4 Intelligent Systems Lab Assignment — Reinforcement Learning

In this lab you will implement Q-learning, the most popular of all reinforcement learning algorithms. The environment for the agent will be the casino game BlackJack, or at least a simplified version of it.

Today's BlackJack game works as follows:

- The goal is to get the total score of the player's hand cards closer to 21 than the dealer total score without going over. Scoring higher than the dealer without going over 21, or staying below 21 while the dealer goes over, gives a reward of 1; scoring equal to the dealer not going over 21 gives a reward of 0; scoring less than the dealer, or going over 21 results in a reward of -1.
- To reach this score, the player is handed a number of cards and can decide after receiving each card to "HIT", which gives the player an additional card, or "STAND" which ends the game. In the real world version, a player can choose to "HIT" or "STICK" from when they hold 2 cards. Since the decision only becomes relevant when the total score reaches 12, our version of the game will only give control to the player when their total card value reaches 12 or more.
- Today's version of the game is played with a regular 52 card deck. Numbered cards (2 through 10) count for their face value. Picture cards (Jack, Queen, King) have value 10. An Ace can be used for value 11 or 1, depending on which value brings the hand total closest to 21 without going over.[1]
- The strategy of the dealer is simple: if the player goes over 21, the player loses and the dealer does not play, if the player stopped with a value below or equal to 21, the dealer will ask for cards until their hand value is 17 or higher and then stop.

Source code is available through the student portal that provides a java implementation of the BlackJack environment (including a basic visualisation of the game state), a random agent illustrating the use of the environment and its supplied helper methods as well as some non-compilable starter code version of the Q-learning agent.

Your Q-learning agent can look at the game's state at different levels of complexity. The simplest version of an agent might only look at its current total hand value. A smarter agent could also take into account whether it is holding an "active Ace", as this might allow it to play more daringly in some situations. Further information supplied to the agent includes the first card held by the dealer, and the specific cards the player is holding. It might be useful to also take this information into account to further increase the agent's probability for winning. No matter what information your agent is using, a table-based representation of the Q-values will work fine.

Implement at least one version of a table-based Q-tearning agent, but you will learn more by implementing the more complex versions. Tuning parameters used could also have an effect on learning and playing performance. Feel free to play around and try.

One thing that should't happen is learning Q-values > 1.0.[2] If you are getting even some Q-values > 1.0, you should take a very close look at and think about the consequences of slide 30 and the remark about terminal states for your code.

Handing in

When you are done, save your agent(s) and upload it to the student portal. Include at least one run-output of the algorithm which includes the progression of average reported rewards and a printout of your learned Q-values.

[1] Holding an Ace that counts for 11 is also reference to as "Holding an Active Ace" in this assignment and the code supplied.

[2] Actually, on average, Q-values will represent the fact that the house always wins (in the long run) and so the average Q-value shouldn't even be positive. Of course some situations have a high win probability and thus should return a value close to 1.0. Oh, and if you planned to give a remark about 1.0^2 and its value ... way ahead of you there. For all you know, I wrote this last year.