RL - Reinforcement Learning

Exercises on Policy Improvement

Prof. Riccardo Berta

2025.10.03

1 Blackjack

The game starts with the dealer having one face up and one face down card, while the player has two face up cards. All cards are drawn from an infinite deck (i.e. with replacement). The card values are:

- Face cards (Jack, Queen, King) have a point value of 10
- Aces can either count as 11 (called a "usable ace") or 1
- Numerical cards (2-9) have a value equal to their number

The player has the sum of cards held. The player can request additional cards (hit) until they decide to stop (stick) or exceed 21 (bust, immediate loss). After the player sticks, the dealer reveals their facedown card, and draws cards until their sum is 17 or greater. If the dealer goes bust, the player wins. If neither the player nor the dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21.

There is two possible actions:

- 0: Stick
- 1: Hit

The observation consists of a 3-tuple containing: the player's current sum, the value of the dealer's one showing card (1-10 where 1 is ace), and whether the player holds a usable ace (0 or 1).

The player receives positive +1 rewards for winning the game, negative -1 for loosing the game and 0 reward in case of drawn.

1 - Import the 'Blackjack-v1' environment from Gymnasium:

```
# You can get the environment from Gymnasium in the same way we got 'Frozen Lake'
# in order to visually plot the environment you can import it
# using render_mode="rgb_array"
```

2 - Create a random policy as a baseline:

```
# You have to create a function that get in input a state and provide a random
# action (in the range [0;1])
```

3 - Show the policy in action by rendering the environment several times after different decisions from the random policy:

```
# Reuse the show_policy function we implment for the lab on DP:
```

4 - Write a brute-force function in order to evaluate the probability of success and the average return obtained by a policy, the evaluate the random policy:

```
# You can reuse the "evaluate" function we implment for the lab on DP,
# however pay attention on how to calculte the success rate, you need to
# consider that the agent reaches the goal if it obtains a reward of 1 at
# the end of an episode.
```

5 - Notice that the state space is not a single integer, but a tuple of discrete states. In order to use the state as an index in the value function, you need to convert the tuple of states into a single value.

```
# To convert the tuple of discrete states
# (Discrete(32), Discrete(11), Discrete(2)) into a single index,
# you can treat the tuple as a coordinate in a 3D grid and
# compute a unique index for each combination of values.
# This is essentially flattening the multi-dimensional space
# into a one-dimensional array.
# Given a tuple of states (s1, s2, s3) where:
# - s1 is in the range 0 to 31 (32 possible values),
# - s2 is in the range 0 to 10 (11 possible values),
# - s3 is in the range 0 to 1 (2 possible values),
# You can calculate the index by assuming that s3 is the least
# significant and s1 is the most significant:

# index = s1 * (11 * 2) + s2 * 2 + s3
```

6 - Use Double Q-learning to calculate the optimal policy and the optimal state-value function:

```
# You can reuse the function we apply to the Slippery Walk environment, in particular
# select_action() to use an epsilon greedy exploration strategy, decay_schedule() to
# calculare learning rate and epsilon values and double_q_learning() to implement the
# algorithm. However, you need to use the state_to_index() function to convert the
# state into an index.
```

7 - Calculate the performance of the obtained optimal policy using the brute force approach:

```
# You can reuse the "evaluation" function written before,
# in order to evaluate the optimal policy
```

8 - Show the optimal policy:

```
# You can reuse the "show_policy" function written before,
# in order to show the optimal policy
```