KEN 4154 Assignment 3

Reinforcement Learning

Adrian Sondermann, Abel de Wit

November 24, 2020

1 Introduction

In this assignment we were tasked to implement a type of reinforcement learning with the OpenAI testbed 'Mountain Car'. In this environment, the car is tasked to drive on a sinus function and its goal is to reach the top of the hill. For most people this task seems trivial; Go back and forth between the two hills to build up some momentum and then use that to reach the top of the hill. This however, is less intuitive for algorithms as driving up the left hill would mean removing the car further from the goal. Because of this, regular pathfinding algorithms will have difficulty with distancing themselves from the goal first, to then be able to reach the goal.

To solve this, the problem is reformulated to be used by reinforcement learning. Using reinforcement learning, the algorithm can learn what the reward will be in a situation if taken a certain action, these are called state-action pairs.

2 Implementation

2.1 Environment

Using the `gym` package provided by OpenAI the environment is set up rather easy. The environment provides us with several things:

2.1.1 Observation

The observation describes the current state of the environment. In the case of our 'Mountain Car' example we are provided with two values.

The first is the horizontal position of the car in the environment. This value ranges from -1.2 to 0.6. The car starts in a random position between -0.6 and -0.4, which is the valley in our environment.

The second value of the observation is the speed of the car. This value ranges from -0.07 to 0.07, where a negative value indicates moving to the left and a positive value indicates moving to the right.

2.1.2 Step function

The environment has a function called step. This function takes as a parameter the action that is to be taken, and outputs four values. The observation (state) after taking the action, the reward of this action, whether the objective is reached '`done`', and some information which is irrelevant for our scenario.

2.1.3 Discretising state space

As Q-learning requires a discrete state space, both variables, speed and position were partitioned into equal-width bins. The speed of the car was rounded to 2 decimal digits, its position to 1 decimal digits. Thus the state space consits of $15 \cdot 19 = 285$ states.

2.2 Q-learning

Using the environment described above, we were able to implement a very basic Q-learning algorithm that is able to teach itself what types of actions should be taken in which states. It teaches itself this trough repeated trial and error, these trials are called `episodes`. For each episode the environment is reset and a certain set of actions are taken by the algorithm.

The Q-learning algorithm uses epsilon greedy[1] (see Figure 1) to determine whether it will choose an action randomly or based on the learned Q-function. The epsilon is lowered while learning to slowly force the algorithm to use the Q-function more than random moves, but these random moves are what makes the algorithm go to unexpected states which increases the learning of the environment.

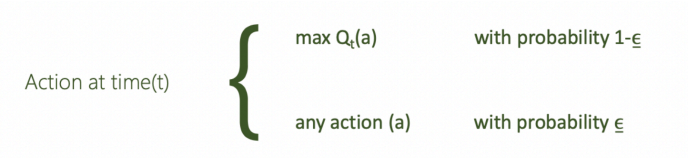

Figure 1: Visualization of the average rewards per episode

After it has chosen an action either randomly or based on the highest value in the Q function for the current state, the step function which is described above is called with the chosen action. The result of taking this action is recorded in the four variables that are returned. The new state is discretised and the Bellmann update is performed on the Q-function.

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma maxQ(s',a') - Q(s,a)]$$

The reward that is achieved by doing action 'a' in state 's' is given by the step function. The new state that we're in 's'' is also given by the step function. And

[1] `https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/`

parameter	value
learning rate	0.2
discount	0.9999
epsilon	0.01
minimal epsilon	0
episodes	5000

Table 1: Tuned parameters

with these values we can lookup the Q-values $Q(s, a)$ and $maxQ(s', a')$. With enough episodes of randomly moving around in the environment, the algorithm starts to reach the goal more and more often, because of this the algorithm will learn a policy of which action is the best to take in a certain state. In the case of our 'Mountain Car' environment, that means that the algorithm learns whether to move left or right given his position on the hills and his current speed.

3 Results

3.1 Visualisation

To visualize the state-value function Q the maximum per discretisised two-dimensional state (velocity, position) is calculated. Then the retreived 2d-matrix is displayed using the *viridis* colormap.

3.2 Tuning

A good visualization of the function must contain information about how the car will behave in different situations, respectively the distance to drive until the destination will be reached. Therefore the parameters in Table 1 were determined. The big discount rate is useful to gain insight of how far the car has to travel, beginning from its current position and velocity. A small epsilon guarantees few random moves, as these are only initially needed. This is because, the algorithm needs to learn driving away from the destination when climbing the hill slowly. Afterwards, random moves won't be needed. 5000 episodes are enough for the algorithm to converge, as shown by the convergence in Figure 2.

The state-value function in Figure 3 beautifully visualises the time left the car has to drive before reaching the destination. It starts in the dark middle of the plot without any speed. Initially it needs to gather speed by driving back and forth, until it is fast enough to climb the hill completely. This results in a snake-like shape of the final function. Hence, if the car is fast enough (upper and lower end of the plot) it takes very few timesteps to reach the goal.

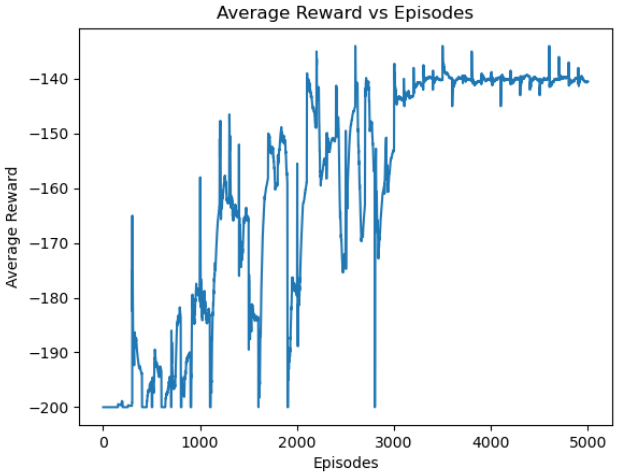

Figure 2: Visualization of the average rewards per episode

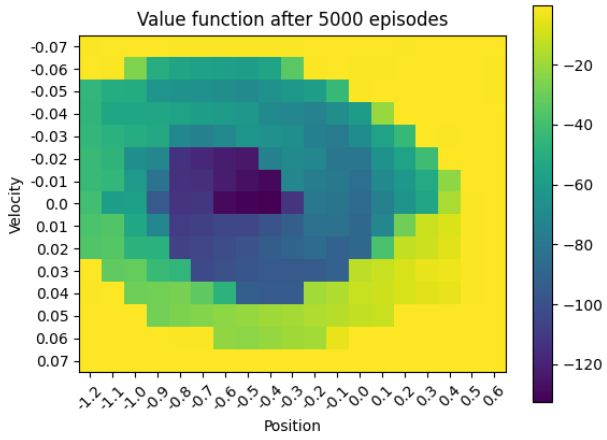

Figure 3: Visualization of the state-value function