iscte TECNOLOGIAS E ARQUITETURA

Introduction to Machine Learning

Reinforcement Learning Exercise

These exercises can be solved in any programming language of your choice. It is assumed you are proficient with programming. It will be helpful if the language has a library to plot graphics.

Questions: https://join.slack.com/t/introductiont-qjc9426/signup (only @iscte-iul.pt users allowed) - Remember that exercises are for evaluation so don't hint about your solutions on public channels.

Imagine a situation where a robot needs to **learn the sequence of actions** that takes it from an initial position to the energy plug (marked with X). Imagine that it can experiment learning a simulated (simplified) environment. Something like the scenario depicted in Fig. 1.

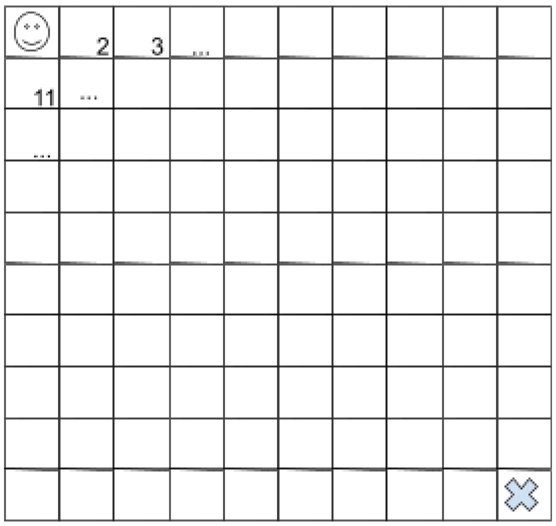

Fig. 1 The robot/agent is in the top left corner and the plug/goal is in the bottom right corner. Some state numbers are depicted as example.

Suppose (by oversimplification) that the room can be divided in squares and these "areas" (that we will call states) are numbered so that the robot can identify each different position in the room. Imagine also that the robot has 4 actions (up / down / left / right) and these go from the middle of one square to the middle of an adjoining square. From this robot's point of view it knows it is in state 1 and if it moves left it will receive information that it arrived at state 2, if it moves down it will be informed of arriving at state 11, if it tries to move in another direction it will be informed that it remained in state 1.

Upon arrival at state 100 it will receive a reward.

1. Build the state-transition function (*s' = f(s, a)*), where a state *(s)* and an action *(a)* are given as argument, and a new state (the arrival state, *s'*) is returned, so that: f(1, left) = 2, f(1, down) = 11, f(1, up) = 1, etc.

2. Create a reward function *r(s)* that will reward all states with 0 and state 100 with 100.

3. Program a search function that will randomly choose an action. When the robot reaches the goal it should be returned to the initial position after getting the reward. Execute this function for 1000 steps and repeat 30 times. Measure the average reward per step in these 1000 steps. Measure also the runtime of doing 20000 steps. Calculate average and standard-deviation of run-times. This will be the baseline result and it will be used to test if the system is doing better than just random guessing in the future.

Tip on results presentation for all exercises: In all situations that have any stochastic process (randomness or pseudo-randomness) involved, one needs to 1) store the random seed to repeat the same exact experiment if necessary, 2) repeat the experiment 30 times with the same parameters, and calculate average and standard deviation of each measurement, for good comparison (box-plot with whiskers is a good representation in most of these situations).

4. Create a vector V to store the utility of each state (initialized to zero). When arriving at a state *s'* **update the utility of the state where the robot came from *(s)***, using:

$$V(s) = (1 - alfa)\, V(s) + alfa * (\, r(s) + discount * V(s')\,) \qquad (1)$$

where *alfa* is the learning rate (*alfa* close to one makes you forget past information and focus on what you learned recently, *alfa* close to zero makes learning secure, based on solid historical value, but slow). The *discount* factor that tells you how valuable is long-term reward. Values below 0.9 make the importance of long-term rewards decrease fast, values (very) close (but never equal) to 1.0 make the importance of a long term reward have more impact.

V(s) can be initialized to zero or to random positive numbers close to zero. Solve ties unevenly so that in case of a tie the chosen state is random between the tied states. Choose alfa and discount making a few runs with different values (the correct process should be running 30 times with each value set, and choosing the best parameter-set, but you can use a less demanding process in this exercíse).

Can you now create a program, that given a vector V built this way, and using **only this information** (using only the values in vector *V*, that will change from problem to problem), can find the sequence of states the robot must pass through to get to its goal with the least number of steps? Can you also say (only with that information) which is the sequence of actions that will get the robot to its goal? What other information is required to know the sequence of actions? Do you have any indication if that sequence of states is a good solution? The best solution? What if the numbers of the states were shuffled before training started and the state transition function was unknown, could you still use vector V to choose an action?

Execute this function for 20000 steps and repeat 30 times. At steps 0 and 20000 run the system for 1000 steps and measure the average reward per step in these 1000 steps. Measure also the runtime of each full test (all 20000 steps) and calculate average and standard-deviation of run-times. Compare the results with those in exercise 3.

Tip: A heatmap is a good way to visualize the information in vector V after a reshape to 10 x 10.

5. Create a matrix Q, indexed by the state index, and the action index Q(s, a) and initialize it with small, positive, random values. When arriving at a state *s'* **update the utility of the state where the robot came from *(s)***, using:

$$Q(s, a) = (1 - alfa)\, Q(s, a) + alfa * (r(s) + discount * (max_{a'} Q(s',a')))$$

where $max_{a'} Q(s',a')$ is the best *Q(s',a')* for all *a'*.

Can you now tell which is the best sequence of actions using **only the information** on *Q*? And the best action from any given state?

Maybe we can do better if we use the knowledge acquired during training to guide the training, what if we always choose the best path calculated so far? Does it converge faster? Is the solution better?

Execute this function for 20000 steps in each experiment and repeat the experiment 30 times. At steps 100, 200, 500, 600, 700, 800, 900, 1000, 2500, 5000, 7500, 10000, 12500, 15000, 17500, 20000 (or other intermediate points that are deemed useful) run the system for 1000 steps and measure the average reward per step in these 1000 steps. Measure also the runtime of each full test (all 20000 steps) and calculate average and standard-deviation of run-times. Plot the steps vs avg reward at the measured points or in others that best depict the evolution of learning.

Tip: If you need to see the full policy, to represent the Q table, the best process is to have a heatmap of the maximum quality for each state and / or a matrix with the best action for each state.

6. The solution of 5 may "*freeze*" too soon on a suboptimal solution. How can we avoid this? The answer is: explore! Include a term *(greed)* in the action selection function that will determine the probability of choosing a random action. For example if *greed* is 0.9, approximately 10% of the actions chosen should be random, the remaining 90% should be the best action available according to Q. If greed is low, for

example 0.2, approximately 80% of the actions are random. Try different greed parameters and compare the results. Finally try an increasing *greed* parameter starting around 30%, for the first 30% of the test steps, and slowly increasing until 100% by the end of the test. Compare results and the Q tables.

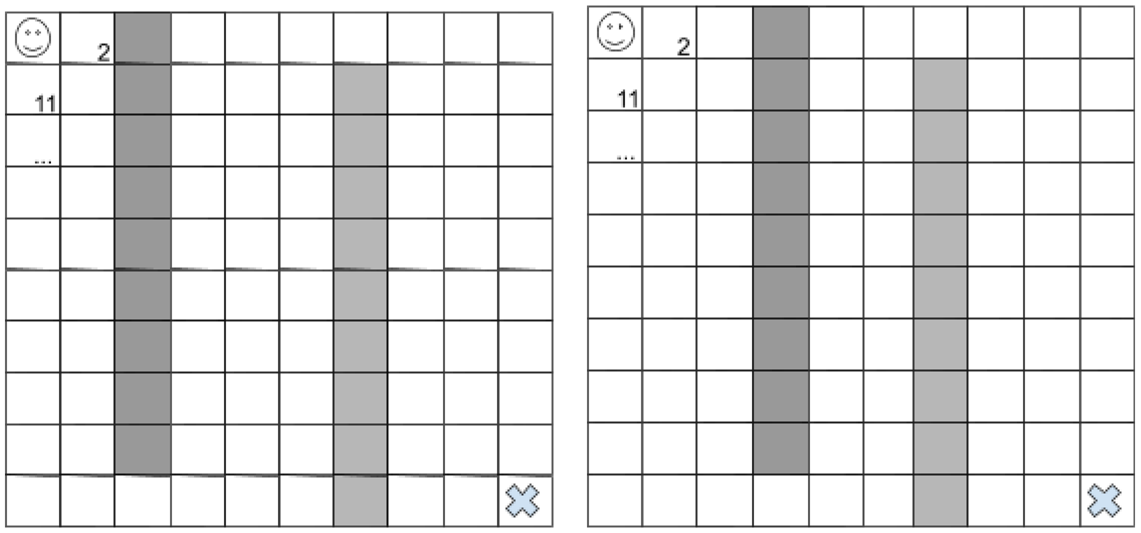

Fig. 2 Walls make the problem harder. Use penalty rewards (small compared to the final reward) to keep the agent in the right track.

7. Imagine that the same action does not always take the robot to the same state. With a 5% probability it can take the robot to any neighbouring state of the current state. How does that affect the result?

8. Change the simulation to include walls (as in Fig 2) and compare with results of the simple scenario. Try on both labyrinths, are they both solvable with the state representation you chose? If not, why?

9. Imagine now a situation, closer to the real scenario, where states are not numbered and the agent can only perceive its position by the echos on the walls. The agents' "perception" of what is around it is an array of floating point values that represent the distance to the wall for each side UP, LEFT, DOWN, RIGHT, e.g. NA, 0.56, NA, 0.14, means: no walls found UP, wall at 0.56 (meters?) to the LEFT, no walls DOWN, wall 14 cm to the RIGHT. How can these states be simplified into a number that can be used as an index? what are the risks of this transformation?