Reinforcement Learning on Markov Decision Processes

Damianos Dimitrios

February 22, 2024

1 Introduction

Reinforcement learning plays an significant role these days, as it is widely used to improve the performance of several algorithms and pre-trained models upon solving specific tasks. In this project, we explore the application of several reinforcement learning algorithms, designed for solving problems modeled as Markov decision processes. Specifically, we'll focus on Q-learning and its variants, applied on OpenAI's Gym library environment.

2 Reinforcement learning

Reinforcement learning (RL) is a machine learning paradigm focused on enabling agents to learn optimal decision-making strategies through interaction with an environment. Unlike supervised learning where labeled data is provided, or unsupervised learning where patterns are inferred from unlabeled data, RL agents learn by trial and error, receiving feedback in the form of rewards or penalties based on their actions. The core challenge in RL is to discover a policy, a mapping from states to actions, that maximizes the cumulative reward over time. This involves navigating the trade-off between exploration, where the agent tries new actions to discover their consequences, and exploitation, where the agent leverages its current knowledge to maximize rewards. Navigating through a problem and finding the optimal solution requires a mathematical modelling, which in the case of RL is provided by the framework of Markov Decision Processes (MDP).

2.1 Markov Decision Processes

MDP is mathematical model used to describe decision-making scenarios in which an agent interacts with an environment over a series of discrete time steps. The key assumption in an MDP is the Markov property, which states that the future state of the environment depends only on the current state and action, not on the history of states and actions that led to it. The components of an MDP are:

1. States: All possible situations of the environment
2. Actions: The available choices of the agent
3. Transition probabilities: The likelihood of taking a particular action
4. Rewards: values that the agent receives upon taking an action and transitioning from one state to another

Common RL algorithms used for solving MDPs are Q-learning and its variants (SARSA, DQN) , and policy gradient methods, which use gradient ascent methods to update policy's parameters towards maximizing expected returns. In this project, we'll focus on Q-learning and its variant, which we'll discuss below.

3 Reinforcement Learning Algorithms

3.1 Q-learning

Q-learning [1] is a *model-free*, *off-policy* reinforcement learning algorithm that finds the next best action given a current state. It does not require a model describing the environment (model-free), and may come with its own rules to solve the task at hand , disregarding any policy given to follow (off-policy). To achieve this, it uses Q-values, which represent the expected values for each actions per state, and are stored in the *Q-table* .
Q-learning operates in an iterative manner, using the following components:

1. Agent: the entity that interacts with the environment
2. States: agent's possible positions in the environment
3. Actions: possible action the agent can perform in a given state
4. Rewards: the response provided by the environment to the agent for a given action

5. Episodes: an episode concludes when the agent stops taking new actions

6. Q-values: Metric used to measure an action for a given state, stored in Q-table

Q-values are a key element of the algorithm, since they affect the action-taking process. These values usually are initiated to zero or in random, and are updated iteratively, using the Bellman's equation:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * (r_{t+1} + \gamma * \max_A(Q(S_{t+1}, A)) - Q(S, A)) \quad (1)$$

where:
$Q(S_t, A_t)$: current estimate of Q-value
r_{t+1}: reward from action A_t
α, γ: learning rate and discount factor
$\max_A(Q(S_{t+1}, A))$: highest expected reward starting from the new state

Q-learning works through a trial-and-error process to learn the optimal behaviour. The behavior is modeled as an action-value function (Q-function) , which provide the Q-values. Training's goal is this function to represent the optimal long-term value of any action A in any state S, thus for the agent to follow the optimal behaviour in every state. At this point, we should note that Q-learning is an *off-policy* algorithm, since any updated Q-value takes in account the highest expected reward of the next state, regardless if the given policy directs the agent towards the action that yields this reward.

3.2 SARSA

SARSA (State-Action-Reward-State-Action) [2] is an *on-policy* reinforcement learning algorithm used for learning a policy to make decisions in a Markov decision process (MDP). It's similar to Q-learning but differs in that it updates the Q-values based on the current policy's action selection.

At each time step, the agent chooses an action based on the current policy. Usually the policy used is the $\epsilon - greedy$, where the agent choose the best action (the one with the greater q-value) with a probability $1 - \epsilon$, and a random action with probability ϵ. Executing the action, the agent transports to a new state, and it receives a pre-defined reward. Using this reward, the Q-values are updated using the following rule:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * (r_{t+1} + \gamma * (Q(S_{t+1}, A_{t+1})) - Q(S, A)) \tag{2}$$

where:
$Q(S_t, A_t)$: expected reward for action A in state S
r_{t+1}: reward from action A_t
α, γ: learning rate and discount factor
$Q(S_{t+1}, A_{t+1})$: expected reward from the next state when taking the next action following the current policy

As one can see, the update depends on current state and action (S_t, A_t) , as well as on the next state and next action (S_{t+1}, A_{t+1}) produced by the policy. This means that SARSA learns the Q-function and the policy simultaneously during training, thus making it an *on-policy* algorithm.Finally, SARSA usually requires more training samples than Q-learning, and a small learning in order to converge.

3.3 Deep Q-Network

Deep Q-Network (DQN) [3], proposed by DeepMind in 2013, uses deep neural-networks to approximate the Q-function. Specifically, replaces the Q-table with a deep neural network (MLP, CNN etc.). This enables to handle high-dimensional state spaces such as images or continuous environments, while it can capture any dependencies between actions and states better.

Some of its advantages are its scalability and generalization, while it has proven that can be successful in complex tasks, such robotic control tasks. Still, it has its disadvantages, where is very sensitive to its hyperparameters, and its possible to have issues during training. such as divergence or slow learning. Despite these challenges, it remains a popular and powerful technique in the field of reinforcement learning.

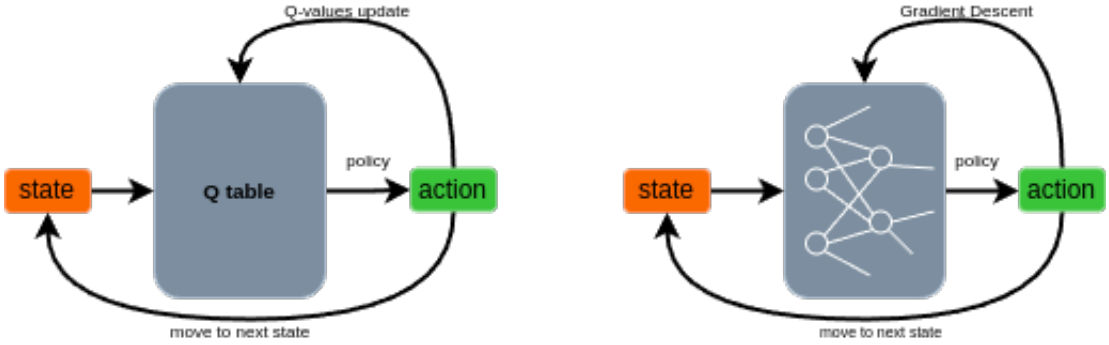

Figure 1: Comparison of Q-learning/SARSA and DQN. In Q-learning/SARSA, Q-function is represented as a table containing Q-values (Q-table). In DQN, Q-function is represented as a neural network.

4 Environments

For our experiments and applications, we'll be using the OpenAI's *gym* library [4]. The Gym interface provides a simple and pythonic representation of several environments representing general RL problems (such as CartPole, Acrobot etc).

4.1 Mountain Car

We are using the Mountain Car environment, which can be seen in the figure below.

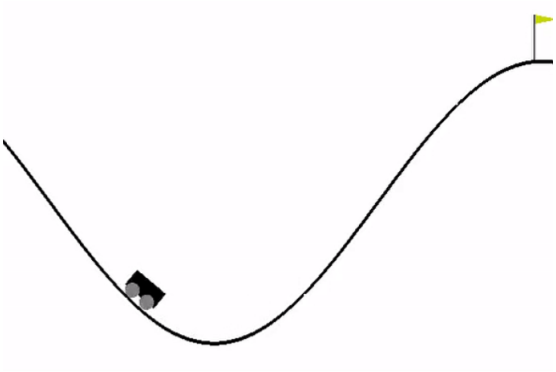

Figure 2: The *MountainCar-v0* environment

Introduced in [5], it is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill. The trick is that the car cannot reach the flag at the right hill by simply accelerating right, but it requires to build momentum by climbing the left hill.

4.1.1 State Space

Each state consists of the position of the car along the x-axis, in the range $[-1.2, 0.6]$, and the velocity of the car, in the range $[-0.07, 0.07]$. The starting state is assigned a uniform random value in $[-0.6, -0.4]$, with velocity assigned to 0.

4.1.2 Action Space

There are 3 deterministic actions:

1. Accelerate left
2. Don't accelerate
3. Accelerate right

4.1.3 Transition Dynamics

Given an action, the mountain car follows the following transition dynamics:

$$velocity(t+1) = velocity(t) + (action - 1) * force - cos(3 * position(t)) * gravity$$

$$position(t+1) = position(t) + velocity(t+1)$$

where $force = 0.0001$ and $gravity = 0.0025$ by default.

4.1.4 Rewards

In our experiments, we used the following 2 reward strategies. The first one is the default reward strategy, provided by the environment

$$reward = \begin{cases} 0, \text{if goal reached} \\ -1, \text{otherwise} \end{cases} \tag{3}$$

For the second one, we decided to experiment and provide reward relative to position as follows:

$$reward = \begin{cases} 10, \text{if goal reached} \\ -(1 + position)^2, \text{if position>-0.4 or position<-0.8} \\ -1, \text{otherwise} \end{cases} \tag{4}$$

The idea was to provide the agent with "motivation" to climb the left hill $[-1.2, -0.8]$ before trying to climb the right one $[-0.4, 0.5]$, and to avoid the valley at the center.

5 Training

5.1 Details

The policy used in all algorithms is the $\epsilon - greedy$ policy, to encourage exploration over exploitation at the beginning. The ϵ parameter is initialized to 1, and each time the goal is reached, it is discount by a factor of 0.995. During the exploitation phase of the algorithm, the action with the maximum Q-value is chosen.

5.1.1 Q-learning, SARSA

For SARSA and Q-learning , we set the learning rate a and the discount factor γ are set to 0.001 and 0.95 respectively. During training, the Q-table is updated iteratively, according to equations 1 and 2.Due to hardware limitations, we limited the possible steps of each training episode to 1000.

5.1.2 DQN

For our agent, we used a dense-network consisting of 2 hidden layers, with 64 and 32 hidden neurons respectively, and ReLU activation layers between them and the output layer. During each training step, expected rewards are calculated using the equation 1, where $Q(S_{t+1}, A)$ is produced through inference to the model using the current state as input. Using these expected rewards, the Q-values of all actions corresponding to the current state are updated and will be used as desired targets. Applying Gradient Descent to the loss between expected and produced rewards results to the learning of the optimal policy.

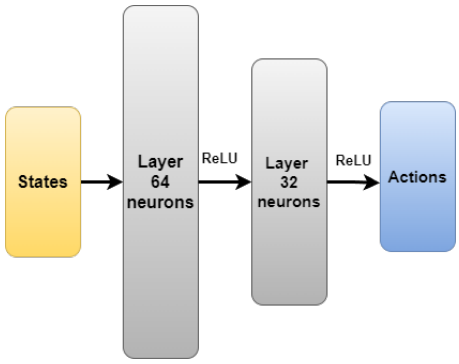

Figure 3: DQN's architecture

5.2 Results

We trained all 3 algorithms for 1000 episodes using both rewarding strategies. In Figure 4 we have the results of training with reward strategy 3. As we can see, DQN reached the goal 5% of the training session, while Q-learning and SARSA 2.3% and 1.8% respectively.

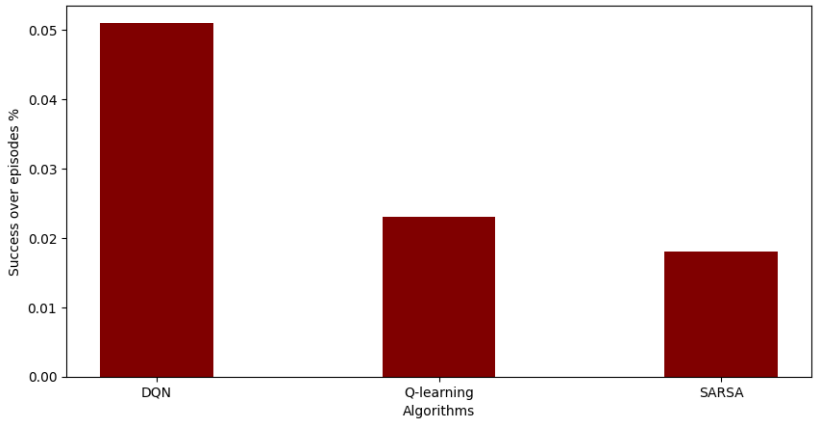

Figure 4: Percentage of successes with reward strategy 3

In Figure 5 we have the results of training with reward strategy 4. Here, DQN reached the goal 20% of the training session, while Q-learning and SARSA 2.5% and 2.3% respectively.

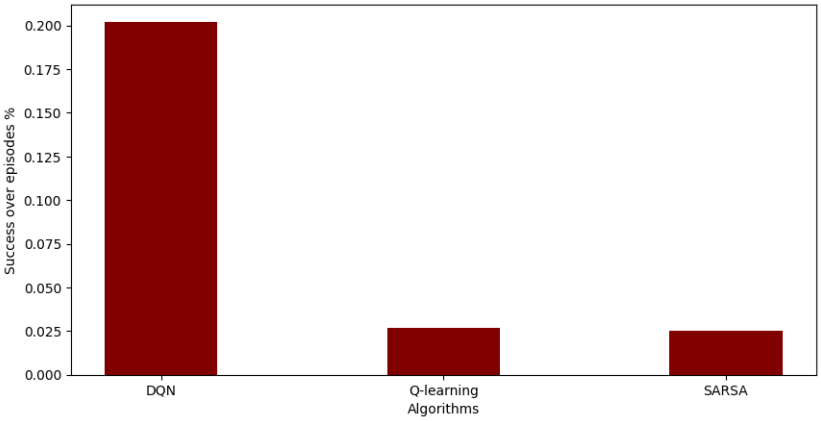

Figure 5: Percentage of successes with reward strategy 4

Below, we present the mean cumulative reward values for both strategies:

Mean cumulative reward		
Algorithm	Reward Strategy 3	Reward Strategy 4
DQN	-654.45	-449.3
Q-Learning	-1451.17	-1161.31
SARSA	-1495.17	-1131.87

As it is indicated above, Deep Q-Learning achieved significantly better results than the other 2 algorithms with both reward strategies, while Q-learning was slightly better than SARSA. It is also indicated that our custom reward strategy 4 achieves better results than the default one.

5.3 Final notes and future work

Due to the use of the $\epsilon - greedy$ policy, in some of our training sessions the agent failed systematically to reach the goal, and resulted to the agent getting stuck in the valley while trying to climb only one of the hills. We believe this happened due to the stochastic nature of the policy, where the randomly-taken actions at the beginning failed to explore sufficiently. We solved this issue by simply starting the training session from the beggining.

In the future, we could experiment with various reward strategies, and perhaps improve the reward strategy 4, while using the algorithms for more episodes, to observe any convergance.

6 References

1. Q-learning (Watkins, 1989)
2. A Theoretical and Empirical Analysis of Expected Sarsa
3. Deep Q-Network
4. OpenAI Gym
5. Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, November 1990.