CentraleSupélec

Reinforcement Learning Project:
Solving the Mountain Car problem

Davide Rendina
Computer Science Department
CentraleSupélec, Université Paris-Saclay
davide.rendina@student-cs.fr

Niccolò Morabito
Computer Science Department
CentraleSupélec, Université Paris-Saclay
niccolo.morabito@student-cs.fr

Contents

1 Problem setting **2**
1.1 Environment and metrics 2
2 Algorithms **3**
2.1 Q-Learning 3
2.2 Deep Q-Networks 4
3 Conclusions and Future Work **6**

1 Problem setting

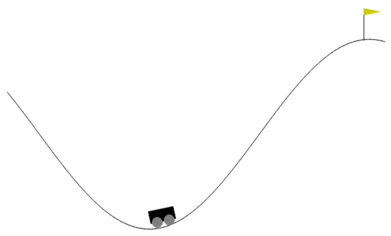


Figure 1: Mountain car [1]

The Mountain Car problem is a classic problem in the field of reinforcement learning. It is used to demonstrate the capabilities and limitations of various reinforcement learning algorithms.

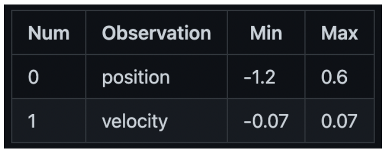
In this problem, an under-powered car is placed on a one-dimensional track, at the bottom of a valley. The car is too weak to climb the mountain directly, so it must build up momentum by driving back and forth within the valley. The goal is for the car to reach the top of the mountain, which is located at the right end of the track.

The car is controlled by three actions: accelerate left, don't accelerate, and accelerate right. The car's position and velocity are the only information available to the learning algorithm. The task is episodic, and the agent receives a reward of -1 at each time step until it reaches the goal. If the car is not able to reach the goal in 200 steps, the episode must be considered failed.

The Mountain Car problem is considered a challenging problem because the state space is continuous and the reward signal is sparse and delayed. The car must build up sufficient momentum to reach the mountain, but if it goes too fast, it will overshoot the mountain and have to start again from the bottom. Therefore, the car must learn to balance exploration and exploitation in order to find the optimal solution.

1.1 Environment and metrics

The environment has been provided by [1] as part of the Classic control environments. In this environment, the **observation space** is a continuous space with 2 dimensions (position and velocity of the car), represented as a Box object. The **action space** is a discrete space with 3 actions (accelerate left, don't accelerate, and accelerate right), represented as a Discrete object. This can be seen below:

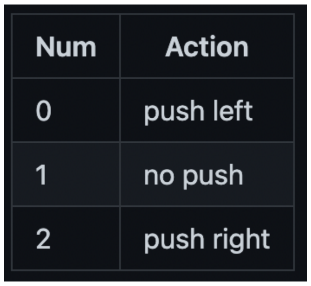


Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

Figure 2: Mountain car observation space [1]

The state of the car is thus represented by its position and velocity represented as a vector. The agent must then select an action from the action space which is sent to the environment that returns a new state as well as a reward. For each step in which the car does not reach the goal the environment returns a reward of -1. Finally, the episode terminates if either of the following happens:

- Termination: The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill)
- Truncation: The length of the episode is 200.



Num	Action
0	push left
1	no push
2	push right

Figure 3: Mountain car action space [1]

2 Algorithms

No model is provided, therefore the choice of algorithm cannot consider model-based algorithms. Moreover, the problem is single-process and, as stated before, the action space is discrete (with 3 possible actions). Prioritizing the sample efficiency, a clever choice to solve this problem should be based on Q-learning [3] [2]. In particular, since the observation space is continuous, it should be necessary to approximate the state-action value function with a neural network. However, for the sake of this project and to be able to compare the tabular and the approximate approaches, we decided to solve this problem by implementing both versions, Q-Learning and Deep Q-Learning. The following sections explain briefly the chosen parameters and add-ons for both approaches and compare the results.

2.1 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that is used to learn the optimal action-selection policy for a given environment. The goal of Q-learning is to learn a function Q(s, a) that estimates the cumulative reward for selecting action a in state s. One of the main challenges of the Mountain Car problem is that the dynamics of the car's movement are complex, as the agent must learn to balance exploration and exploitation in order to find the optimal solution. In order to try to overcome this challenge in our implementation we applied an ϵ-greedy policy, where the value ϵ will determine the likelihood of the agent choosing a random action rather than the action with the highest estimated cumulative reward (as determined by the Q-table).

In our implementation of the Q-learning algorithm, in each episode, the agent selects an action to take in the current state based on the current policy and the Q-table. The agent then takes action, observes the resulting state and reward, and updates the Q-table to reflect the new information. The Q-table is updated using the Bellman equation, which states that the value of a state-action pair is equal to the reward for taking the action in that state plus the expected value of the best action in the next state. Finally, the policy is updated at each step using an exploration-exploitation trade-off, where the agent will either take a random action with probability ϵ or select the action that is expected to maximize the reward with probability 1-ϵ. We tested two different experiments: in the first one, ϵ is set to 0.5 which means that the agent will take a random action at each state with 50% probability; in the second one, we set ϵ initially to 0.3 and we set a decay rate of 0.01 which will decay the ϵ at each state and allows the agent to gradually shift from exploring to exploiting as it learns more about the environment.

The parameters are set as follows:

- *number_states = 40*: This variable determines the number of states that the agent will use to discretize the continuous state space of the environment.
- *max_iteration = 5000:* This variable determines the maximum number of iterations that the learning algorithm will run for.
- *initial_learning_rate = 1.0:* This variable determines the initial learning rate that the learning algorithm will use. The learning rate determines the step size at which the agent updates its estimates of the action values (Q-values) based on new experiences.

- *min_learning_rate = 0.005:* This variable determines the minimum learning rate that the learning algorithm will use. The learning rate will decrease over time, and this variable sets the lower bound on the learning rate.
- *max_step = 10000:* This variable determines the maximum number of steps that each episode in the learning process will last.
- $gamma = 1$: This variable determines the discount factor that the learning algorithm will use. The discount factor determines the importance of future rewards compared to immediate rewards.

In both experiments, the Q-table learned by the Q-learning algorithm is used to derive a policy by selecting the action with the highest expected reward in each state of the environment. This policy is then tested on 50 episodes from which an average score is derived.

Less exploratory experiment: The value of ϵ is initially set to 0.5 with no decay function. In this case, the **average score** obtained by the 50 episodes is -200 which means that on average the car failed to reach the top.

Less exploratory experiment: The value of ϵ is initially set to 0.3 and it is decreased over time using an exponential decay function with a decay rate of 0.01. In this case, the **average score** over the 50 episodes is -118 which means that on average the car reached the top.

In conclusion, by using a moderate ϵ and slowly decay at each episode the ϵ-greedy policy is able to exploit the exploration-exploitation trade-off and make the car learn to reach the top.

2.2 Deep Q-Networks

As explained previously, the observation space of Mountain Car is continuous, which is why for Q-learning it was necessary to discretize it. Using neural networks, discretization can be avoided and it is possible to represent the state-action value function by Q-network with weights w [4].

$$\hat{Q}(s, a; w) \approx Q(s, a)$$

In order to do this, a very simple neural network with two dense hidden layers is used, taking 2 values in input (state size) and outputting 3 values, one per each possible action. Also for the sake of this algorithm, a **ϵ-greedy policy** is applied to choose the next action balancing exploration and exploitation. To guarantee convergence, the value of ϵ is decreased during the training as the random exploration becomes gradually less important.

As a reminder, some important issues with online deep RL are related to the sampling of experience: consecutive updates are strongly correlated (even if the gradients are supposed to be sampled i.i.d.) and an update is carried out for every new experience tuple (while mini-batches are typically better). In order to face these problems, a **replay buffer** is implemented, where the newly generated experiences are stored. As soon as the buffer becomes full enough, some experience tuples are randomly sampled for the training. This allows us to reduce the correlation between records, improve sample efficiency and support mini-batch updates.

Moreover, a **separate target network** is added to address the soft divergence. Its parameters, used to compute the bootstrap targets, are fixed and updated only after each episode.

The training, for each episode, is carried out in 200 iterations (which is the maximum allowed by the environment). An action is chosen with the ϵ-greedy approach (random action with probability ϵ, greedy action otherwise), and the consequent experience is stored in the replay buffer. Then, the train network is fitted to a batch sampled from the buffer by adding the bootstrap targets to the targets' rewards. For an experience i,

$$y_i = r_i + (1 - d_i) * \gamma * \max_{a'} \hat{Q}(s_{i+1}, a'; w^-)$$

where d_i is 1 if the episode is over, 0 otherwise; γ is the discount factor and w^- are the frozen parameters. The parameters are set as follows:

- *neural network architecture*: the neural network is composed of two hidden layers of dimension 24 and 48 respectively, both activated by a ReLU function;
- $\gamma = 0.99$: this discount factor makes the algorithm consider basically all the rewards. The discount factor determines the importance of future rewards compared to immediate rewards;
- *learning rate = 0.001*: it is the learning rate of the Adam optimizer for the neural network;
- *max dimension of replay buffer = 20000*;
- *number of episodes = 400*;
- *batch size = 32*: number of elements that are sampled from the replay buffer for the training;
- ϵ: the parameter for the ϵ-greedy policy is initialized to 1 and, after each episode, decreased by 0.05 until it reaches the minimum value of 0.01;
- *max_iteration = 200:* This variable determines the maximum number of iterations that the learning algorithm will run for.

Using the above parameters, it has been empirically seen that the algorithm starts to stably reach the goal after around 250 episodes. Figure 4 shows the behaviour of the score (computed as the opposite of the reward at the end of each episode) during the DQN training. Only when the score is below 200 it is possible to consider that the car reached the goal. Even if the car starts to reach the goal fairly often after 130 episodes, a sufficiently stable convergence is reached only after 250 episodes, where it can be seen that the majority of the episodes are positively terminated. However, around 300 episodes, there are still too many failed episodes.

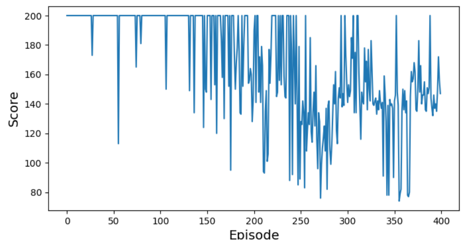


Figure 4: Score per episode during DQN training

In general, even after convergence, the score significantly varies among episodes, which is why we decided to test the score of the models at different points of the training, not only the finally obtained one. The idea is that, even if we expect all the models obtained after 250 episodes of training to be on average able of completing the task, there could be interesting differences in the quality of the completion. Also in this case, to test a model we made it run for 50 episodes with the learned policy, averaging the final reward. The results are summarized in the following table.

Number of training episodes	Average score on 50 episodes
250	142
275	108
300	200
315	155
400	141

Table 1: Results of models trained with a different number of episodes averaged on 50 episodes.

Table 1 confirms the behaviour that was shown during training. The models obtain a varying average score depending on the number of episodes they have been trained on, but not with a constant trend. In fact, the score decreases until reaching the minimum from 250 to 300, then it explodes (i.e. it is no longer able to reach the goal in time) and then starts decreasing again, not reaching the best score of 108 anymore. In order to converge, the algorithm learns a slightly worse behaviour, and the animations generated by the Gym environment visually show that: the car tends to reach the goal in a very optimal way (swinging a bit on the left at the beginning to have the right energy to reach the goal immediately after) when trained for 250 episodes. After that, it learns to swing more, reaching the goal only on the second attempt; clearly, this affects the average score which is higher for the greater number of actions necessary to reach the goal. A possible explanation could be the size of the replay buffer that, while deleting the old experience because the tuples of experience are more than 20.000, causes catastrophic forgetting, making the model forget previously learned information upon learning new information. Further attempts could study the behaviour of training with a bigger buffer.

3 Conclusions and Future Work

In summary, we propose two algorithms to solve the Mountain Car problem using the Gym environment, namely Q-Learning and Deep Q-Learning (DQN).

For Q-Learning, two different settings have been tested using an ϵ-greedy policy, testing two different values of ϵ. Results showed that with a slow decay of ϵ the agent was able to exploit the balance between exploration and exploitation obtaining, averaging on 50 episodes, a score of 118.

Using DQN, the best-achieved score is 108, slightly better than Q-learning. This could be explained by the lack of discretization in this second case, modelling the whole environment no information is lost through discretization. However, the small difference is paid with an important increase in the training time: even if Q-learning requires an order of magnitude more episodes for the convergence, the overall training time is significantly faster because of the simpler steps.

Regarding the improvements, different attempts have been carried out to tune the hyperparameters, including the value of ϵ, to adjust the exploration-exploitation trade-off, the value of γ to focus on the last rewards, but also all the neural networks' parameters (learning rate, number and dimension of hidden layers, activation functions, etc.) to improve the training time or to have a better convergence. However, all the attempts have led to no improvement or deeply slower convergence.

Regarding future work, trying other algorithms could be more interesting. In particular, actor-critic algorithms (like ACER [5]) could show different and interesting insights into the policy gradient, and could lead even to better results in terms of the number of steps necessary to reach the goal.

Moreover, another interesting part that has not been studied for this project is the reward system. By default for this problem, the reward is simply -1 for each additional step that does make the car reach the goal. This is clearly reasonable to minimize the number of steps to reach the goal, but smarter approaches could be tried as well. For instance, giving rewards proportionally to the velocity of the car (since the velocity is a prerequisite to reaching the uphill goal) or giving a positive reward for every action that keeps the car going in the same direction (to ease the necessary swinging behaviour).

References

Gym. Mountain car. URL https://www.gymlibrary.dev/environments/classic_control/mountain_car/.

B. Jang, M. Kim, G. Harerimana, and J. W. Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7:133653–133667, 2019. doi: 10.1109/ACCESS.2019.2941229.

A. G. B. Richard S. Sutton. *Reinforcement Learning*. 2018.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016. URL http://arxiv.org/abs/1611.01224.