DQN, DQN with Fixed Q-targets, and Double-DQN in MountainCar environment

Alessandro Franca
Department of Computer Science
University of Bologna
Bologna, BO 40131
alessandro.franca@studio.unibo.it

Abstract

The goal of this project is to compare, from a performance point of view, three different implementations of one of the most common reinforcement learning algorithms: Deep Q-Learning. In particular, DQN, DQN with Fixed Q-targets, and Double-DQN were chosen. The comparison was made using the same hyperparameters. The results showed that although all three implementations solved the environment under consideration, the two optimizations of the base algorithm led to better results.

1 Background

Q-learning [1] is an off-policy, TD control reinforcement learning algorithm, defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

The learned action-value function Q directly approximates the optimal action-value function. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

A Q-matrix with states as rows and actions as columns, is created in order to store each updated state-action pair. These methods are called *tabular methods*.

This approach works well for small environments, but as the number of states and actions increases, it is no longer a viable solution. One way to solve this problem is to make use of *function approximation methods* in order to approximate the Q-values.

1.1 Deep Q Learning

In Deep Q Learning, artificial neural network are used to approximate the Q value function. The network, called Deep Q Network (DQN) [2], receives the state of the environment as an input, and outputs a Q-value for each possible action. The maximum Q value determines which action the agent will perform. While training, the weights are updated accordingly to the *TD-Error*, which is the difference between the maximum possible value for the next state (Q-target) and the current prediction of the Q-value. As a result, Q-tables can be approximated using an artificial neural network.

1.1.1 Experience Replay

Experience Replay is a concept created to help the agent remember and improve. A buffer is maintained in memory, in which, at each instant t, the so-called *experience* is inserted: a quadruple that maintain the current state S_t, the chosen action A_t, the obtained reward R_{t+1}, and the state in which the agent would be at the next instant S_{t+1}.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

The Q-learning updates are performed by sampling a random batch of previous experiences from memory and feeding it to the network. In this way, the agent relives its past and improves, avoiding to chose the same action over and over again, due to the high correlation between some states.

1.1.2 Fixed Q-Targets

The original DQN implementation consists in a single neural network. This means that the same weights are used for estimating the Q-target and the Q-value. Therefore, at every step of training, when weight updates are made, the Q-values will update accordingly and so will the Q-targets, since the targets are calculated using the same weights. The output of the network is moved closer to the target, but also the target moves. This ends up in a highly oscillating training process. To solve this problem, a second neural network, called *target network*, is introduced [3]. This network will be used to estimate the Q-targets. This technique is called Fixed Q-Targets: since the target network's weights are fixed and are updated after τ steps.

1.1.3 Double DQN

DQNs could overestimate a Q-value for an action. In this scenario, that action will be chosen as the go-to action for the next step and the same overestimated value will be used as a target value. To address this problem a second DQN is introduced [4] and, while the first is responsible for the selection of the next action, the latter is responsible for the evaluation of that action. The *target* network is used to calculate the target Q-value of taking that action at the next state

By decoupling the action selection from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

2 The Environment

The agents will be compared on an OpenAI-Gym environment, using the best parameters found during tuning phase. At every step, an action is taken, and the environment returns the following informations:

observation an environment-specific object representing your observation of the environment. For example: car speed and car velocity.

reward reward achieved by the previous action.

done whether it's time to reset the environment again. If *True* indicates the episode has terminated. (For example, episode length greater than a threshold)

2.1 MountainCar-v0

The environment under consideration consists in a car 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. This environment corresponds to the MountainCar-v0 of `gym.openai.com` [5] MountainCar-v0 is considered "solved" when the agent obtains an average reward of at least -110.0 over 100 consecutive episodes [6].

2.1.1 Observations

Num	Observation	Min	Max
0	Car Position	-1.2	0.6
1	Car Velocity	-0.07	0.07

2.1.2 Actions

Num	Action
0	push left
1	no push
2	push right

2.1.3 Reward

Reward	Condition
0	the agent reached the flag (position = 0.5)
-1	the position of the agent is less than 0.5.

2.1.4 Starting State

	Value
Car Position	random value in [-0.6 , -0.4])
Car Velocity	0

2.1.5 Episode Termination

	Condition
1	The car position is more than 0.5
2	Episode length is greater than 200

3 Experiments

All experiments were performed on MountainCar-v0 environment. Three different agents were implemented, using three different learning algorithms. All agents were built with the same network architecture, and the same hyperparameters, so that comparisons could be made between them. In training, the average rewards of the last 100 episodes were calculated and, whenever an agent solved the environment, the model was saved. In testing, the parameters corresponding to the best solution were used.

3.1 Network Architecture

The network consists in two fully-connected hidden layers, and an output layer. All these layers are separated by Rectifier Linear Units (ReLu). The first layer accepts an input with dimension equal to the dimension of the state, and has 24 outputs, therefore, the second linear layer accepts 24 inputs and has 48 outputs. Finally, a fully-connected linear layer projects to the output of the network, i.e., the Q-values. The optimization employed to train the network is Adam.

3.2 Hyper-parameters

In all experiments, the discount was set to $\gamma = 0.999$, and the learning rate to $\alpha = 0.001$. The number of steps between target network updates was $\tau = 20$. Training is done over 5000 episodes. The size of the experience replay memory is 300K tuples. The memory gets sampled to update the network with batches of size 128. The exploration policy during training was ϵ-greedy with ϵ exponentially decreasing from from 1 to 0.01, with a decay rate of 0.99.

3.3 DQN Agent

The first agent implemented uses the Deep Q-learning algorithm in order to master the MountainCar environment.

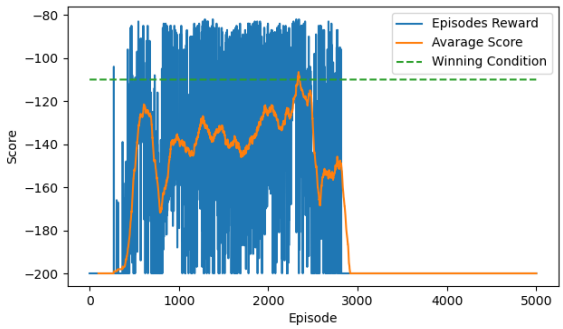

Figure 1: Training phase

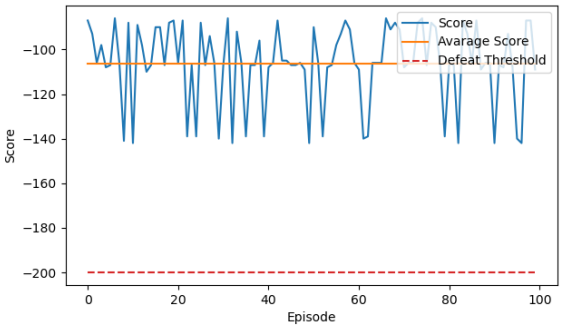

Figure 2: Test on 100 episodes using best weights found in training

The agent solved the environment in less than 2.5k episodes, before having a significant drop down. In testing phase, the agent won 100 consecutive games, with -106.9 as average score.

3.4 Fixed Q-Targets DQN Agent

The second agent implemented uses the Deep Q-learning algorithm with Fixed Q-Targets in order to master the MountainCar environment.

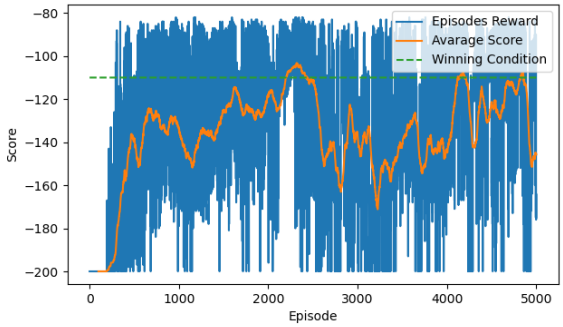

Figure 3: Training phase

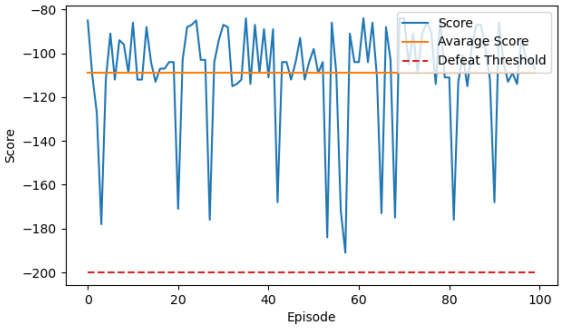

Figure 4: Test on 100 episodes using best weights found in training

The agent solved the environment in less than 2.5k episodes. In testing phase, the agent won 100 consecutive games, with -109.6 as average score.

3.5 Double DQN Agent

The third agent implemented uses the Deep Q-learning algorithm with Double DQN in order to master the MountainCar environment.

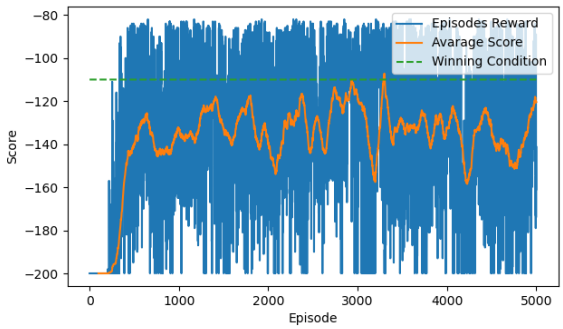

Figure 5: Training phase

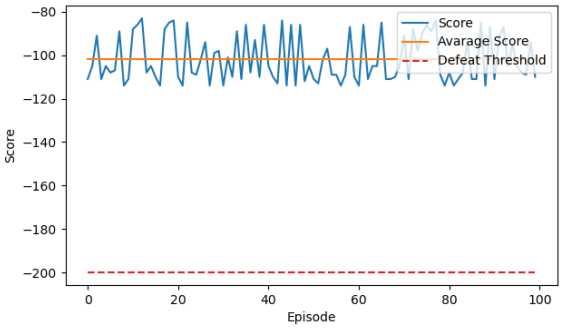

Figure 6: Test on 100 episodes using best weights found in training

The agent solved the environment in about 3.2k episodes. In testing phase it wins 100 consecutive games, with a pretty stable average score of -101.1 .

3.6 Results

All the agents managed to solve the environment in about 3k episodes. As shown in Figure 7, the DQN Agent has a significant drop down before episode 3000, while FQTDQN Agent, and DDQN Agent keep winning the game. The second agent shows a little drop down from episode 2500 to 3200, while the third shows the most stable overall trend.

The parameter that proved to be crucial for the resolution of the environment was the ϵ decay rate. Other tests were carried out varying this parameter: by decreasing it, the Fixed Q-Targets and Double-DQN agents were able to solve the environment in a considerably reduced amount of episodes compared to the tests taken into consideration. The basic DQN, however, failed to resolve the environment. Therefore, it was decided not to report the results of these tests, since the basic assumption was to bring a comparison only on resolved environments.

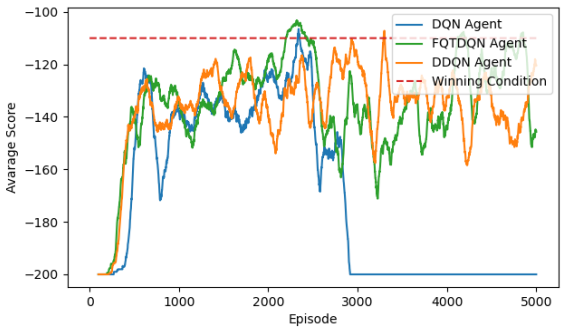

Figure 7: Training comparison between all the agents

4 Conclusion

The objective of this project was to to compare three different agents on the same environment. The differences in terms of performance have been highlighted, coming to the conclusion that, even if the environment has been solved by all agents, the two implementations using Fixed Q-Targets and Double DQN are much more performing than the simple one. These two, in fact, solve the problems introduced in the basic implementation. The Fixed Q-Targets implementation proved to be the least stable in testing, while the third gave excellent results.

References

[1] Richard S. Sutton & Andrew G. Barto (2018) Reinforcement Learning: An Introduction.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller - Playing Atari with Deep Reinforcement Learning. https://arxiv.org/abs/1312.5602

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves , Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg1 Demis Hassabis - Human-level control through deep reinforcement learning. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[4] Hado van Hasselt, Arthur Guez, David Silver - Deep Reinforcement Learning with Double Q-learning. https://arxiv.org/abs/1509.06461

[5] Gym.OpenAI - MountainCar-v0 https://gym.openai.com/envs/MountainCar-v0/

[6] Gym.OpenAI - MountainCar-v0 wiki. https://github.com/openai/gym/wiki/MountainCar-v0