Middle East Technical University | Department of Computer Engineering

CENG 462

Artificial Intelligence

Fall '2022-2023

Homework 5

Due date: 8 January 2023, Tuesday, 23:55

1 Objectives

This assignment aims to familiarize you with three Temporal Difference (TD) learning methods, Q-learning, and SARSA (State-Action-Reward-State-Action) (which are direct approximations for Value Iteration and Policy Iteration methods) at the implementation level and provide hands-on experience with these algorithms.

2 Problem Definition

Markov Decision Process (MDP) is a mathematical framework for modeling sequential decision-making problems, where an action consequence is modeled with probability distributions and aims to extract optimal action sequences. It is a tuple of (S, A, T, R), where S is the set of states of the problem, A is the set of actions which are available in each state, R is the reward function that governs the outcome of an action (reward/cost) in a particular state and T is the state-transitioning probability distribution which models the dynamics of the problem. Interacting with this framework is carried out in discrete time steps by the problem solver entity (agent) and in each time step a new action is fed and its consequences are observed (next state, reward/cost). Solving an MDP corresponds to extracting an action sequence (policy) that maximizes/minimizes the cumulative expected reward/cost: $E[\sum_{t=0}^{\infty} \gamma^t R_t]$ (where γ regulates the importance of future reward/cost values in decision-making).

In this assignment, we shift our focus to more realistic problems where we are devoid of having the knowledge of the transition and reward functions completely and attempt to learn optimal policies. In this case, we employ the algorithms that approximate them (model-free methods). In most of the problems, actually we seek an optimal policy, which is the solution to the given problem. Instead of learning only state utility scores, we could learn utility values for state-action pairs, from which the policy can directly be extracted. Q-learning and SARSA methods operate on state-action pairs and can be regarded as a direct approximation to Policy Iteration. The main difference between these two techniques is whether the policy used for learning is different from the actual behavior policy (the policy that is actually applied to a problem). Q-learning learns utility scores without necessarily following its behavior policy hence it is called an off-policy method. The update rule for Q-learning is as follows:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{1}$$

While obtaining the utility value of the transitioned (next) state the max operator is used. This is where its "off-policy"ness lies since it does not have to take the action whose utility score is the highest

in the next state among all other candidate actions. In contrast, SARSA leverages its behavior policy for learning utility scores and is called an on-policy method. The update rule for SARSA is as follows:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma Q(s',a') - Q(s,a)) \tag{2}$$

where a' is the action to be taken in s'.

Due to the lack of knowledge of transition and reward functions, agents have to interact with problems and learn from the experience that they gather. During the interaction, the agent should enrich its knowledge about the problem (environment) with exploratory moves, at the same time it has to act according to the information it has learned (exploitation) so far in order to solve the problem. This confliction (how much the agent should explore and how much it should depend on its gathered experience) is coined as the exploration-exploitation trade-off. In this assignment, we utilize the ϵ-greedy policy (with a probability of ϵ, the agent takes an exploratory action, e.g. random action) as a solution to this trade-off.

In this assignment, our focus is on game-playing environment and you are expected to implement Q-learning, and SARSA methods in order to solve tic-tac-toe.

2.1 Tic-tac-toe Game

It is a very-well known two-player game where both opponents contend for marking one of the rows or columns or diagonal cells of the 3x3 game board completely with their corresponding symbols (`X`, `O`). The game is played in turns and each opponent may mark any available location (empty locations) with their symbol.To formulate reinforcement learning problem, as mentioned above, state, action, and reward are used.The state of this game is the board state of both the agent and its opponent, so we will initialise a 3x3 board with zeros indicating available positions and update positions.The action is what positions a player can choose based on the current board state. At the end of game, 1 is rewarded to winner and 0 to loser.

- If player X wins, it gets 1, and player O gets -1.
- If player O wins, it gets 1, and player X gets -1.
- If game result is drawn,they get 0.

3 Simulation

In order to realize the interaction between an environment and an agent, we need to create a simulation for the problem where the agent starts from a initial game state and applies an action on each state till game ends.Solving the problem once is not sufficient to calculate correct state values in the problems so the agent has to visit every possible state multiple times over a sufficient period of time in order to learn their correct utility scores. We are going to simulate them each game as episodic tasks. An episode refers to the agent's journey (all state-action-reward sequence) starting from a initial state and ending game. After a game end, a new episode is started and the the game is initialize with zeros in the problem. During the episodes, the agent amasses experience (history of states, rewards, actions). Utilizing one of the methods of this assignment it can extract the optimal policy for the solution of the problem after getting exposed to sufficiently many episodes.

For this assignment, during the simulation execution, there are one place where randomization should take place: ϵ-greedy policy. For this purpose, we are going to employ the random package of Python. While sampling depending on the ϵ-greedy policy the following code piece should be used:

```
if random.random() <= epsilon:
    selected_action = actions[np.random.randint(0, len(possible_actions)-1)]
else:
    # For Q-learning: select the action whose utility score is the highest
```

4 Algorithms

Figures 1, 2 show pseudo codes of the methods. SARSA and Q-learning algorithms.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 1: SARSA pseudo code [1]

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2: Q-learning pseudo code [1]

5 Specifications

- You are going to implement SARSA and Q-learning in Python 3.
- Each MDP problem is represented with a file and this file will be fed to your implementation along with one of the methods.
- To be able to run all methods from a single place, you are expected to write the following function:

```
def SolveMDP(method_name, problem_file_name, random_seed):
```

  The parameter specifications are as follows:

  - `method_name`: specifies which method to be run for the given MDP problem. It can be assigned to one of the values from `[ "SARSA", "Q-learning"]`.
  - `problem_file_name`: specifies the path of the MDP problem file to be solved. File names take values such as "mdp1.txt", "mdp2.txt", "mdp3.txt".
  - `random_seed`: specifies the value for the random generator package (for the random.seed function).

  During the evaluation process, this function will be called and returned information will be inspected. The returned information should be as follows for all methods:

```
Q_table = SolveMDP(method_name, problem_file_name, random_seed)
```

  - `Q_table`: is the value table that contains the information the reinforcement agent has about the tic-tac-toe policy(e.g. {(0,0): [('-O-XO-X-O': -0.75),('-OXOXOOXX': 0.875),...], (0,1): ['(–X—O-X': 0.225),('-OXOXOOXX': 0.875),...]...})

  Note 1: In the given sample outputs, the visited nodes are represented with 9-character strings, which basically corresponds to the game state in that particular node. The first 3 characters, the next 3 characters, and the last 3 characters depict the first, second, and third rows of a state, respectively.

  Note 2: While training your agent, Set Player X to given learning method. For example, SolveMDP("SARSA" , " mdp1 . txt " , 37) set the Player X to Sarsa and set O player to Q-learning, and vice versa.

- The grid-world MDPs are described in ".txt" files and have the following structure:

```
[alpha]
alpha value
[gamma]
gamma value
[epsilon]
epsilon value
[episode count]
episode count value
```

MDP information and algorithm parameters are provided in separate sections and each section is formed with '[]'. They are as follows:

- `[alpha]` section specifies the α parameter used in algorithms.
- `[gamma]` section provides the value of the γ parameter for all methods.
- `[epsilon]` section specifies the value for the ϵ parameter that is used for ϵ-greedy policy sampling.
- `[episode count]` section provides the value for the k parameter of simulation. It specifies how many episodes should be run in the simulation for learning.

- No `import` statement is allowed except for **copy** and **random** modules with which you may perform a deep copy operation on a list/dictionary and sample a random number, respectively. All methods should be implemented in a single solution file. You are expected to implement all the necessary operations by yourself.
- Commenting is crucial for understanding your implementation and decisions made during the process. Your implementation can be manually inspected at random or when it does not satisfy the expected outputs.

6 Sample I/O

Here, in addition to textual outputs, visual outputs for the solution of the MDP problems are provided. The actions are depicted via their symbol. The utility values are shown at the top left of the states.

mdp1.txt:

```
[alpha]
0.1
[gamma]
0.9999999999
[epsilon]
0.1
[episode count]
10000
```

Expected outputs of the SolveMDP function with all methods applied on mdp1.txt:

```
>>> import hw5solutioneXXXXXXX as hw5
>>> hw5.SolveMDP("SARSA", "mdp1.txt", 37)
({(0, 0): [('-----O---', 0.0), ('-----OX--', 0.0), ('-----OX-O',
   0.4474170622969852), ('---------', 0.0), ('---X-----', 0), ('---X-O---',
   0.0), ('---X-O--X', 0), ('---XOO--X', 0), ('---XOO-XX', 0), ('--OXOO-XX',
   0.0)...],...)
>>> hw5.SolveMDP("Q-learning", "mdp1.txt", 462)
({(0, 0): [('---------', 0.0), ('--X--O---', 0.0), ('-------X-', 0), ('-O-----X
   -', 0.0), ('-O-X---X-', 0), ('-O-X---XO', 0), ('-O-X--XXO', 0), ('-O-XO-XXO
   ', 0.16390000000000002), ('----O--X-', 0.0), ('-X--O--X-', 0),...],...)
```

7 Regulations

1. **Programming Language:** You must code your program in Python 3. Since there are multiple versions and each new version adds a new feature to the language, in order to concur on a specific version, please make sure that your implementation runs on İnek machines.

2. **Implementation:** You have to code your program by only using the functions in the standard module of python. Namely, you **cannot** import any module in your program (except the **copy** module).

3. **Late Submission:** No late submission is allowed. Since we have a strict policy on submissions of homework in order to be able to attend the final exam, please pay close attention to the deadlines.

4. **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from the internet included) will be punished according to the university regulations.

5. **Discussion:** You must follow ODTUClass for discussions and possible updates on a daily basis. If think that your question concerns everyone, please ask them on ODTUClass.

6. **Evaluation:** Your program will be evaluated automatically using the "black-box" technique so make sure to obey the specifications. A reasonable timeout will be applied according to the complexity of test cases. This is not about the code efficiency, its only purpose is avoiding infinite loops due to an erroneous code. In case your implementation does not conform to expected outputs for solutions, it will be inspected manually so commenting will be crucial for grading.

8 Submission

Submission will be done via OdtuClass system. You should upload a **single** python file named in the format **hw5_e<your-student-id>.py** (i.e. hw5_e1234567.py).

9 References

1. SARSA, and Q-learning methods, you can refer to: Reinforcement Learning: An Introduction R. Sutton, and A. Barto. The MIT Press, Second edition, (2018) (Chapter 6)

   (https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf).

2. Announcements Page

3. Discussions Page