Sapienza University of Rome

1052217

Artificial Intelligence

Resolving MDP domain problem using Q-learning algorithm

Author:
Horpynchenko, Dmytro

Student Number:
1807584

Contents

1 Reinforcement learning **2**
1.1 Overview . . . 2
1.2 Markov Decision Process . . . 2
1.3 Properties . . . 2
1.4 Q-learning algorithm . . . 3

2 Project domain **4**
2.1 Overview . . . 4
2.2 GYM library . . . 4
2.3 Domain specifications . . . 5

3 Implementation **6**
3.1 Project code structure . . . 6
3.2 Agents . . . 6
3.2.1 Random Agent . . . 6
3.2.2 Look-up table Agent . . . 6
3.2.3 Function approximation Agent using Multi Layer Perceptron . 6

4 Results and conclusions **8**

1 Reinforcement learning

1.1 Overview

Learning is a process to improve the performance of a system based on its past experiences. [1] This method occurs when the problem seems too complicated to solve in real time, that is, system mush collect some knowledge to produce a correct decision or behavior.

The reinforcement learning (RL) is a technique which is to acquire the agent executor behavior desired by methods based on the concept of reward or punishment. [1] For simplifying RL algorithms assumed that problem domain has Markov property, as such, it is Markov Decision Process.

1.2 Markov Decision Process

Markov decision process (MDP) is a discrete time stochastic control process. At each time step, the process is in some state s, and the decision maker may choose any action s that is available in state s. The process responds at the next time step by randomly moving into a new state *s'*, and giving the decision maker a corresponding reward $R_a(s, s')$.The probability that the process moves into its new state *s'* is influenced by the chosen action. Specifically, it is given by the state transition function $P_a(s, s')$. Thus, the next state *s'* depends on the current state s and the decision maker's action a. [3]

1.3 Properties

A Markov decision process is a 5-tuple (S, A, P_a, R_a, γ) [3], where

- S is a finite set of states,
- A is a finite set of actions (alternatively, A_s is the finite set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state *s'* at time *t+1*,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state *s'*, due to action a,

- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

The agent's action selection is modeled as a map called *policy*:

$$\pi : S \times A \to [0, 1]$$
$$\pi(a|s) = P(a_t = a|s_t = s)$$

The policy map gives the probability of taking action a when in state s.[3]

Value function $V_\pi(s)$ is defined as the expected return starting with state s, i.e. $s_0 = s$, and successively following policy π. Hence, roughly speaking, the value function estimates "how good" it is to be in a given state.

$$V_\pi(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s],$$

where the random variable R denotes the return, and is defined as the sum of future discounted rewards

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,$$

where r_t is the reward at step t, $\gamma \in [0, 1]$ is the discount-rate.[3]

1.4 Q-learning algorithm

Q-function or state-action utility function provides a utility value for executing action a in state s:

$$Q : S \times A \to \mathbb{R}$$

Agent that learns a Q-function does not need a model of the form $P(s|s, a)$, either for learning or for action selection. For this reason, Q-learning is called a model-free method.

Q-function learning algorithm is one of the Temporal Difference learning algorithms (TD) that update state-action function iteratively until it represent all constraints of the domain.

$$Q'(s, a) \to Q(s, a) + \alpha(R(s) + \gamma \max_a Q(s', a') - Q(s, a)), \tag{1}$$

where $R(s)$ - reward for moving to the state s, γ - future reward discounting factor, α - learning rate. [4]

2 Project domain

2.1 Overview

A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum (Figure 1).

The mountain car problem appeared first in Andrew Moore's PhD Thesis (1990). It was later more strictly defined in Singh and Sutton's Reinforcement Leaning paper with eligibility traces. The problem became more widely studied when Sutton and Barto added it to their book Reinforcement Learning: An Introduction (1998).

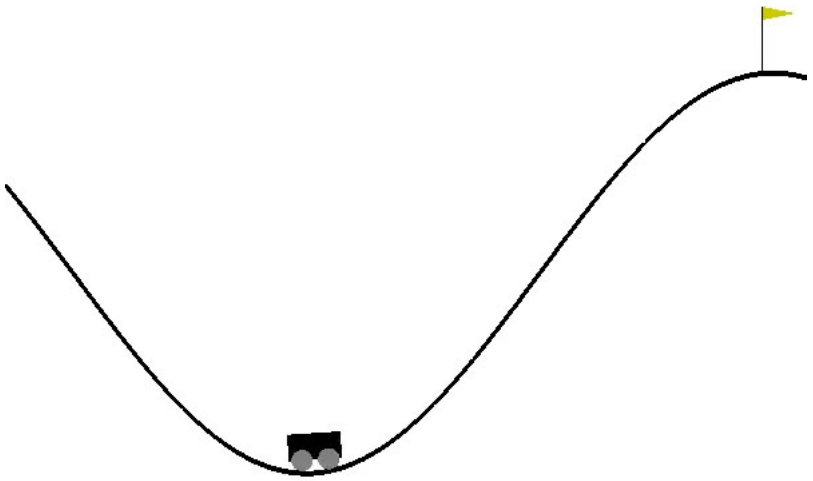

Figure 1: Mounting car domain

2.2 GYM library

As a model of the domain was used GYM library (`https://gym.openai.com`), which contain list of environments such as classical "Inverted pendulum swing up" domain, "Atari" computer games, environments with physical world simulation.

Each environment communicate with algorithm through observation, action and reward . To process action and obtain new state observation agent must call step()

function. As a result, environment returns a tuple with new state observation, reward receiver from environment and boolean episode termination value.

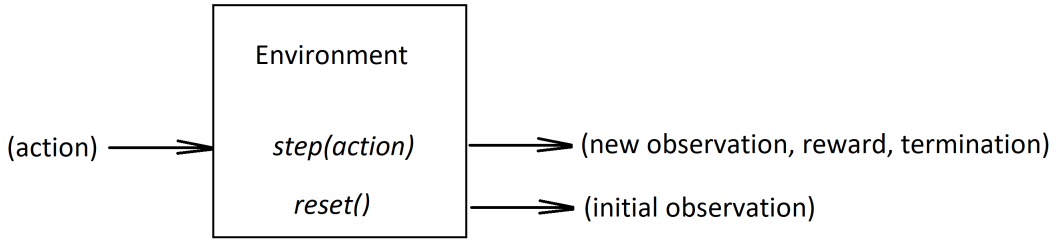

Figure 2: GYM environment

2.3 Domain specifications

Each environment in GYM library has different data sizes. For the "Mounting Car" they are:

Observations: are contain of 2 continuous values of velocity and position:

$$O = \{Vel, Pos\},$$

where $Vel \in [-1.2, 0.6]$ and $Pos \in [-0.07, 0.07]$.

Actions: are discrete action ids:

$$A = \{0, 1, 2\},$$

where 0 - move back, 1 - no action, 2 - move forward.

Rewards: are floating point values. Each step, including terminal, gives -1.0 reward.

Termination: boolean value.

$$T = \{True, False\},$$

where $True$ - termination of episode due to reaching of the goal or taking *200* steps in current episode.

3 Implementation

3.1 Project code structure

Code base consist of following classes:

- *Environment* - wrapper class for GYM environment class
- *Academy* - class responsible for properly instantiating agent instances, managing training logs, final model saving and restoring.
- *Agent* - base class for RL agent implementation
- *Couch* - class that provides methods to train and evaluate agent's performance in the domain

3.2 Agents

3.2.1 Random Agent

For comparison with other agents and development purposes there was created agent that acts randomly.

3.2.2 Look-up table Agent

To determine value of Q-function for every state-action pair agent use a table of a finite size.

Since, as described in Section 2.3, domain state observation consist of 2 continuous variables. Thus, observation received from domain discretized into natural numeric value in range [0, 40].

In such a way, Q-values table has resulting size of 40x40x3 of floating point values.

3.2.3 Function approximation Agent using Multi Layer Perceptron

Since domain state observation is continuous values Q-learning can be combined with function approximation. As solution Multi Layer Perceptron (MLP) architecture was selected. Figure 3.2.3 shows the approximation of the Q-function with MLP for Q-learning.

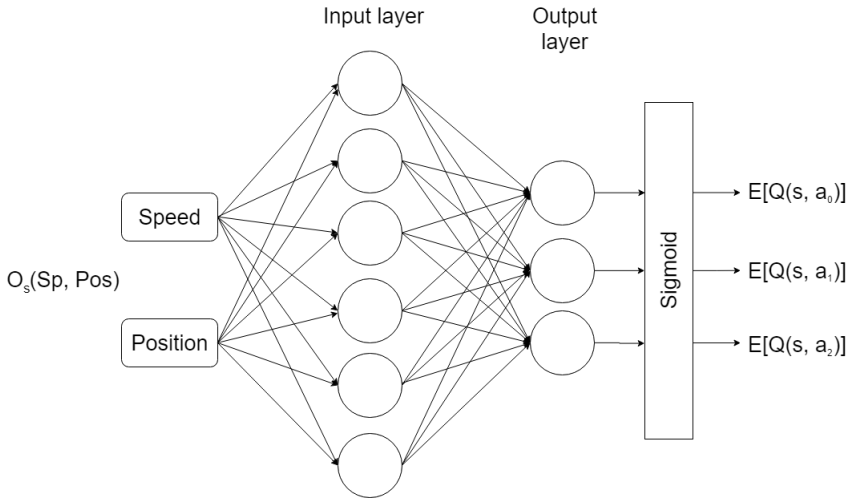

Figure 3: Q-function approximation model using MLP architecture

- Input layer - 2x6
- Output layer - 6x3
- Sigmoid activation function after final layer
- Argmax function over output vector to determine optimal policy.

The learning is based on updating or adjusting the weight matrices of the Neural Network using the equation of the update of the classic algorithm of Q-learning and the algorithm Back propagation.

Loss function is determined as squared difference between current state-action utility estimate and updated one:

$$Loss(s, a) = \frac{1}{2}(Q(s, a) - Q'(s, a))^2,$$

where $Q'(s, a)$ - updated utility value according to (1).

Described model was implemented in Python using Tensorflow computation library.

4 Results and conclusions

Conclusions about performance of agent's model were done by testing agent for 2 parameters:

- Average game score - amount of steps during episode to reach goal state.
- Convergence speed - amount of episodes after which agent's Average game score doesn't improve more than by 1%.

Results:

- *Random Agent*

  Random Agent wasn't able to reach goal even once. It seams because of tremendous domain state space - two continuous input variables. Thus, probability of choosing acceptable sequence of action is so low that there are no chances for this type of agent to reach final state.

- *Q-table Agent*

  Training of look-up table based agent was done during 200000 episodes with learning rage α=0.95. After around 7000 episodes agent start reaching goal state but only after 50000 episodes it was able to finish successfully in more than 50% of iterations. Algorithm converged after around 155 thousands episodes and reached value of 86% won episodes out of 100.

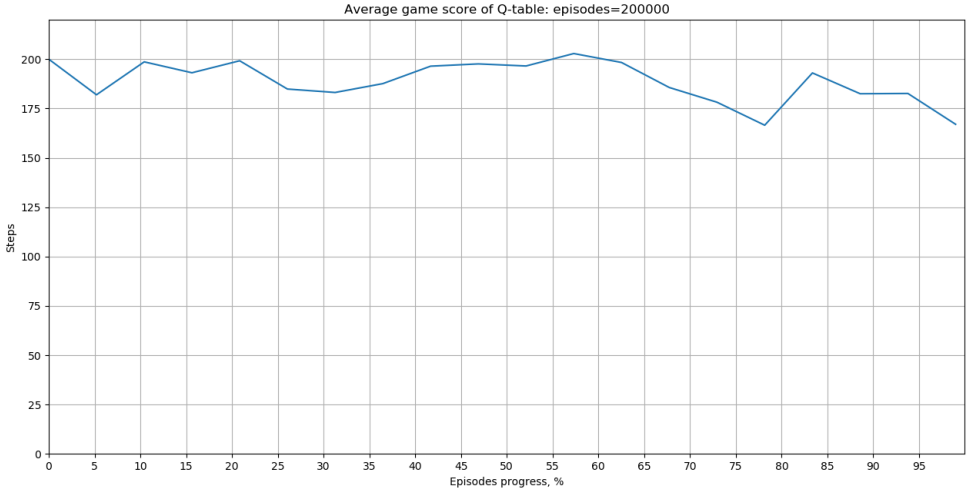

Figure 4: Smoothed average game score during training Q-table agent

- *MLP Agent*

  MLP agent was able to converge in less then 1000 iterations, reaching out up to 94% of won episodes out of 100 tries.

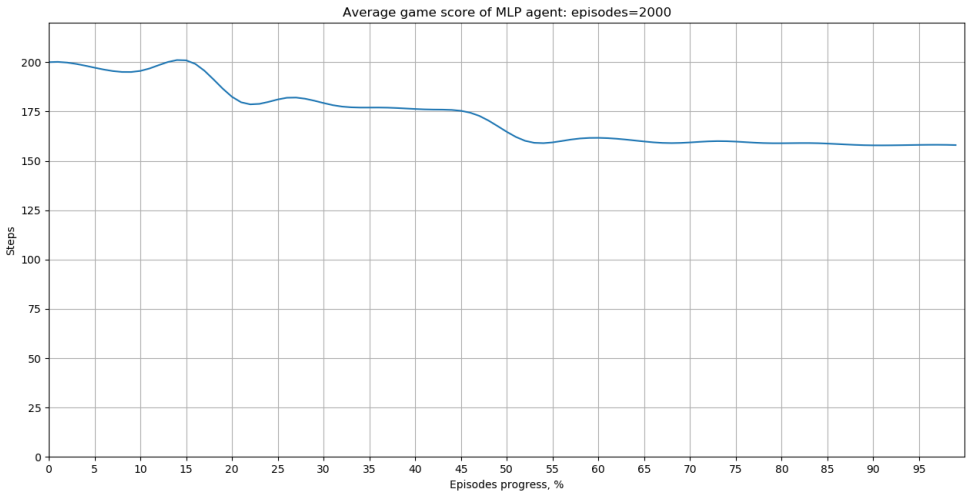

Figure 5: Smoothed average game score during training MLP agent

In general Q-function learning algorithm show as robust and as pointed in [2] it always find optimal policy if it if exist but convergence speed depends of initial hyper-parameter values. MLP model agent over-performs look-up agent model in final game score and convergence speed with final 94% of won games.

References

[1] Mezaache Hatem and Foudil Abdessemed. Simulation of the navigation of a mobile robot by the qlearning using artificial neuron networks. 547, 01 2009.

[2] Francisco S. Melo. Convergence of q-learning with linear function approximation, 2007.

[3] Wikipedia, The Free Encyclopedia. Markov decision proces, 2018. [Online; accessed 20-July-2018].

[4] Wikipedia, The Free Encyclopedia. Q-learning, 2018. [Online; accessed 20-July-2018].