Reinforcement Learning Project

Ioannis Ioannidis - Dimitris Patiniotis
University of Piraeus
NCSR Demokritos

Abstract: In the present report we describe how we dealt with the examination task in the reinforcement learning section of the machine learning course. Firstly, in the introduction we represent the problem we are dealing with, while in the theoretical background section we give an intuitive taste of the algorithm that will be used. The steps followed towards the solution, as well as some plots showing the final results we came up with are provided in the experiments chapter. Finally, all the remarks highlighted during the solution procedure and the observations we ended up are quoted in the conclusions part.

1. Problem Introduction

Consider 15 agents, where :

- 2 of them keep a weight of 0.6
- 5 of them keep a weight of 0.4
- 8 of them keep a weight of 0.2

All agents show up together at the beginning of the game's scenario. During each round they have to select a cell on a 2×2 grid described as following

	1	2
1	2.5	2.5
2	2.5	2.5

The value at each cell stands for its capacity. In each round all players make their selection at the same time and they teleport to the cell chosen. The sum of the weights of the agents that led at a cell constitutes the cell's call. After selection and teleport is made, all calls are measured. Each player gets a reward corresponding to the difference between *capacity - call* for the cell that he chose. If all calls are lower that the respective capacities, round ends, players receive their payoffs, return to the initial status and we head for another iteration. On the other hand, if there exists a cell where call is greater than the available capacity, each agent that belongs to this cell gets a negative reward (obviously based on the type that describes the reward mentioned above) and he is obliged to make another selection, until he ends up at a cell he can fit in. At that point, the round ends.
Our goal is using a Q-learning, ε-greedy algorithm to train this multiagent system, in order to maximize each player's performance.

2. Theoretical Background

Q-learning is a reinforcement learning technique used for finding the optimal policy in a Markov Decision Process. Optimal policy stands for achieving that the expected value of the total reward over all successive steps is the maximum possible. So, in other words, the goal of Q-learning is to find the best actions by learning the optimal *q-values* for each state-action pair.

For that purpose, we'll be making use of a table, called a *Q-table*, to store the q-values for each state-action pair. The horizontal axis of the table represents the actions, and the vertical axis represents the states. So, the dimensions of the table are the number of actions by the number of states.

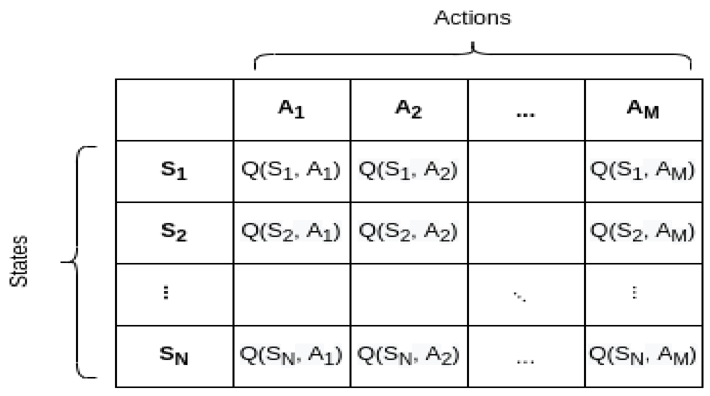
$Q(S,A)$ in our Q-table corresponds to the state-action pair for state S and action A. R stands for the reward, while t denotes the current time step, hence $t+1$ denotes the next one. Finally, alpha (α) is the learning rate and gamma (γ) is the discount factor.

All possible values of state-action pairs are calculated iteratively by the formula:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},A_t) - Q(S_t,A_t)]$$

This is called the action-value function or Q-function and it approximates the value of selecting a certain action in a certain state.

Calculating and updating q-values, the output of the algorithm is a Q-table. Such a table for N states and M actions looks like this


Actions
States

	A_1	A_2	...	A_M
S_1	$Q(S_1, A_1)$	$Q(S_1, A_2)$		$Q(S_1, A_M)$
S_2	$Q(S_2, A_1)$	$Q(S_2, A_2)$		$Q(S_2, A_M)$
$\vdots$			$\ddots$	$\vdots$
S_N	$Q(S_N, A_1)$	$Q(S_N, A_2)$	...	$Q(S_N, A_M)$

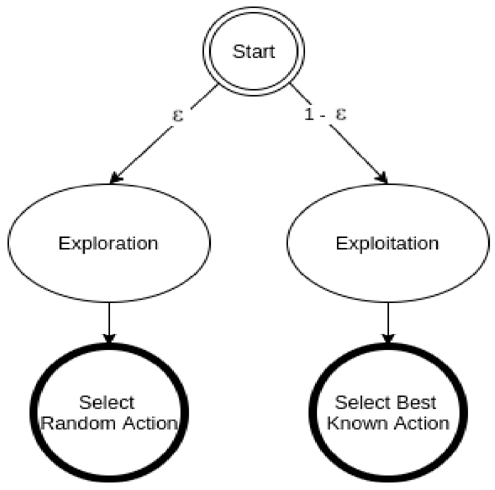
The pseudo-code for updating the table as was described above is

```
Algorithm 1: Epsilon-Greedy Q-Learning Algorithm
Data: $\alpha$: learning rate, $\gamma$: discount factor, $\epsilon$: a small number
Result: A Q-table containing Q(S,A) pairs defining estimated
        optimal policy $\pi$*
/* Initialization                                          */
Initialize Q(s,a) arbitrarily, except Q(terminal,.);
Q(terminal,.) $\leftarrow$ 0;
/* For each step in each episode, we calculate the
   Q-value and update the Q-table                          */
for each episode do
    /* Initialize state S, usually by resetting the
       environment                                         */
    Initialize state S;
    for each step in episode do
        do
            /* Choose action A from S using epsilon-greedy
               policy derived from Q                       */
            A $\leftarrow$ SELECT-ACTION(Q, S, $\epsilon$);
            Take action A, then observe reward R and next state S';
            Q(S, A) $\leftarrow$ Q(S, A) + $\alpha$ [ R + $\gamma \max_a$ Q(S', a) - Q(S, A)];
            S $\leftarrow$ S';
        while S is not terminal;
    end
end
```

What we haven't cleared up yet is the way that an agent selects an action. To answer this question, we'll introduce the trade-off between exploration and exploitation.

- *Exploration* is the act of exploring the environment to find out information about it.
- *Exploitation* is the act of exploiting the information that is already known about the environment in order to maximize the return

In ε-greedy action selection, the agent uses both exploitation to take advantage of prior knowledge and exploration to look for new options, depending on fixed point given.



Specifically, with a probability of ε, we choose to set aside what we have learned so far and choose an action randomly, independent of the action-value estimates. Exploration allows us to have some room for trying new things, sometimes contradicting what we have already learned.
The pseudo-code for selecting an action at each round as described above is

```
Algorithm 2: Epsilon-Greedy Action Selection
  Data: Q: Q-table generated so far, : a small number, S: current
        state
  Result: Selected action
  Function SELECT-ACTION(Q, S, $\epsilon$) is
      n ← uniform random number between 0 and 1;
      if $n < \epsilon$ then
          A ← random action from the action space;
      else
          A ← maxQ(S,.);
      end
      return selected action A;
  end
```

3. Experiments

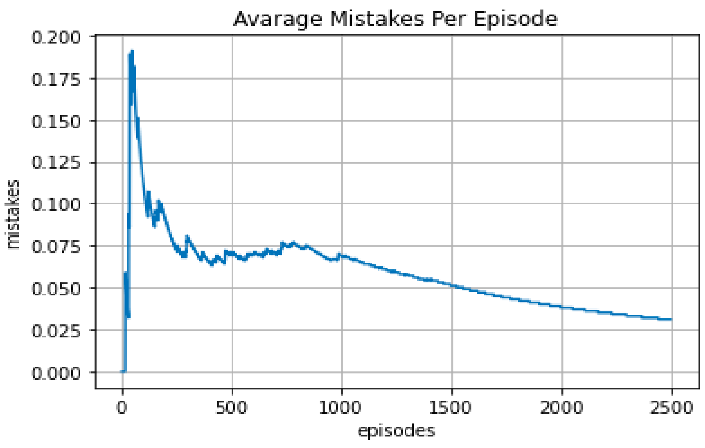
3.1. Experimental Setup

Using a Q-learning ε-greedy algorithm, as described above, we tried to maximize each player's reward in the problem we presented. For that purpose, we modeled the game's description, as well as the environment and the competitors, in Python scripts.

For the learning procedure we ran 2500 episodes, initializing ε at 1 and γ at 0.9. For the first 2000 runs, our goal was to gradually decrease ε, in order to keep great balance between the exploration and exploitation actions. In that way, our agents would have had enough space for both learning new policies and using their -already- formed beliefs in a satisfactory ratio. Finally, we kept last 500 episodes, where ε reached the minimum value of 0, only for exploitation.

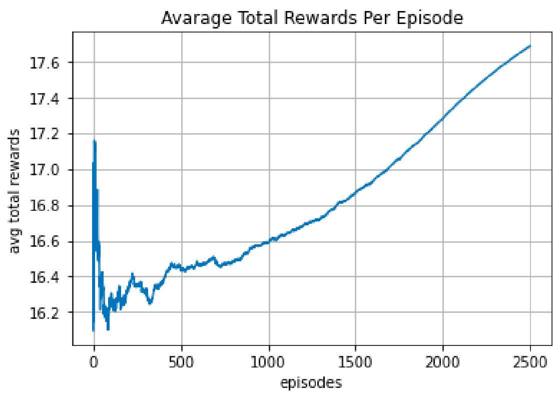
3.2. Results

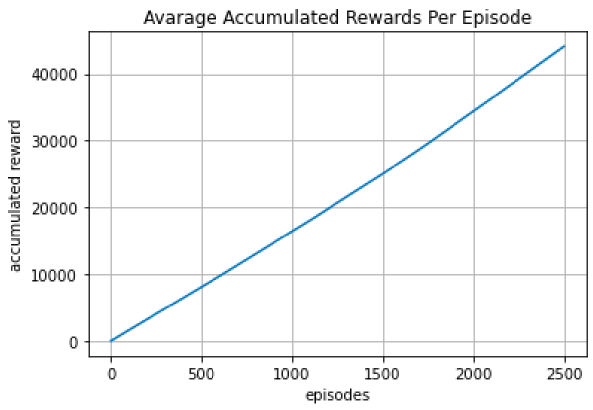
Considering that we' ve been looking for the optimal solution, our goal was to minimize the mistakes of each round. As a mistake we counted the action of an agent ending up at an overweighted cell, where he gets a negative reward and he is obliged to choose another one. The following plot imprints the value of the average mistakes made during all rounds over number of episodes runned.



As we can see, those values converge to zero, as while ε decreases (and finally reaches and remain at 0), agents choose their best actions and no mistakes at all are made.

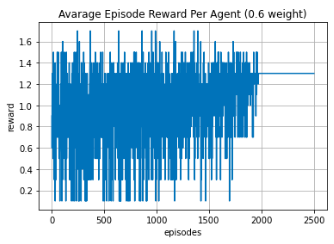
Obviously, as the mistakes are gradually eliminated, agents only receive positive payoffs and the value of total and accmulated rewards received keep increasing. We can visualize this observation by plotting those values over the number of episodes runned.

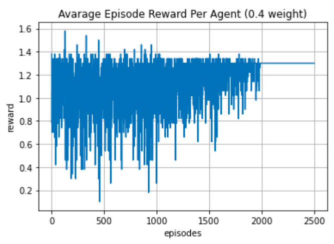


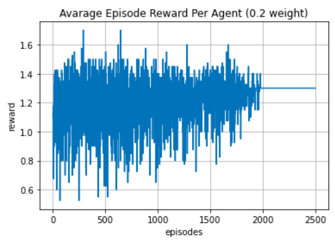


From the point, where the agents have learned and choose their best actions, it is crystal clear that that rewards keep improving noticeably.

Reamining on the rewards examination, we tested if there were any dependences between the receives of an agent and its weight type. For that purpose we created plots displaying the average payoffs per weight class over time.

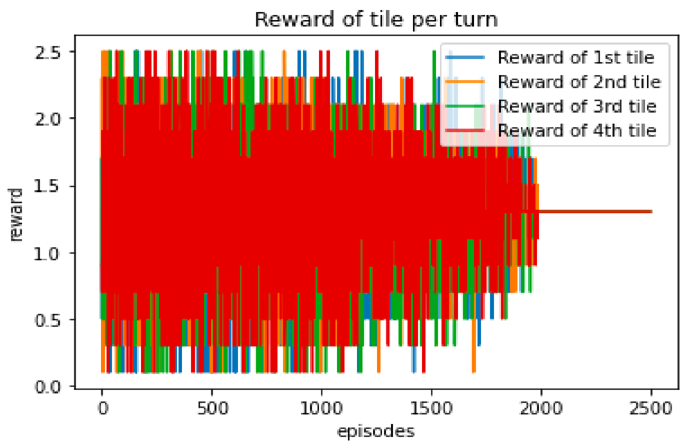






We conclude that weight does not affect the final reward, as all agents end up having the same income regardless of their type. This is quite sensible, as all competitors learn to distribute equally in the cells. In that way, they all get the maximum possible plus equivalent receives.

Finally, we examined the reward that each state provides over the game's progress. As expected from the previous plots, while in the beginning payoffs are observed to distribute randomly with big variance, as the game evolves rewards converge and all state return the exact same value.



4. Conclusions

Epsilon greedy is a very useful algorithm for reinforcement learning. Selecting the suitable parameters ε and γ we can expect great performance, as the method always converges to optimal solutions. In addition, it is a technique based on simple fundamentals, easy for someone to undestrand and implement. Finally, it constitutes a great choice over the different Q-learning models, as it can be applied to various MDPs, such as the present project's problem and many more situations.