (See 3.2 and 3.3). While the average number of steps and the success rate were different, it is to be expected that those converge to the same values for more tests.
The optimal trajectory from start to goal state can be achieved by following the learned policy π^*.