Machine Learning Project

July 18, 2023

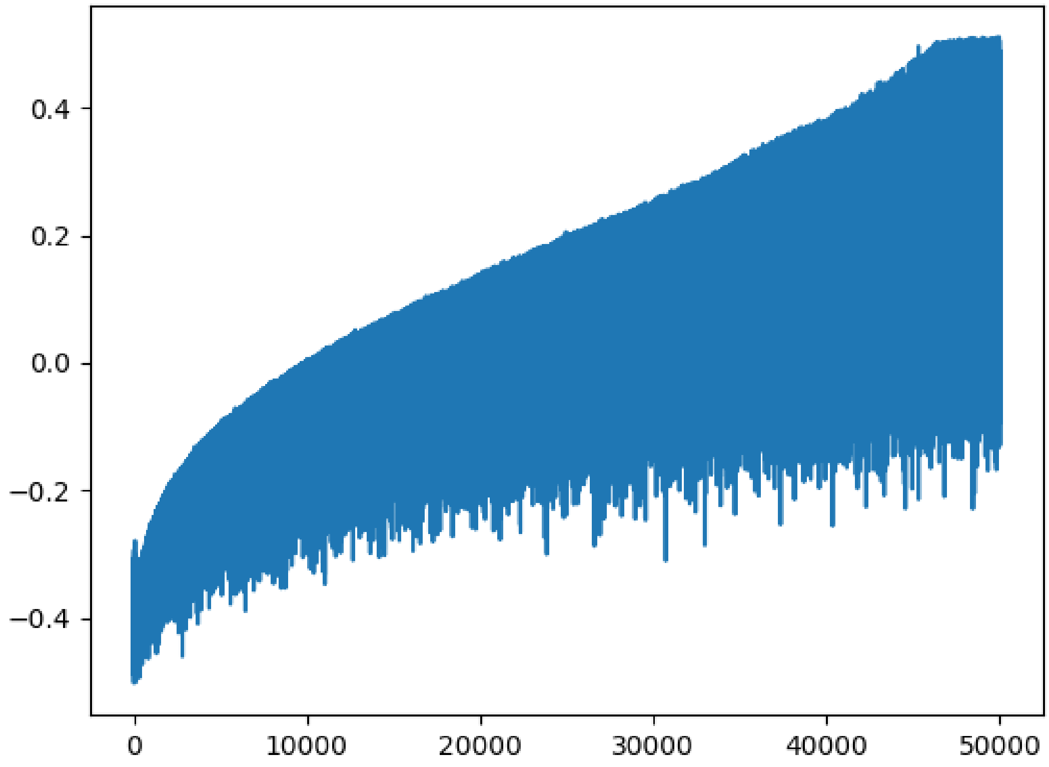
1 Introduction

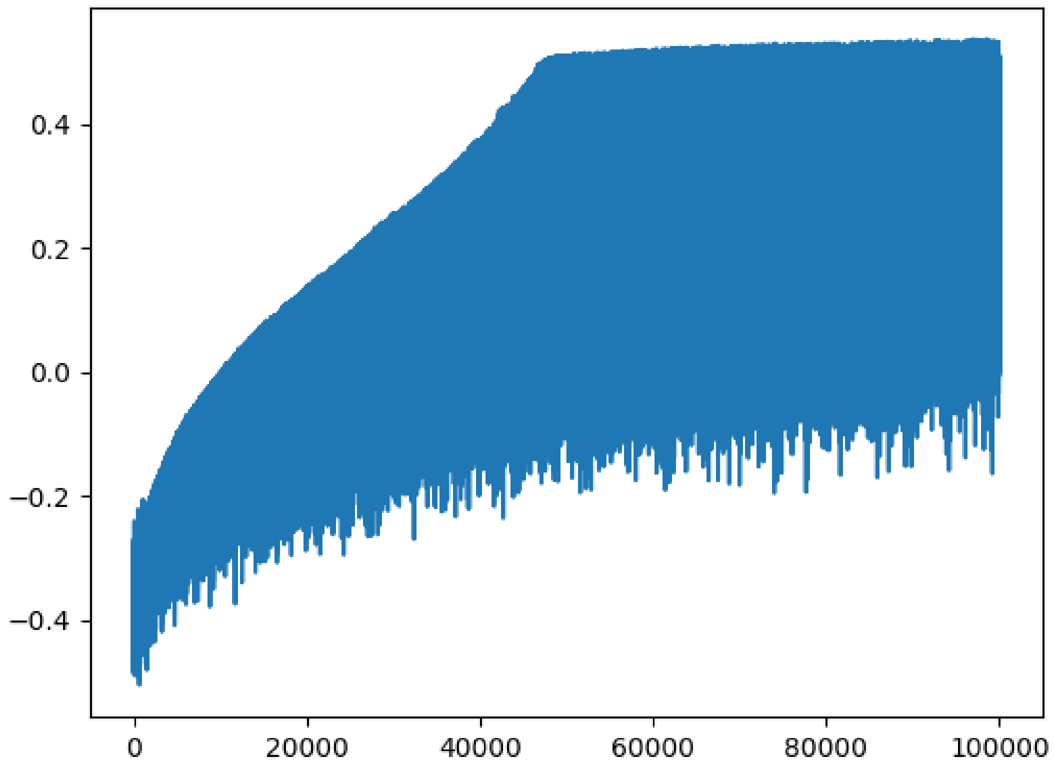
The environment used is Mountain Car, part of the Classic Control environments. The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill. There are two versions of the mountain car domain in gymnasium: one with discrete actions (the one used in this project) and one with continuous.
The action space is limited to 3 actions: accelerate to the left (0), don't accelerate (1), accelerate to the right (2).
Instead, the observation space is composed of a Box of two elements: the first element (the position of the car) can have a value between -1.2 and 0.6, while the second element (the velocity of the car) can have a value between -0.07 and 0.07. Both of them are represented as float32.
The transition dynamics update the position and the velocity of the car every time an action is performed on the environment.
The goal is to reach the flag placed on top of the right hill as quickly as possible, as such the agent is penalised with a reward of -1 for each timestamp. No reward is provided to the agent when he reaches the goal.
The starting position of the car is assigned a uniform random value in [-0.6, -0.4]. The starting velocity is always assigned to 0.
The episode ends either if the position of the car is greater or equal to 0.5, that represents the achievement of the goal (terminated), or if the length of the episode is 200 (truncated).

The provided implementations are done in Python: *Numpy* library is used for the Q-Table and *Tensorflow* library is used for the Neural Network.

2 Q-Table

Firstly, I developed a Q-Table based program to solve the environment.
It is composed of one class: the Q_Learning class. It is in charge of creating the Q-Table, storing values inside of it (*store* method), mapping states into indexes (*mapToVel* and *mapToPos* methods), selecting the action that must be performed in a certain state from the Q-Table (*act* method), and decreasing the exploration rate according to the decay rate (*adjust* method).
The first problem was that both position and velocity of the car were not natural numbers, but they were real numbers; in order to overcome it, I discretized both of them by approximating the real values to a fixed number of decimals, that can be arbitrarily set (I found out that 3 decimals both for position and velocity was pretty good).
Then, in order to store the data (state, action, Q-value), I used dictionaries, where the key was set equal to the combination of two strings: the one representing the state and the one representing the action; while, the value was equal to the Q-value computed. Although operating on it was very easy, it was also very slow; so, the final choice fell back to numpy arrays. Basically, I built a 3D numpy array, where the first dimension represented the position of the car (so its size depended on the number of decimals taken into account for position), the second dimension represented the velocity of the car (so its size depended on the number of decimals taken into account for velocity), and the last dimesion represented the action taken (there are only 3 actions, so its size is constant); obviously, the stored value is the Q-value computed.
In order to make everything working fine, I had to develop two mapping functions for both position and velocity; each one maps the value provided as input into an index of the 3D numpy array. By doing so, the search for a value was drastically sped up (the most common operation).
I made it run for both 50.000 and 100.000 episodes with an exploration rate that went from 100% to 1%, by decreasing it of 5% every 5 episodes, and the results were very good.

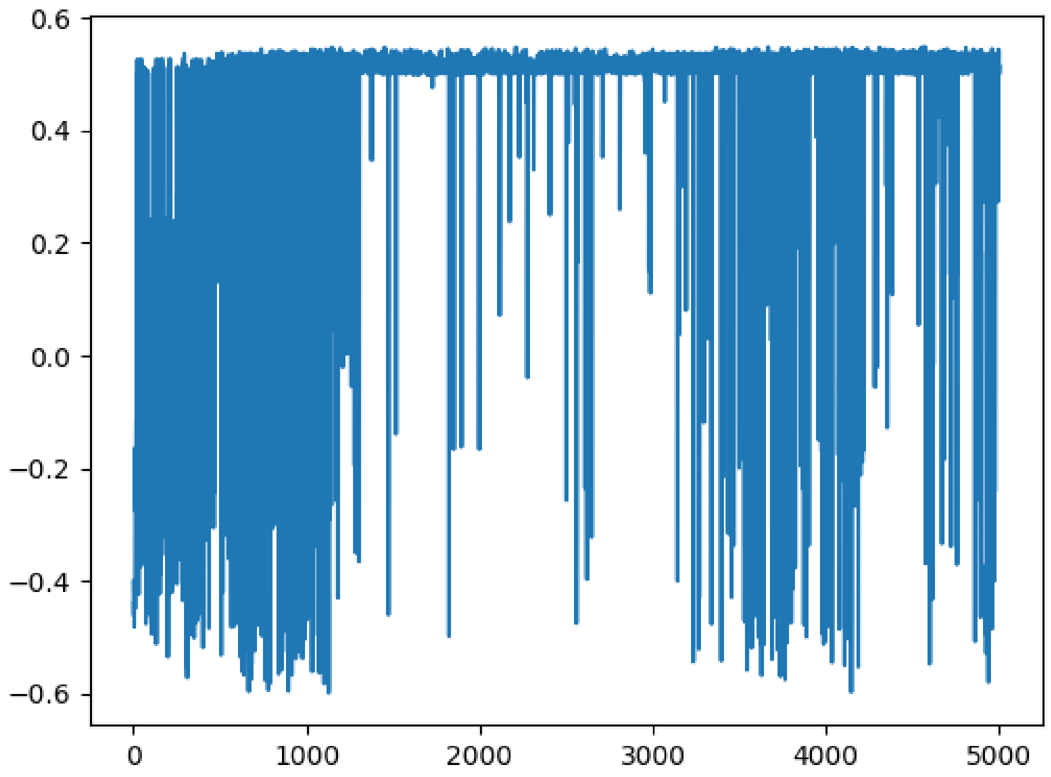

0.4
0.2
0.0
−0.2
−0.4
0
10000
20000
30000
40000
50000



3 Neural Network

Then, I developed a Neural Network based program to solve the environment; I encountered much more issues for this one than the one based on the Q-table.
It is composed of one class: the Deep_Q_Learning class. It is in charge of creating the Neural Network, storing data received from the environment into the Replay Buffer (*store* method), selecting the action that should be performed on the current state according to the strategy used (*act* method), training the Neural Network through the data stored in the Replay Buffer (*learn* method), adjusting the learning rate (*adjust* method), and saving the model of the Neural Network such that it can be reused (*save* method).

The first problem that I encountered was the creation of the Neural Network according to the environment parameters; indeed, while the input layer was quite easy to decide about, the output layer was much more difficult. At first, I set the network to have only one output value, but after many tries, I realized that having a number of output values equal to the number of possible actions was the best option.
By doing this, the output values of the Neural Network were the predicted values obtained by performing each action in the state passed as input. This helped me to select the best action to perform for each state of the car (the action with the highest predicted value). I also used it to train the Neural Network: for every stored state (each state was previously stored along with the action performed on it, the reward received by performing the action, the state in which the car was after performing the action and a boolean variable that specified if it was the last state for the episode), the values returned by performing each action were predicted by the Neural Network and the value of the one action that was actually performed during the training was changed according to the reward and the subsequent state stored. In order to achieve good performance, the Neural Network was trained every 10 steps (20 times per episode); such a thing affected drastically the time needed to properly train the network.
Then, I had to face a problem of memory leak that was caused by a function that I used for the training of the Neural Network: *predict*. Indeed, while using the function during the training of the Neural Network, the RAM usage kept increasing quickly until there was no more available memory. In order to fix it, I replaced all the occurrencies of it with the function *predict_on_batch*, that takes only one batch at a time (instead of taking as input the whole set of batches together). Furthermore, I also converted all the arrays that were being passed to the Tensorflow functions into constants; it helped to reduce the memory used.
I made it run for 5.000 episodes with an exploration rate that went from 100% to 1%, by decreasing it of 10% every episode, and the results were very different from the previous ones.



4 Comparison

The Q-Table approach is faster and easier to implement and works well for environments which have limited states. On the other hand, the Neural Network approach is more difficult to implement, but it is more suitable for more complex environments. For the case of Mountain Car with discrete actions, both of them solve it, but the Neural Network approach achieves the first goal faster (within a few episodes), while the Q-Table approach requires almost 50.000 episodes; furthermore, in order not to have an infinite Q-Table, the observation space must be discretized (action space is already discrete). In general, the simpler the environment is, the faster the Q-Table approach converges; conversely, the Neural Network approach converges faster if the environment has many possible states; indeed, in the Q-Table approach, the majority of states (in some cases, it is required the totality of the states) must have been explored for the convergence.