MountainCar-v313

Riccardo Vasumini*
Scuola di Ingegneria - Corso di Ingegneria Informatica M
University of Bologna - Alma Mater Studiorum
Bologna, BO 40132
riccardo.vasumini2@studio.unibo.it

Abstract

The MountainCar-v0 is one of the most famous problem among the OpenAI Gym's environments. In this report there will be shown an implementation with different techniques: deep Q learning, fixed Q targets and double deep Q learning. Each of them reaches a success average of 90% very quickly, but only with fixed Q targets and double Q learning the problem is solved as intended by OpenAI Gym's GitHub site: getting average reward of -110.0 over 100 consecutive trials.

1 Introduction

OpenAI Gym is a famous toolkit for developing and comparing reinforcement learning algorithms(2), many people has posted their own solutions for the environments offered by the platform using different techniques of reinforcement learning, sometimes also with some personalized implementation.

The description given by OpenAI Gym of the mountain car problem is the following:

> 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.

Speaking of reinforcement learning, usually the deep reinforcement learning is the most used by the users of OpenAI Gym, and discussions or suggestions of the best set of hyperparameters to use are a daily matter both for newbies and experts.

In this report I've tested three different implementation of deep reinforcement learning, in particular I've used:

- A simple deep Q network (DQN)
- DQN with fixed Q targets
- Double DQN

1.1 Simple DQN

A simple deep Q network (DQN) is a neural network used to approximate the Q value function:

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

The network receives as a input the state and gives as output all the possible Q values, we have a Q value for each possible action and we have to chose the biggest one.

*GitHub: https://github.com/JJonahJson

The improvement of the estimation of the Q values is due to the use of Experience Replay, a solution which allows to remember random states, actions and rewards saved in a buffer and feeds the network with them.

1.2 Fixed Q Targets

The fixed Q targets technique uses two DQNs instead of one to perform a more stable training, these are named train net and target net.

The train net has the same behaviour of the previous described simple DQN, the target net copies the weights of the train net every time a fixed amount of time steps has passed. The training results more stable because before the update of the weights was done using the same DQN in the TD (Temporal Difference) Error:

$$\Delta w = \alpha[R(s,a) + \gamma max Q'(s',a',w) - Q(s,a,w)])\nabla Q(s,a,w)$$

but now we use the target net in the first term of the TD, and the train net in the second; the weights of the target remains fixed while the weights of the train improves, so we can see TD error reducing.

1.3 Double DQN

If my DQN overestimate an action in a certain state, predicting a high Q value, the overestimated weight will be propagated for future estimation; double DQN solves this problem of overestimation. The train net will select the next actions and the the target net will evaluate it.

2 Implementation

Going forward, after the decision of the techniques to test and compare to solve the mountain car problem, the main argument of discussion when it comes the moment to implement your own solution is how you model your DQN and what hyperparameters setting you use.

Another relevant still open discussion is wether or not is proper to shape the reward function, which sees two fronts emerging: one that claims that it is lawful, especially in environments with a high number of variables at stake, and one that accuses this change of negating the concept of reinforcement learning.

I will go into these topics in the sections below.

2.1 Reward shaping

Many users, while they're trying to solve the mountain car problem, start to think if it could be useful or not shape the reward. That is because the mountain car problem gives as reward -1 for every time step the car hasn't reached the goal, which means that no matter if you're fixed or if you were so close to the goal, the reward will be the same.

It's true that reward shaping is accepted in many other cases of reinforcement learning, due to their complexity, but it seems that it's not the case.

When I was at the first step of this project I also tried to change the reward a little bit, trying not to make the problem too simple, so I've simply encouraged the car when it was nearer to the goal giving as reward still the -1 but adding to it the current position of the car.

It worked a little, but the average reward was not improving as much as I wanted, furthermore, after changing the reward, it is not possible to say whether the problem was solved or not because it is defined on the reward itself.

I understood that the original reward is the best for solving the problem, because, defined as it is, pushes the car to reduce the number of step required to reach the goal, so I've left apart the reward shaping.

2.2 Network architecture and hyperparameters

The first thing to do, before attempting to do the first approaching steps, is to search for already solved implementations of the problem, then try to correct and/or improve them. Personally, I've used as structure code the one I've found in this article of towards data science(3), which I've found very good for starting although it's for the cartpole problem.

Then I've searched for other already solved implementations in the wiki section of the OpenAI Gym's GitHub(4) but no one had settings already working,they were too tied to their implementation.

At last I've added to the structure code the implementation of the techniques I wanted to test, using a model of DQN with two hidden layers with 24 and 48 neurons respectively, and tried reasonable configurations of the hyperparameters.

2.2.1 Hyperparameters explanation

Surely trying to find the best set of hyperparameters gave me a more detaliled view of how the training of DQN works, and not to underestimate certain implementations, which could completely change the performance of your net if they aren't thought along with the parameters.

The epsilon decay parameter, for example, has to be set watching carefully how often do you want your epsilon parameter to decrease. Initially I've underestimate this aspect and, as I was decreasing my epsilon once per episode, with a high value of epsilon decay (0.995), I found myself having an epsilon of 0.5 after nearly a thousand of episodes: that's too much exploration! In fact at every step it's selected a random number and if it's lower than epsilon the action will be taken randomly (exploration), otherwise will be selected by the DQN (exploitation); clearly it isn't a good behaviour if after hundreds of episodes you still pick your action randomly with a probability of 50%. Now I've reduced the epsilon decay so after about 30 episodes I reach the minimum epsilon (0.01) with a probability to select a random action of 1%

Another relevant improvement was given by changing the memory lenght of my buffer and also the batch size, which represents the amount of states and actions I use everytime for the experience replay. I've searched the internet to find a way to speed up the learning and I've found some suggestions redirecting to this article (1) where it's explained that is common to decay the learning during the training but it has the same effect to increase the batch size.Along with the increasement of the batch size, from 64 to 128 and then 256, I've also increased the memory length from a 100 thousands to 400 thousands.

After all of these changes all the techniques used were giving very good results, they were reaching the goal almost always and had reached a good average reward of -130 more or less, which means that the problem wasn't solved yet. The last change which brought to the solution was increasing the discount rate from 0.98 to 0.999; this parameter determines how important rewards in the distant future are compared to those in the immediate future. This has permitted the fixed Q targets and double DQN techniques to the average reward of -110, even surpassing it, reaching a result of -105. Unfortunately, the behaviour of simple DQN has worsened, mostly in terms of number of times the car can reach the goal, with this set of hyperparameters(Table 1) the best discount rate for simple DQN remains 0.98.

Table 1: Best set of hyperparameters

Name	Value
Learning rate	0.001
Discount rate	0.999
Epsilon	1
Minimum epsilon	0.01
Epsilon decay	0.85
Copy step	25
Batch size	256
Memory length	400k
τ	0.1

2.3 Soft update

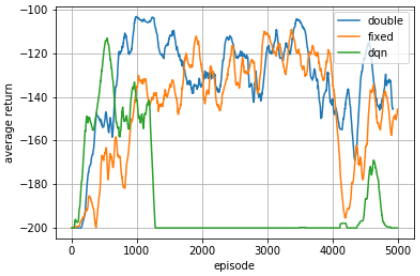
There are two main techniques to copy the weights from the training net to the target net: hard update and soft update. Hard update is as simple as I've already described before talking about fixed Q targets, soft update, as the name suggests, doesn't copy all the weights from the training network at once, but slowly in small steps following this formula:

$$w' = w\tau + w'(1 - \tau)$$

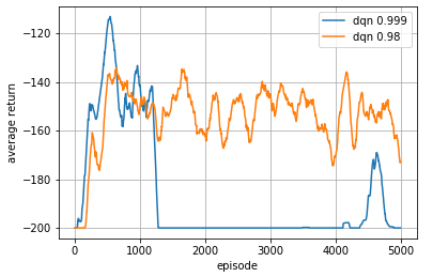
I've set τ to 0.1 so the weights w' of the target net will copy only 10% of the training net's weights at every copy step, moving slowly to their value. Personally I've found the substitution from hard update to soft update very performing.

3 Results

Below i will show the graphs with the running average reward over 100 consecutive episodes of the three techniques used in this project .



(a) Average reward of DQN, Fixed Q Targets and Double DQN



(b) Average reward of DQN with discount rate set to 0.999 and 0.98

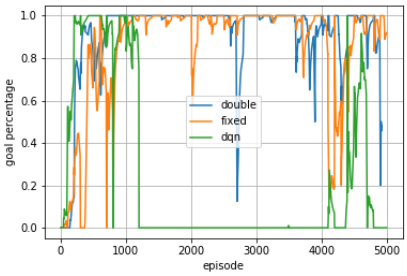
As you can see in figure (a) the first to solve the problem is the double DQN around the 1000th step, followed by the fixed Q targets around the 3000th step. The two of them could behave differently from run to run, for example during the tests has happened that the fixed Q targets reached the solution around the 1500th step, but the double DQN has always proven itself to be the first.

The simple DQN shown in figure (a) has usually two principals behaviours, it remains for the entire cycle with average around -190 or has a peak at the start then followed by a fast drop, it has never reached the solution but got close to it.

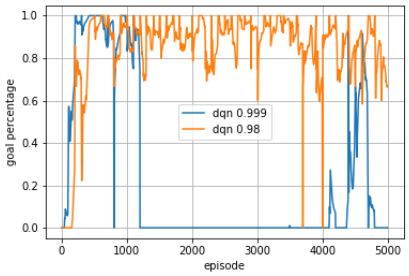
There is never convergence, at least in the 5000 episodes analysed.

In figure (b) two different behaviours of the simple DQN are shown, one with the discount rate set to 0.999 as in figure (a) and the other with that parameter set to 0.98. It's clear that with 0.98 it has a worse perform in terms of the best average reached, but it's more stable, and once it starts reaching the goal it never stops. It cannot be said the same for the discount rate set to 0.999.

In the next two images is shown the running percentage of goals achieved over 100 consecutive episodes.



(a) Goal percentage of DQN, Fixed Q Targets and Double DQN



(b) Goal percentage of DQN with discount rate set to 0.999 and 0.98

The one with more convergence in terms of percentage of goals reached is clearly the double DQN followed by the fixed Q targets. As anticipated, the DQN with decay factor set to 0.98 might have not the best performances in terms of average returns, but it has a very good behaviour in terms of percentages of goal achieved, almost always over the 80%.

4 Conclusion

The results of this project met my expectations and even exceeded them, due to the bad performance, at the beginning I started to think that it would take many more steps to reach the goal.

It is curious the behavior of the DQN with the discount rate set to 0.999, but that value has solved the problem with the other techniques, so I think there are other changes that must be made.

It is precisely to continue to try to improve the performance that I have included in the project the possibility of changing the parameters at runtime, as well as the possibility of launching the game by loading a net with weights chosen among the best saved during the training.

References

[1] Samuel L. Smith and Pieter-Jan Kindermans and Chris Ying and Quoc V. Le *Don't Decay the Learning Rate, Increase the Batch Size*, 2017

[2] OpenAI Gym: `https://gym.openai.com/`, searched on the 24th June 2020

[3] Structure code: `https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow-2-and-gym-8e105744b998/`, searched on the 24th June 2020

[4] MountainCar-v0 GitHub: `https://github.com/openai/gym/wiki/Leaderboard#mountaincar-v0`, searched on the 24th June 2020