Continuous Control with Deep Deterministic Policy Gradient (DDPG)

Description

As described in the README.md, the current project solves a continuous control task, the Unity Reacher environment. In order to solve this problem, I have used the Deep Deterministic Policy Gradient (DDPG) algorithm [1].

The code is prepared to run it either with 1 agent or 20 agents, as the mode in which the experience tuples are collected is different in both cases. Furthermore, it has been coded one of the variants of Prioritized Experience Replay (PER)[4] and one of the parts of the well known TD3 [5] algorithm, which could be seen as an evolved DDPG.

The project is structured in 4 scripts:

- **Main.py.** It is the main script from which we would select the hyperparameters and we will run the code.
- **Models.py**. Implements the Actor-Critic neural network architectures.
- **Ddpg_agent.py**. Provides all the necessary code to execute an agent following the DDPG approach.
- **Utils.py**. Implements utilities that may be used by the agent, such as replay buffers and normal noise generator to get actions.

A lot of experiments could be done with all the available hyperparameters. For this particular project, the provided results of both the scores and the weights (of the models) are based on the next experiments:

- Experiment 1. Reacher 1 agent, batch size = 64
- Experiment 2. Reacher 20 agents, batch size = 64
- Experiment 3. Reacher 1 agent, batch size = 256
- Experiment 4. Reacher 20 agents, batch size = 256

Learning algorithm

As mentioned above, I have implemented the DDPG approach to solve this environment, which is a model-free, off-policy actor-critic algorithm.

Why DDPG?

- *It is not possible to apply Q-learning to continuous action spaces because each output/action is not discrete and this would require either optimise each action selection in each step (a very slow process) or to discretize the possible action values (losing information in that process).*
- *Deterministic Policy Gradients (DPG) [2] shows very good results outperforming stochastic counterparts in high-dimensional action spaces. It uses an off-policy actor-critic algorithm to learn a deterministic target policy from an exploratory behaviour policy. However, it suffers from some instabilities during the learning process.*

The key of DDPG resides in the fact that combines the actor-critc solutions provided at DPG and the success approaches used in DQN [3] that have good stability properties. Thus, DDPG makes modifications to the DPG algorithm using approaches used in DQN in order to allow neural networks to learn in large state and action spaces online. To address the instabilities during the learning process, DDPG applies:

- **Replay Buffer**. It is assumed that the samples are independent one of each other and equally distributed. However when sample experiences are obtained from exploring sequentially in the environment, this is not held. This is possible because DDPG is an off-policy algorithm.
- **Target networks and soft updates.** When using the same network that we are going to update to compute estimates used to make calculations in that update, we generate a high correlation and that could lead to divergence in the learning process. Thus, a solution is to use target networks.
    - At DDPG instead of copying the values of the local network into the target after C steps (as explained in [3]), the target network weights are updated every time the local network is updated too. This is done using the weights of both local and target networks and only updating a minor part of the target network, yielding in a slow update. The way in which how much the target network is updated is parametrized by a tau value, which has to be a positive close to zero value (i.e: 0.001).

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

Referring to the exploration, continuous action spaces such the one we are going to solve are challenging. We could not use a policy to explore like in DQN (i.e. e-greedy) as the action selection is not discrete. Thus, an alternative is to obtain the continuous possible values from the actor directly, and some noise to them (i.e. Ornstein-Uhlenbeck, Gaussian):

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

The advantage of using an off-policy algorithm is that the exploration part could be treated independently from the learning and that allows us to store the experience tuples in a experience buffer to use them for training. If we used an on-policy algorithm, we would have to generate experience tuples every time we want to train our agent.

How are the networks updated?
The network parameters are updated in different ways depending on the network.

In the case of the critic, it calculates the target (based on the action specified by the actor) and updates its parameters based on the expected value.

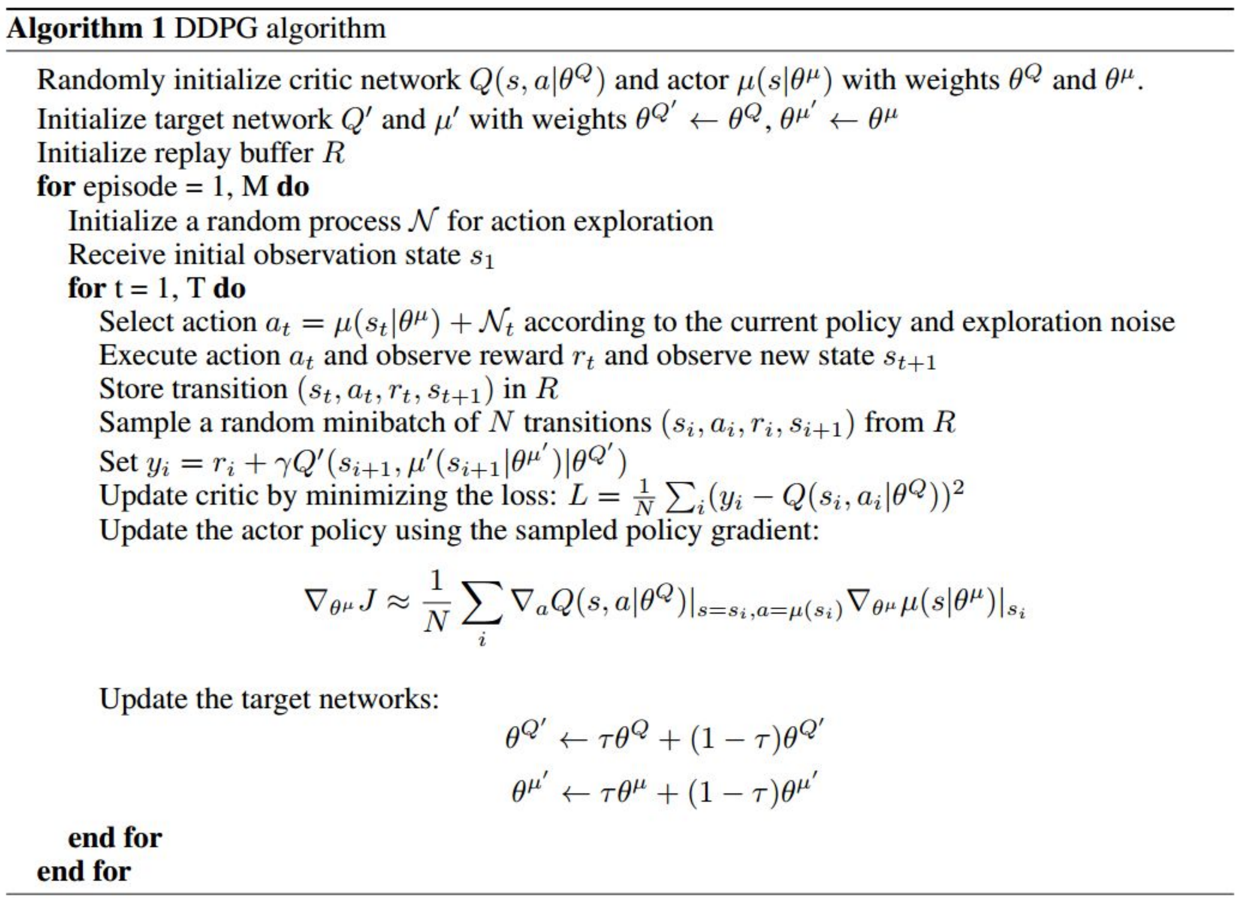
$$\delta_t = r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1})) - Q^w(s_t, a_t)$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t)$$

In the case of the actor, the policy is moved in the direction of the gradient of Q (rather than globally maximising Q). Specifically, for each visited state s, the policy parameters are updated in proportion to the gradient, and as each state might suggest different directions, the gradients should be averaged taking into account each occurrence(state) probability.

$$\theta^{k+1} = \theta^k + \alpha \mathbb{E}_{s\sim\rho^{\mu^k}}\left[\nabla_\theta \mu_\theta(s)\ \nabla_a Q^{\mu^k}(s,a)\Big|_{a=\mu_\theta(s)}\right]$$

All these information could be summarized in the next pseudocode:



Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
end for
```

Hyperparameters

I have followed the guidelines provided in the experimentation of [1], although I have changed some parameters as I have realized that I got better results.

parameter	value
GAMMA	0.99
TAU	0.001
BUFFER_SIZE	1,000,000
BATCH_SIZE	{64,256}
LR_ACTOR	0.001
LR_CRITIC	0.001

Moreover, other hyperparms have been let to be parametrized by the user if decides to use either the PER or the policy/targets delayed updates:

parameter	value
BUFFER_TYPE	{'replay','prioritized'}
alpha, beta	0.6, 0.9
POLICY_UPDATE	{1 - 4}

In the case of using PER, it is needed to specify the alpha and beta parameters. They are used to set how much prioritization is used to select samples inside the probability distribution, and the degree of using importance weights respectively.

Neural Network

Our model architecture is composed of two neural networks: a policy-based network with multiple action continuous spaces (actor); and a value-based network with a single output that evaluates how good the policy-based networks outputs are through estimation. This is considered an actor-critic architecture.

On one hand we have the actor with 4 continuous actions that might throw values between -1 and 1. Thus, a Hyperbolic Tangent (tanh) function fits perfectly for our purpose.

```
DDPG_Actor(
  (input_layer): Linear(in_features=33, out_features=128, bias=True)
  (hidden_layers): ModuleList(
    (0): Linear(in_features=128, out_features=128, bias=True)
  )
  (output_layer): Linear(in_features=128, out_features=4, bias=True)
)
```

On the other side, the critic architecture is a little bit different, as in the first hidden layer we process the output of the input layer (128 neurons) alongside the 4 possible actions in order to pass it through the reminder network layers.

```
DDPG_Critic(
  (input_layer): Linear(in_features=33, out_features=128, bias=True)
  (hidden_layers): ModuleList(
    (0): Linear(in_features=132, out_features=128, bias=True)
  )
  (output_layer): Linear(in_features=128, out_features=1, bias=True)
)
```

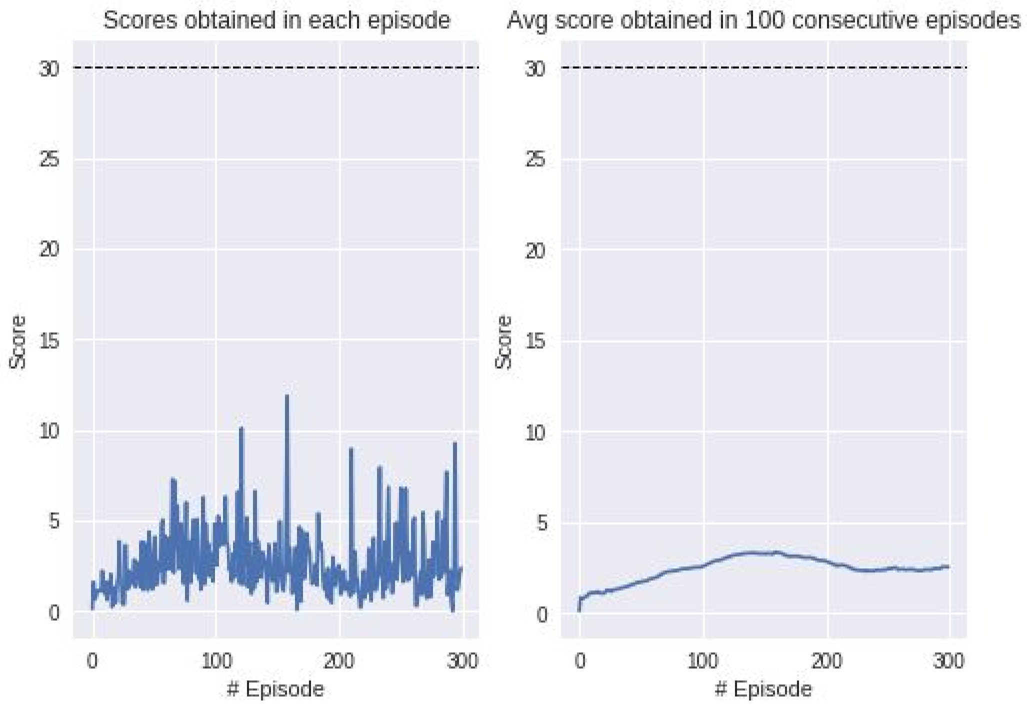
Plot of rewards

The graphs shown in this section could be generated with the script provided in the /score folder in this repository. For that purpose, it is needed a pickle with the scores (list) obtained in each episode and the checkpoint (integer) in which the algorithm has realized it has solved the environment.

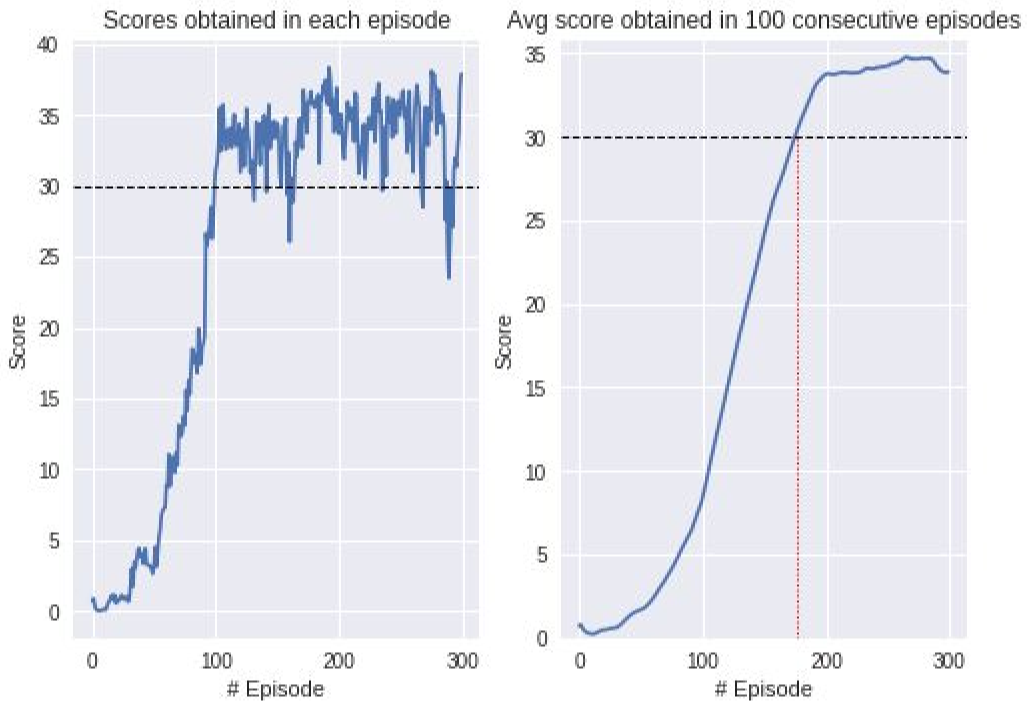
In this project we have tried to solve the environment for 1 and 20 agents. Thus, in the latter case, the reward obtained in each trajectory/episode represents the average score obtained by all the agents in a full episode.

Next, they are attached the results obtained for the experiments described in the description. *The needed score to solve the environment has been plotted with a black horizontal line; in the case that the environment it is solved, a vertical red line is plotted in the episode in which it has been achieved the milestone.*

Experiment 1

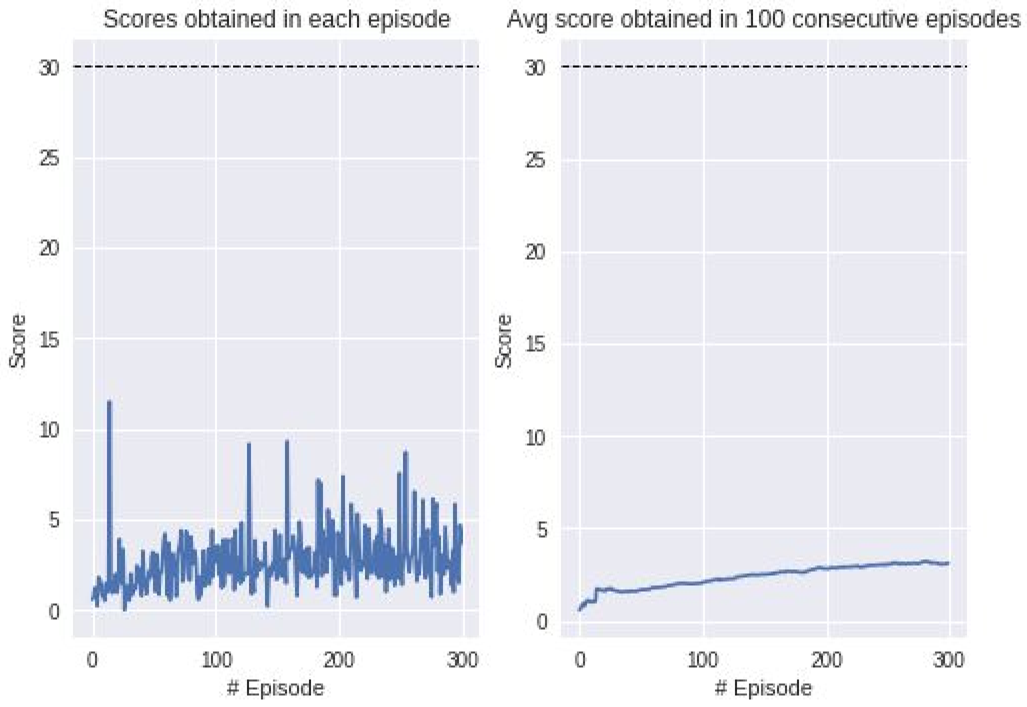


Experiment 2

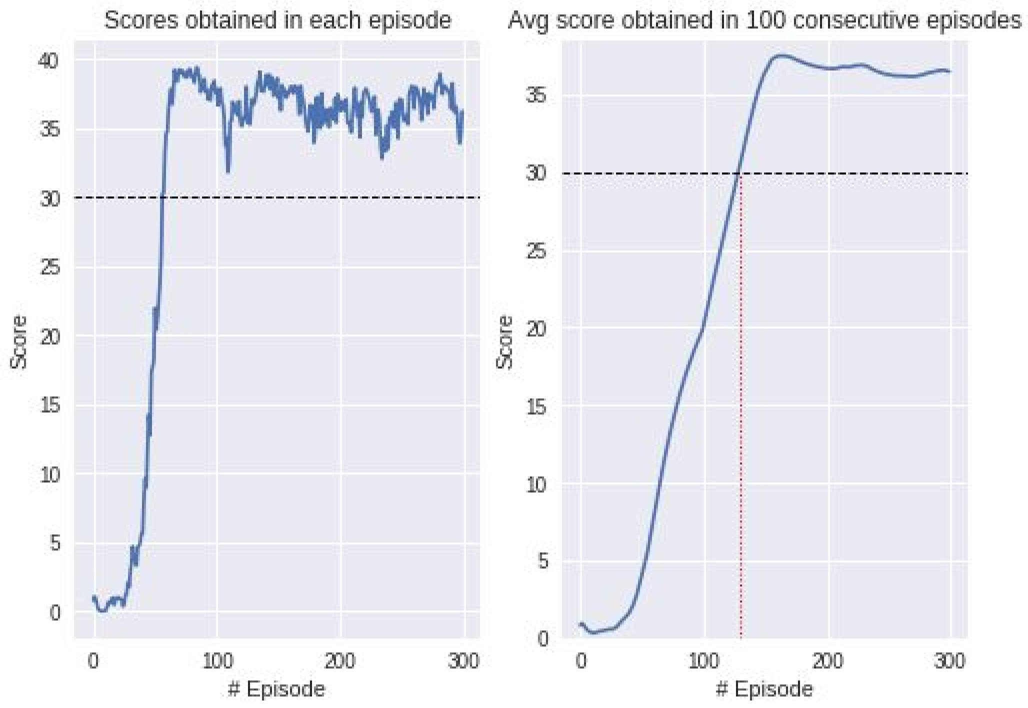


The environment is solved after 176 episodes of training.

Experiment 3

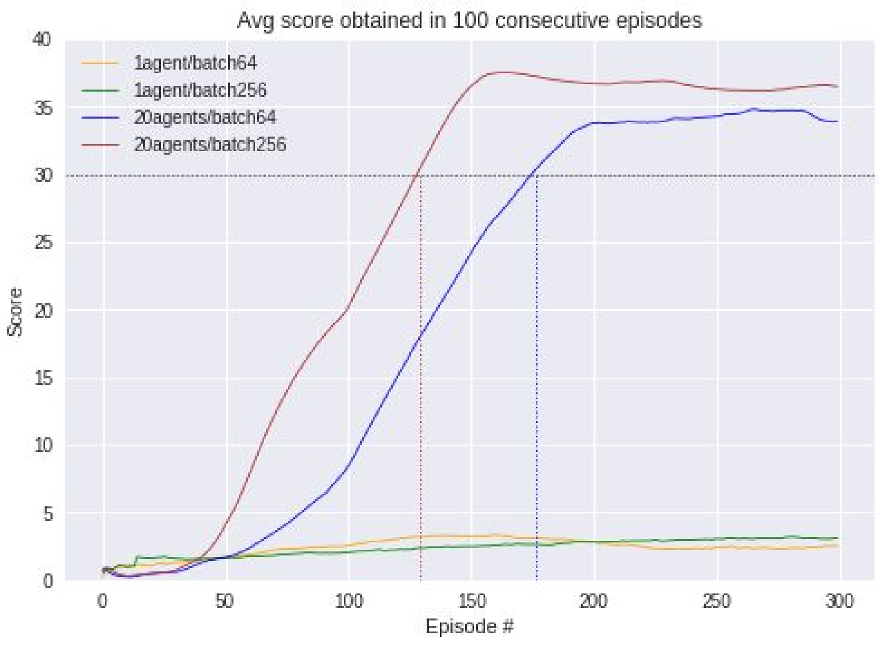


Experiment 4



The environment is solved after 129 episodes of training.

Finally, all these values could be summarized in the next graph:



Conclusions

We were able to solve the environment with the DDPG approach and using the 20 agents environment!

From the obtained results it could be seen that the batch size accelerates the learning process.

The learning rates have been seen (empirically) to be very sensitive; I set a value of 1e-3 instead of 1e-2 as used in [1] because the learning process became very unstable. Moreover, using an input and hidden layers of 64 units was insufficient to achieve the expected learning.

Although using the same hyperparms, there is a huge difference between using the 1 or 20 agents environment. I have been able to solve only the 20 agents environment. The 1 agent environment shows that it also learns (but much more slowly). This might well be because the rewards are very sparse, and in the case of 20 agents, the probability of having good/representative samples (with reward information different than 0) for training is higher than with only 1 agent.

Ideas for future work

In the future it would be interesting to implement other suitable algorithms that have shown good results in other environments, such as Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO), A3C/A2C and TD3.

Some hyperparameters have been shown to be sensitive. Thus, the idea would be to play a little with them. I would suggest playing with the learning rate values, the tau, and the number of layers alongside the number of neurons of each layer.

It would be interesting to see whether Prioritized Experience Replay may improve the results too, as it would collect and maintain the most remarkable experience tuples in the buffer used to train more efficiently the critic network, which could lead to a better learning process in the actor.
The code provides a PER implementation. Thus, I encourage anyone to play with all the hyperparms to try to beat the provided scores.

Acknowledgement

The developed code is based on the ideas and code examples provided in the book Grokking Deep Reinforcement Learning (good book with clear explanations, recommendable for any RL passionate).

Bibliography

[1] Continuous control with deep reinforcement learning
[2] Deterministic Policy Gradient Algorithms
[3] Human-level control through deep reinforcement learning
[4] Prioritized Experience Replay
[5] Addressing Function Approximation Error in Actor-Critic Methods