Assignment 03: Reinforcement learning for Hunt the Wumpus

Group project by
Green team

Master in Computational Data Science
Artificial Intelligence: Methods and Applications

LIBERA UNIVERSITÁ DI BOLZANO
Bolzano, Alto Adige

Authors: Di Panfilo Marco, Lorefice Alessandra, Mugisha Denis, Pellè Gianluigi

Professor: Sergio Tessaris

Submission date: 04/01/2021

Academic year: 2020-2021

ABSTRACT

The task of the assignment was to implement an uninformed player to solve the "Hunt the Wumpus" game getting the best reward with the use of the q-learning technique. This technique attempts to learn the value of taking a specific action within a given state. A brief description of the game is as follows: An agent in a chess-like environment, where each cell can either be empty, being a pit, containing a wumpus or containing the gold. We would like to focus on the role of q-learning and how it affects the behaviour of the agent in the world. In this world, the agent can only move(up, down, left, right) in empty cells, in order to avoid the pits and the wumpus, to reach for the gold; and its movements are restricted by the orientation the agent has in a specific state.

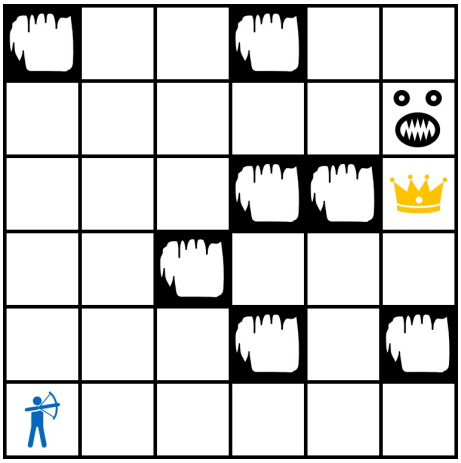
The objective of the game is to have an agent learn to navigate from start to the gold location, grab the gold, and climb out of the world i.e. attaining the best possible reward. It has to grab the gold if possible, with the ability to kill the wumpus, if necessary, in order to fulfil the goal. Following the objective of this assignment, the agent we have to implement is uninformed in nature. This agent, in a reinforcement learning problem only has access to the environment through its own actions, with no prior knowledge of the environment. To make things more interesting, we have built an intuitive visualization to provide a better sense of understanding of how this exploration strategy functions based upon the q-learning technique.

TABLE OF CONTENTS

Chapter 1

PROBLEM DESCRIPTION

Let's have a look at the problem:



We are a hunter looking for treasures in a **grid-map world** environment. The goal of the agent is to try to grab the gold and then exit the world (from where we came from) with the **cheapest sequence of actions**.

The agent (aka the hunter) has an orientation, and he can only move straight in the direction he is oriented. Otherwise, he first has to rotate (either LEFT or RIGHT) to move to a different tile. The possible orientations an agent can have are North, East, South or West, so he can only **move on an orthogonal position**. There may be several pits in the world. If the agent ever enters such a tile, he will die immediately and the game will be over. The agent also has an **arrow** (just one) that he can use to kill the wumpus; if you don't kill the wumpus and you enter the tile containing it the agent will die and the game will terminate. The available actions for the agent are:

- LEFT
- RIGHT
- MOVE
- SHOOT

- GRAB
- CLIMB

They all cost 1, except shooting which costs 10 if you have an arrow (shooting without an arrow is still possible but will have no effect). The same goes for the GRAB and CLIMB action, you can grab wherever you are, but if the agent is not in the gold location or the exit location respectively, then the action will have no effect. If you grab the gold, you'll get a 1000 points reward when you climb out of the game, and if you die, you'll get a -1000 points reward.

The world is not fully observable upon request, so, to infer information about it, we have to explore it. Information is gathered by some sensors the agent has, and they are perceived at the end of each action performed by the agent.
The agent perceives a BREEZE, if there are one or more pits in the 4 orthogonally adjacent locations (but doesn't know where exactly that pit is, and how many of them there are). It perceives a STENCH, if the wumpus is in one of the orthogonal adjacent locations, and it perceives a BUMP if it bumps on the border of the world. Finally, the agent perceives GLITTER, which means there is gold in that location.

The aim of the agent is to get the best possible reward from running a sequence of actions. We used the reinforcement learning technique to solve the game, implementing a **q-learning algorithm** which trains himself running the game a lot of times with the same environment. During the training, it builds a q-table with all the expected rewards the agent will get from running a specific action in a specific state. Once the table is made, we use it to run the game, executing at each step the action leading to the best reward.

Chapter 2

MODEL DESCRIPTION

2.1 SmartCoordinate and SmartVector

Since it's movements are restricted to the orientation of the agent, we decided to model the world as a **vector space** introducing the *linear_space* module where we defined the *SmartCoordinate* class, to represent locations in the world, and the *SmartVector* class to represent the orientation of the agent.

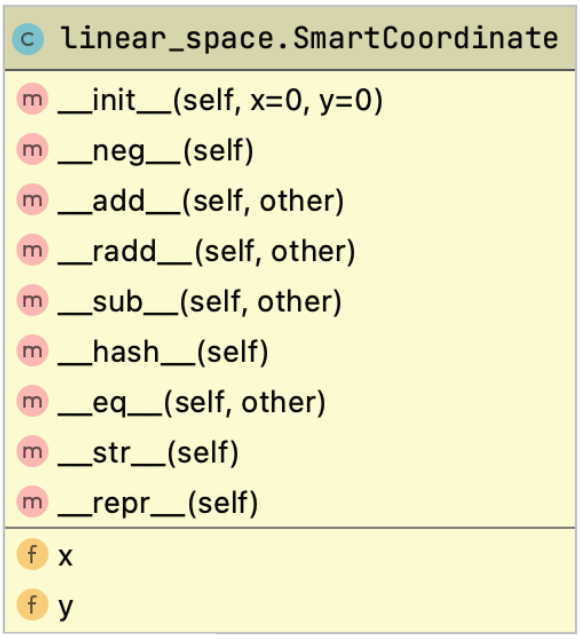
We implemented the operator (*SmartCoordinate* + *SmartVector*) so that you can sum a location with an orientation and get a *SmartCoordinate* representing the location of the tile you're moving into following that orientation.
We also defined some other helper methods. A noteworthy one is the
get_perpendicular_vector_clockwise() which returns the perpendicular vector (in clockwise order) to the vector calling the method, which is useful for some future calculations.
The definition of both classes is shown in Figure 2.1.

2.2 HuntWumpusSafeQLearningAgent

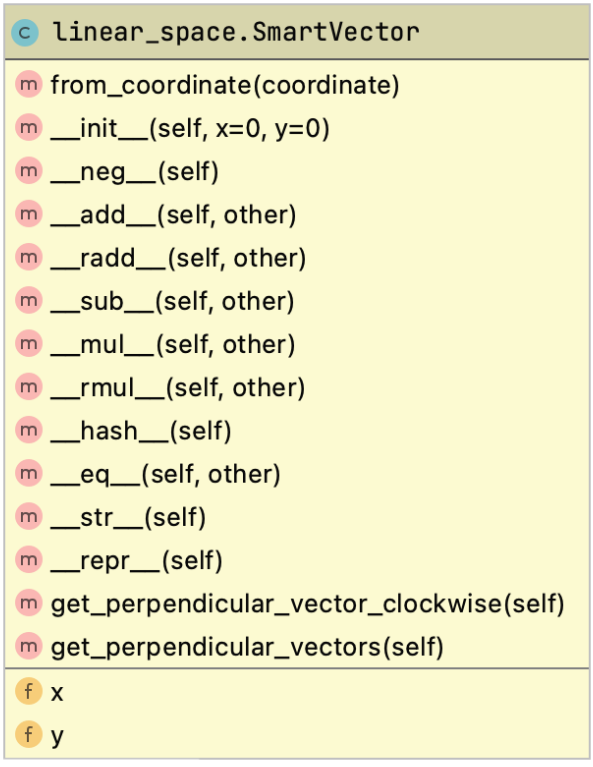
We started simple focusing on solving a world environment where there is a safe path with no obstacles (wumpus) in between to reach the gold. We made the class *HuntWumpusSafeQLearningAgent* to model an agent which train itself running the game over and over and build a q-table to keep track of his discoveries. The q-table is a dictionary data structure which maps [state, action] -> reward (int) and it stores the expected reward from executing a specific action in a specific state. It takes into consideration also the discounted reward that would result from executing future actions after the chosen one. Since it has to deal with an environment where the transition model may be not known or non-deterministic, it has to take into consideration all the experiences he had for a specific state, given that the results may be different from time to time. To handle all of this we used the following formula 2.1, in which alpha indicates the learning rate, and gamma the discounted factor:

$$Q[s,a] \leftarrow (1-\alpha)\, Q[s,a] + \alpha(r + \gamma\, max_{a'}\, Q[s',a']) \tag{2.1}$$

Another important aspect of the learning technique is the exploration strategy. We first had a try to the epsilon-greedy strategy, but it didn't make sense to keep the epsilon value constant overtime; a better approach is to make it dynamic as time goes by, so that the more experience the



(a) *SmartCoordinate class diagram*



(b) *SmartVector class diagram*

Figure 2.1: Linear Space classes

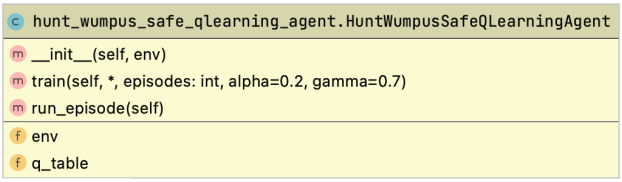


Figure 2.2: HuntWumpusSafeQLearningAgent class diagram

agent had the less it tries to explore new possibilities. So, we implemented a simple exponential decay algorithm which makes our epsilon value start from 1.0 and going all the way down to 0.1 at the end of the training in an exponential way. After some tests, we found out that this solution was not well-suited for trainings with a small number of episodes (we only trained for at most 150'000 episodes in order not to wait too long for the execution process) since it modeled an exponential-curve decrease. Eventually, we decided to follow a simple approach: for the first 3/4 of the training we explore 70% of the time and exploit 30% of the time, for the last 1/4 of the training we explore 15% of the time and we exploit 85% of the time.

Yet another important aspect we had to face with was the reward to give to the agent. We first tried only using the final reward received by the agent when it climbs out of the game, but we were too much optimistic: the agent could very rarely find the solution of the game during the training so it needed a very huge training to learn the solution path. So, we decided to make an intermediate goal: grabbing the gold. When the agent grabs the gold we infer a 1000 points reward. Even with this intermediate reward, the training converges very slowly and only for big enough training samples, because until it is able to find a way to grab the gold it would always prefer to climb out immediately. Because of that, we had to make a trade-off between execution time and the probability of solving the game: we ended up with a 150'000 episodes training which takes about 1 minute and a half execution time and it finds a solution most of the times in a 6x6 world environment (about 80% of the times). To be sure that it always finds a solution the training should be very high, and it must be chosen based on the world size.

Once the table is made, the agent has a ***run_episode()*** method which plays the given game executing at each time the action leading to the best reward for the state the agent is currently in. The definition of the class is shown in Figure 2.2.

2.3 HuntWumpusCustomQLearningAgent

It is very similar to the *HuntWumpusSafeQLearningAgent* but it works in every kind of environment. The definition of the class is shown in Figure 2.3.

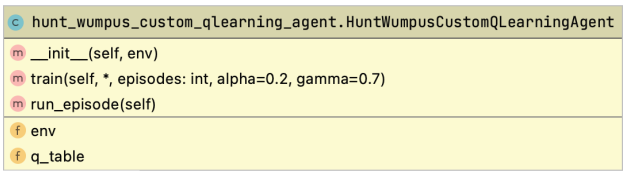


Figure 2.3: HuntWumpusCustomQLearningAgent class diagram

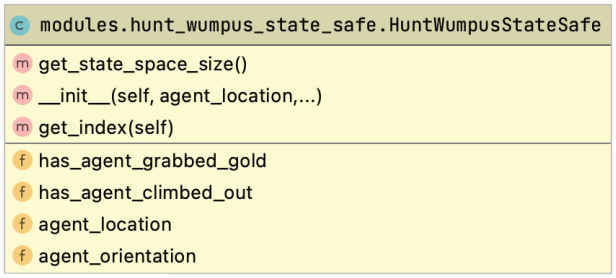


Figure 2.4: HuntWumpusStateSafe class diagram

2.4 HuntWumpusStateSafe

This class represents the minimum information needed by the learning algorithm to differentiate game states. It only needs to keep track of the agent location, agent orientation, whether or not the agent has grabbed the gold and whether or not the agent has climbed out, resulting in a q-table with a maximum of 6144 elements (1024 different states and 6 different actions). The class has also a method ***get_index()*** which maps the object into an integer that can be used to index the q-table. This class is used by the *HuntWumpusSafeQLearningAgent* class to distinguish different game states. The definition of the class is shown in Figure 2.4.

2.5 HuntWumpusStateCustom

This class is very similar to the *HuntWumpusStateSafe* class but it works for any kind of world environment. Even the ones where killing the wumpus is the only way to reach the gold safely. In order to handle that, however, it has to keep track of more information: whether the arrow is available or not, whether the wumpus is alive or not, and whether the agent perceives a scream on a particular state. Adding this information, it has to keep track of a total of 8192 different states, resulting in a q-table of 49'152 elements size. This class is used by the *HuntWumpusCustomQLearningAgent* class to distinguish different game states. The definition of the class is shown in Figure 2.5.

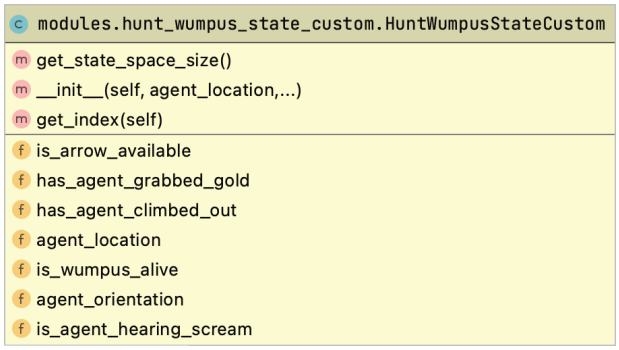


Figure 2.5: HuntWumpusStateCustom class diagram

Chapter 3

HOW TO RUN THE CODE

To run the code you have to create an anaconda environment with the configuration file environment.yml and then activate it to run the code. The required commands to make it work are the following:

1. conda create env -f environment.yml
2. conda activate aima_gym
3. jupyter lab

To run the code regarding the assignment you just need to run the code cells in the file `wumpus_safe_sample.ipynb`. The first block contains all the imports of the libraries and the import of our *HuntWumpusSafeQLearningAgent* class. The next code block creates the environment. After that the agent is initialised and the training starts by calling the ***HuntWumpusSafeQLearningAgent.train()*** method. The fourth block shows the training plot with the training curve. In the last block the agent runs the episode and a graphical representation of the actions of the agent is printed.

The notebook `wumpus_blocking_sample.ipynb` contains a sample world where the only way to find a safe path leading to the gold is by killing the wumpus.

The notebook `wumpus_no_way_sample.ipybn` contains a sample world where there is no way to reach the gold without dying, so the best move is to climb out immediately.

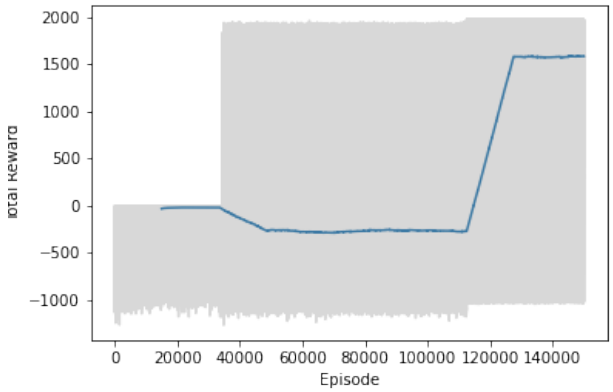
When changes are performed in imported objects or in the notebook itself we recommend to restart the kernel for each run in order to avoid caching of older versions.

Chapter 4

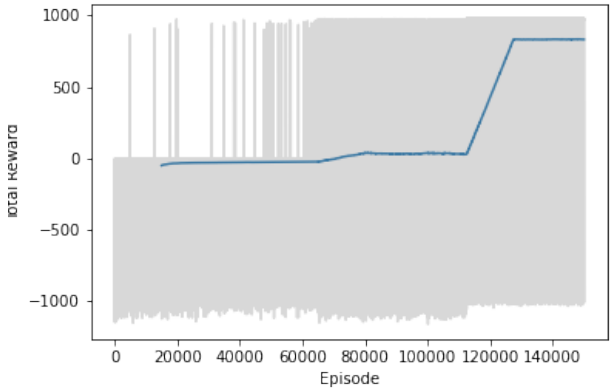
CONCLUSIONS

The main goal of the assignment was to solve the game getting the best possible reward from playing the given game. In the very beginning, we got very weird behaviour from the agent, it kept rotating forever without actually doing anything. It happened because we tried to model the state keeping track of the agent location only without its orientation, and that was not enough to define two distinct states. But in the end, we managed to make it working properly on any world environment, even the ones where the wumpus needs to be killed.
We made a run test for the three main scenarios of the game, which are listed below:

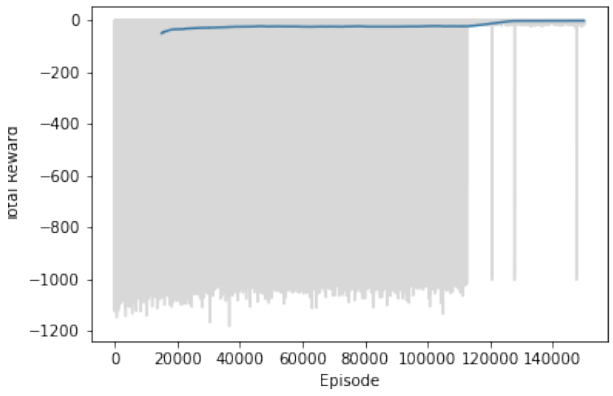
1. There is a safe way leading to the gold. The training plot for this scenario is shown below.



2. There is a safe way leading to the gold but the wumpus need to be killed. The training plot for this scenario is shown below.



3. There is no way leading to the gold. The training plot for this scenario is shown below.

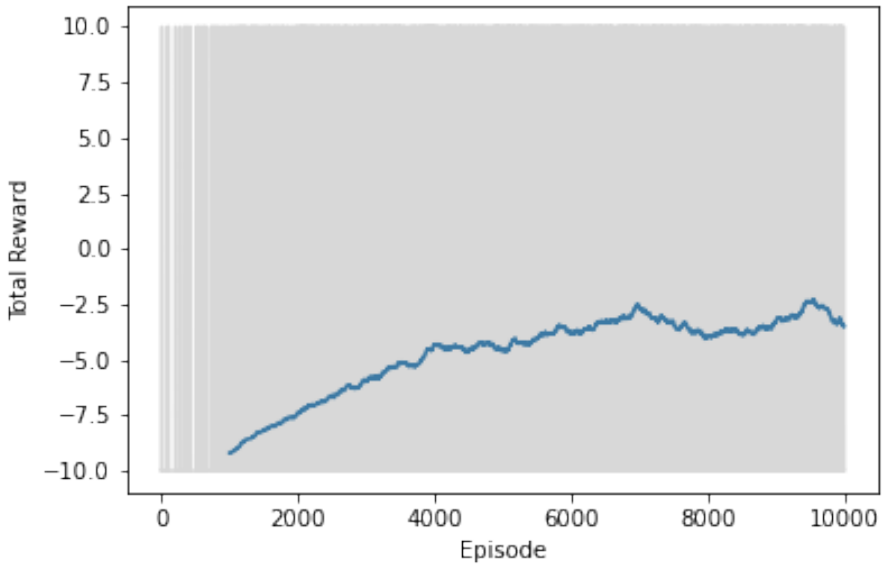


Chapter 5

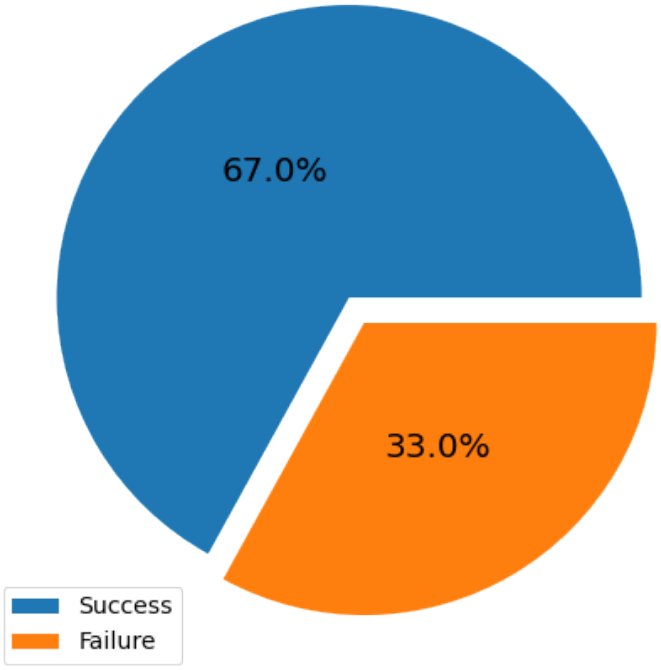
APPENDIX

To dive into the reinforcement learning techniques, the assignment first required to solve the Frozen Lake game from the openAI gym toolkit[1]. We both had to solve the FrozenLake-v0 and FrozenLake8x8-v0 environments. The game is really simple, there is an agent moving in a chess-like grid world, and it needs to reach a specific goal location, paying attention not to fall into holes. It can move North, East, West or South, but since the surface of the lake is slippery, it is not guarateed that it will end up in the expected location he wanted to move to. We developed the solution of it in a very similar way to the one we used for the wumpus game, but since the transition model is non-deterministic, it is not guaranteed that it will solve the game. You can see the training and testing results we got from the frozen lake exercise in Figure 5.1 (a) and (b).

[1] openAI gym: https://gym.openai.com



(a) *Training diagram*



(b) *Testing diagram*

Figure 5.1: Frozen Lake