Project 3 - Reinforcement Learning

B(E)4M36SMU

Monday 1^{st} April, 2019

In your last assignment, you will implement an agent capable of playing a simplified version of the *blackjack* game (sometimes called *21-game*). The complete rules are in detail explained on Wikipedia [1]. However, in our project, we will restrict ourselves only to a simplified version.

The game is played with a standard deck of 52 cards, which is shuffled. Your goal is to score more than the dealer; however, you do not want to get over 21. In the beginning, you are given two cards and see one card that the dealer has. You can decide whether you draw one more card or stop playing. Once you stop playing, it is the dealer's turn. The dealer has to follow a fixed strategy — as long as the sum of his cards is less than 17, he has to draw a card. Dealer stops when this condition becomes false.

Face cards (Jack, Queen, and King) have value 10. Ace can be counted as 1 or 11.

At the end of the game, the player loses if the value of his cards exceeds 21. We call this situation *bust*. The player loses even if the dealer busts too. If the dealer busts and player not, the player wins. If neither the player nor the dealer busts, the winner is determined by the value of cards. The player with a higher sum of cards wins. Equal sums mean tie.

1 Implementation

We will use the *OpenAI Gym* [2] library as an environment for the game. You can find the environment implementation in file `blackjack.py`. File `carddeck.py` contains a model of card, card deck, and player hand. After each step, your agent will get an observation as an instance of `BlackjackObservation` class and a reward. In a terminal state, you get reward 1 for winning, -1 for losing and 0 for a tie. In any other state, you get zero as a reward. You are not allowed to modify files `blackjack.py` and `carddeck.py`. The same holds for file `main.py` above the comment stating that you cannot modify the code.

In file `randomagent.py` you may find a dummy agent that makes decisions completely at random. File `dealeragent.py` contains an implementation of a fixed strategy identical to the strategy of the dealer. You are encouraged to check those two files and reuse the code as you want. File `tdagent.py` should contain your implementation of passive reinforcement learning agent that learns utility estimates using temporal difference. File `sarsaagent.py` should contain your implementation of SARSA. In file `evaluation.py`, you may find some ideas on how to compare various agents. You may modify the code as you want to; however, it is not a requirement. In file `tdagent.py`, implement method `get_u_value`; in file `sarsaagent.py`, implement method `get_q_value`. Those will be used for testing your code.

2 Problem Specification

1. (3 points, mandatory)
   Propose three possible nontrivial reasonable[1] ways how to define the state in the game.

[1] For example you cannot expect points for a representation with two states - sum of values of cards is ≤ 21 and > 21.

2. (1.5 points)
For each state space representation from 1, provide a rough estimate of the overall number of states.

   You cannot just guess a number; you have to justify it somehow (e.g., by calculation). You do not need to provide an exact number; however, your estimate should not be too far from the true count. If you are not sure how to calculate the number of states, you can write a program that counts them for you and submit the code together with your report.

3. (2.5 points, mandatory)
Pick one of the state space representations you proposed in 1. Explain why you consider it the best one and answer the following questions. Does this representation capture all information that can be used for agent decision? Or is there any simplification? If yes, will the simplification influence the result (final policy, utility values)? If yes, how much will the result be influenced? Can you use exact methods (value iteration/policy iteration) to solve the game? If yes, how? If not, why?

   In one or two sentences explain your choice of discount factor and the number of games that you need to learn the Q-values.

4. (1 point, file `tdagent.py`)
Modify your implementation of a passive reinforcement learning agent that learns utility estimates using the temporal difference method. Take your implementation from the lab and make it work in the blackjack environment. Use policy that is identical to the dealer's policy[2] and estimate the value of each state.

   You should have a working implementation of the agent after the following tutorials. Because you will be working on the implementation in the lab, there is some cooperation allowed. Therefore the scoring for this point is low, and you will get points mostly for being able to use implementation you already have. **You are not allowed to cooperate when you are modifying your implementation to work with the blackjack environment.**

   If you are not sure what you should implement, you may want to read chapter 21.2.3 in AIMA book [3] or chapter 6.1 in book [4].

5. (6 points, mandatory, file `sarsaagent.py`)
Implement SARSA algorithm.

   SARSA implementation must be your own work. This means for example that if you cooperated in the lab on the implementation of passive reinforcement learning agent, you have to write the code again by yourself.

   If you are not sure what you should implement, you may want to read chapter 21.3.2 in AIMA book [3] or chapter 6.4 in book [4].

6. (6 points)
Test your code and provide an experimental evaluation. Compare the random strategy (provided), the dealer strategy (provided), the result from 4 and the strategy learned by SARSA.

   In this question, you should show that you believe that your implementation gives valid results, learns, and is well tested.

   For example, you may answer the following questions. What is the agent's expected or average utility? How fast do algorithms implemented in 4 and 5 learn? Does the learned

[2] Draw a card if and only if sum of your cards is less than 17.

utility contradict your intuition? What is the utility for drawing a card when you have club nine, diamond jack and spades two in your hand and dealer has club four? What is the utility of situation when you have diamond ace and spades five and dealer spades ace? Is it better to draw a card in this situation or not? Did your utility values/Q values converge? Does the strategy learned by SARSA follow the recommendation on the bottom of [1]?

3 Submission and Evaluation

- All students must work individually. Cooperation on anything else than the lab part of task 4 is strictly forbidden.
- Upload the results to `https://cw.felk.cvut.cz/brute/`.
- Strict deadline is Wednesday 8$^{\text{th}}$ May, 2019 11:59 pm.
- A penalty for late submission is -3 points for each day of delay.
- Submit all source code and a pdf report with answers to questions 1, 2 3 and 6.
- The solution must be compatible with Python version ≥ 3.6 [tested versions are 3.7.3 on Windows and 3.6.0 on Ubuntu].
- The project is worth 20 points in total. You are required to do all parts marked as mandatory to successfully finish the project and to get an assessment.
- Be sure that your state representation follows the assumptions from the lecture. If it only seems to work, it is not enough. Also, make sure that all assumptions needed for convergence are true and the parameter settings are reasonable. Failing this will cost you points unless you explicitly state that you violated the assumptions and explain why you can do that.
- Make sure that your implementation *learns* the Q-values. Do not include any pre-computed or hard-coded values or conditions in your code. In your evaluation, I may slightly change the rules of the game to find whether the algorithm works properly.
- Be honest with your answers in the report.
- Should you have any questions or you found a bug in code or project specification, feel free to email me and/or ask for a consultation.

References

[1] `https://en.wikipedia.org/wiki/Blackjack`

[2] `https://gym.openai.com/`

[3] Russell, Stuart and Norvig, Peter. *Artificial Intelligence. "A modern approach."* Prentice-Hall, Egnlewood Cliffs 25 (1995): 27.
`http://books.google.com/books?id=8jZBksh-bUMC`

[4] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction.* Vol. 1. No. 1. Cambridge: MIT press, 1998.
`https://mitpress.mit.edu/books/reinforcement-learning`