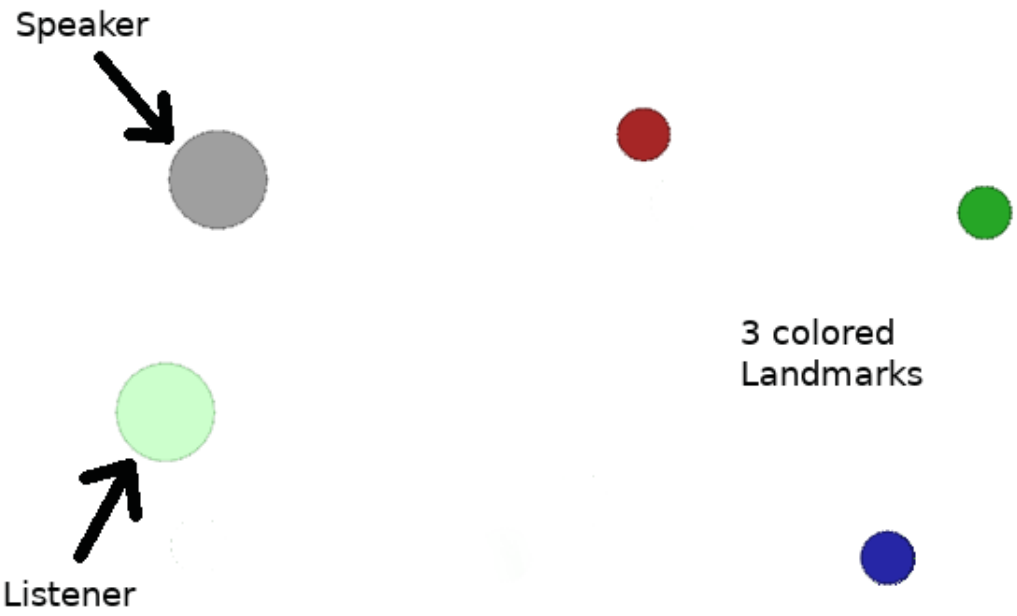
KAIST

Fall Semester
December 2019

Second Paper

Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments



Student : David Albert
Professor : Jinwoo Shin

Contents

Introduction

In this second project, we explore methods relative to deep reinforcement learning for multi-agent domains. More especially, we will be focused on the paper *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments* [1] published to the conference NIPS in 2017.

Reinforcement learning (RL) is an active research field. In the lasts few years, many papers in RL led to results that convinced people that it can be applied to solve challenging problems, from game playing to robotics. Most of the successes of RL have been in single agent domains, where predicting the behaviour of other actors in the environment is unnecessary.

However, there are number of important applications that involve interaction between multiple agents. For example, multi-robot control, the analysis of social dilemnas or multiplayer games such as football requires mixed cooperative and competitive behaviour. Also, successfully scaling RL to environments with multiple agents is crucial to building agents that can interact helpfully with humans and cooperate with them to improve performances and/or productivity in several tasks.

Unfortunately, learning in multi-agent setting is challenging and traditional reinforcement learning approaches such as Q-learning or policy gradient are not designed for this. In fact, one main issue when learning in multi-agent setting is that the environment becomes non-stationary from the perspective of each agent due to the fact that each agent's policy is changing during training process. As the consequence, one of the main challenges is to deal with unstability.

In this report, we will first explain how the choices made by the authors improve stability and face the problem of non-stationarity of the environment in this context. Then, we will introduce our implementation that we wish to be as generic as possible and whose goal is to provide a reusable API for diverse multi-agent reinforcement learning tasks. This includes that the API can be used for any multi-agent environments and several types of agents training methods. Finally, we will put forward some experiments made on some environments from multi-agent particle environments repository [2].

Part I

Paper Presentation

In this first part, we present the background considered by this paper, we remind the problem and their difficulties and explain the methods used to tackle these difficulties.

1 Background

Before being able to solve a reinforcement learning problem for multi-agents systems, we need to define a generalization of Markov decision processes which is called Markov games and present state-of-the-art algorithms to solve RL problem in single agent case. The algorithm introduced in the paper is strongly related to these algorithms.

1.1 Partially Observable Markov Games

Markov decision processes (MDPs) and partially observable Markov decision precesses (POMDPs) are particularly interesting mathematical frameworks which describe a single agent progressing in an environment. A common extension of the framework for N agents are Markov games (also called stochastic games). A markov game for N agents is described by a state space $\mathcal{S}$ describing the environment, a set of action spaces $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_N\}$ and a set of observation spaces $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_N\}$ for each agent. At each time step, each agent choose an action according to its own policy given by:

$$\pi_{\theta_i} : \mathcal{O}_i \times \mathcal{A}_i \to [0, 1] \qquad (i.e.\ \ \pi_{\theta_i}(a|o) = \mathbb{P}\{A_{i,t} = a | O_{i,t} = o\})$$

and thus the environment changes and produce the next state according to the transition function:

$$\mathcal{T} : \mathcal{S} \times \mathcal{A}_1, ..., \mathcal{A}_N \to \mathcal{S} \qquad (i.e.\ \ p(s'|s, a_1, ..., a_N) = \mathbb{P}\{S_{t+1} = s' | S_t = s, A_{1,t} = a_1, ..., A_{N,t} = a_N\})$$

After every state transition, each agent i obtains rewards r_i and receives a private observation correlated with the state $o_i : \mathcal{S} \to \mathcal{O}_i$. The initial state is determine by a distribution ρ and the goal for each agent is define by maximizing its total expected return $G_i = \sum_{t=0}^{T} \gamma^t r_i^{(t)}$ where γ is a discount factor and T is the time horizon. If there is at least one agent i such that o_i is not injective then we say that the system is a partially observable Markov game.

1.2 Popular methods in RL

There are two popular approaches to solve RL problem. The first approach models the value function $q_\pi(s, a) = \mathbb{E}_\pi(G_t \,|S_t = s, A_t = a)$ and learns to estimate $q_*(s, a)$. The second approach learns the policy π directly. Both have their advantages and drawbacks.

Value-based methods → Q-learning and DQN

In value-based methods, the main purpose is to estimate the value function v_* or q_* using generated samples from the random variable S_t, A_t, R_t, S_{t+1}. One way to do this is by iteratively updating the estimation Q of q_* as follow:

$$Q(S_t, A_t) = (1 - \alpha_t)Q(S_t, A_t) + \alpha_t \left(R_t + \max_{a'} Q\left(S_{t+1}, a'\right)\right) \tag{I.1}$$

In fact, since we have

$$q_\pi\left(S_t, A_t\right) = \mathbb{E}_\pi\left\{G_t | S_t, A_t\right\} \approx \frac{1}{N}\sum_{i=1}^{N} G_{t_i} \tag{I.2}$$

We can iteratively estimate $(I.2)$ using moving average (MA) process:

$$Q\left(S_t, A_t\right) = \left(1-\alpha_t\right) Q\left(S_t, A_t\right) + \alpha_t G_t \tag{I.3}$$

Finally, by combining Bellman optimality equation given by $q_*\left(s,a\right) = \mathbb{E}\left\{\right\}$ and one-step TD-learning applied to action-value function, i.e. $Q\left(S_t, A_t\right) = \left(1-\alpha_t\right) Q\left(S_t, A_t\right) + \alpha_t R_t + Q(S_{t+1}, A_{t+1})$ we can iteratively approximate q_* using equation $(I.1)$.

If the state space $\mathcal{S}$ is continue, we can use a neural network to approximate Q. Doing this is usually called Deep Q-learning and lead to the DQN algorithm. DQN learns q_* corresponding to the optimal policy by minimizing the loss:

$$\mathcal{L}\left(\theta\right) = \mathbb{E}_{s,a,r,s'}\left\{\left(Q_*(s,a|\theta) - y\right)^2\right\} \quad where \quad y = r + \gamma \max_{a'} \bar{Q}_*(s', a')$$

$\bar{Q}$ is a target Q function. That is to say it is a copy of Q whose parameters are periodically updated with parameters of Q. DQN also uses experience replay buffer $\mathcal{D}$ to allow data efficiency. Both tricks also help stabilizing learning.

Multi-agent case: We can directly apply Q-learning to multi-agent setting where each agent i learn independently optimal action-value function Q_i. However, in this case the environment appears non-stationnary from the view of each agent because the policies of other agents change over time. As a consequence, the Markov property is not respected and there is no guarantee for convergence of Q-learning. Moreover, experience replay buffer cannot be used because $p\left(s'|s,a,\pi_1,...,\pi_N\right) \neq p\left(s'|s,a,\pi'_1,...,\pi'_N\right)$ when $\exists\ i;\ \pi_i \neq \pi'_i$.

Policy-based methods $\rightarrow$ Policy Gradient (PG) Algorithms

Policy-based methods are composed of every methods attempted to directly approximate the optimal policy π. The advantage is that doing this we can learn stochastic policies (this is not possible with value-based methods). Moreover, stochastic policies are usually better than a deterministic one when the environment is partially observable. The drawback is that these methods often suffer from high variance.

Most of the policy-based algorithms learn an approximation π_θ of the optimal policy by adjusting the parameters θ in order to maximize the objective $J\left(\theta\right) = \mathbb{E}_{\pi_\theta}\left\{G\right\}$ where G is the total expected return (i.e. G_0). To this end, the policy gradients theorem provides a convenient formulation $(I.4)$ for the gradient of $J(\theta)$ using the action-value function.

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\pi,\ a\sim\pi_\theta}\left\{q_\pi(s,a)\ \nabla_\theta log\ \pi_\theta(a|s)\right\} \tag{I.4}$$

Algorithms based on $(I.4)$ are named policy gradient algorithms (PG) and are differentiate by the way they approximate the action-value function. One could simply use a sample return $G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ (**REINFORCE** algorithm) or learn an approximation of the true action-value function by e.g. temporal-difference learning. The second method is called **Actor-Critic** algorithm (q_π is called the critic whereas π_θ is the actor).

It is also possible to extend the policy gradient framework to deterministic policies $\mu_\theta : \mathcal{S} \rightarrow \mathcal{A}$. The method in this case is called **Deterministic Policy Gradient (DPG)** and under some conditions we can write the gradient of the objective $J(\theta)$ as :

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\mu}\left\{\nabla_\theta \mu_\theta(a|s)\ \nabla_a q_\mu(s,a)|_{a=\mu_\theta(s)}\right\} \tag{I.5}$$

Note that it involve that the action space $\mathcal{A}$ must be continuous since the theorem relies on $\nabla_a q_\mu(s,a)$. **DDPG** algorithm is a variant of DPG where μ and q_μ are approximated with neural networks and which use some of the DQN algorithm tricks (e.g. target network and replay buffer of experience).

Multi-agent case: In multi-agent case, PG algorithms have even higer variance. Futhermore, traditional PG algorithms use a baseline to ameliorate high variance but it is not possible in multi-agent settings due to non-stationnarity issues.

2 Method

Now that we reviewed traditional deep reinforcement learning methods and the issues involved when applying to multi-agent settings, we will describe the choices made by the authors to cope these issues.

2.1 Multi-Agent Decentralized Actor and Centralized Critic

In multi-agent reinforcement learning problems it is important that agents operate independently, in a decentralized way. That is to say, we don't want to allow each agent to have access to a centralized database containing informations on the environment but we prefer that it uses only its own local information. Learning in this setting is less simple because we have very few information about the environment and the other agents.

To this end, the authors adopted the framework of centralized training with decentralized execution. More concretely, consider a game with N agents with policies parameterized by $\theta = \{\theta_1, ..., \theta_N\}$, and let $\pi = \{\pi_1, ..., \pi_N\}$ be the set of all agent policies. Then, the gradient of the expected return for agent i is given by:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^\pi,\ a_i\sim\pi_{\theta_i}} \{\nabla_{\theta_i} log\ \pi_{\theta_i}(a_i|o_i)\ q_i^\pi(x, a_1, ..., a_N)\} \tag{I.6}$$

where

- x are some state information (typically $x = (o_1, ..., o_N)$)
- $q_i^\pi(x, a_1, ..., a_N)$ is a centralized action-value function for agent i

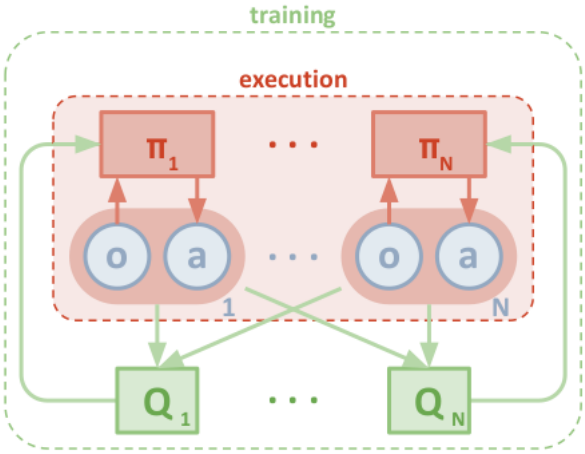


Figure I.1 – Multi-agent decentralized actor, centralized critic approach

Similar to DDPG algorithm which can be viewed as the deterministic case of the actor-critic algorithm. We can extend equation $(I.6)$ to deterministic policies and continuous action spaces $\mathcal{A}$. The authors called this method **MADDPG** and the algorithm is given below in Figure I.2. The gradient is rewritten as:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{x,\ a\sim\mathcal{D}} \left\{\nabla_{\theta_i}\mu_{\theta_i}(a_i|o_i)\ \nabla_{a_i} q_i^\mu(x, a_1, ..., a_N)|_{a_i=\mu_{\theta_i}(o_i)}\right\} \tag{I.7}$$

where

- $\mathcal{D}$ is the experience buffer and contains the tuples $(x, x', a_1, ..., a_N, r_1, ..., r_N)$
- q_i^μ is the centralized action-value function and is updated using a way similar to the DQN algorithm

The main motivation behind MADDPG is that if we know the actions taken by all agents then the environment become stationnary even if the policies are changing, i.e. $p(s'|s, a_1, ..., a_N, \pi_1, ..., \pi_N) = p(s'|s, a_1, ..., a_N) = p(s'|s, a_1, ..., a_N, \pi'_1, ..., \pi'_N)$ for any $\pi_i \neq \pi'_i$.

Thus the authors tried to reduce the non-stationnarity issue by using multi-agent actor-critic framework with a centralized training and deterministic policies.

2.2 Improving statibility using Policy Ensembles

The authors also proposed to train a collection of K different sub-policies for each agent in order to obtain multi-agent policies that are more robust to changes in the policy of competing agents. Thus, at each episode, a particular sub-policy is chosen randomly for each agent and is use until the next episode.

I didn't have time to try to use policy ensembles which requires more programming complexity and ressources but I think this is a key idea when training competitive agents.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Figure I.2 – MADDPG Algorithm

Part II

Implementation details

In this project, we provide an high-level API allowing to train simultaneously N agents. The API also allows to consider an environment with non trainable agents. That is to say we can include external agents to our system whose policies won't change when we call *learn* function (e.g. humans actions or any hand-written policy). In this report, we won't deal with non-trainable agents but only trainable ones. There are several types of trainable agents, each corresponds to a learning methods (e.g. DQN, Actor-Critic, DDPG, MADDPG). We have implemented neither tested all the methods which are numerous but we mainly stayed focus on the MADDPG algorithm which is the main topic.

Below, an exemple of how to easily train two agents (one with DQN method, the other with MADDPG).

```
from marl.model import MlpNet, GumbelMlpNet
from marl.agent import DQNAgent, MADDPGAgent
from marl import MARL

# Declare your Gym environment
env = make_env("env_name")

obs_space_0 = env.observation_space[0]
act_space_0 = env.action_space[0]
obs_space_1 = env.observation_space[1]
act_space_1 = env.action_space[1]

# Decalre an agent trained following DQN algorithm
first_agent = DQNAgent( model='MlpNet', observation_space=obs_space_0,
                        action_space=act_space_0, name="DQN_Agent")

# Declare an agent trained following MADDPG algorithm
n_inputs_critic =  obs_space_0.shape[0] + obs_space_1.shape[0] + act_space_0.n
                + act_space_1.n

critic = MlpNet(n_inputs_critic,1)

ag2 = MADDPGAgent(critic_model=critic, actor_model='GumbelMlpNet',
                    observation_space=obs_space_1, action_space=act_space_1,
                    index=1, name="MADDPG_Agent")

# Declare the multi-agent system ()
marl_sys = MARL(agents_list=[ag1, ag2], name='Two-Agent-RL')

# Multi-Agent algorithms as MADDPG usually need to have access to ohter agents attributed
# --> Centralized Training
ag2.set_mas(marl_sys)

# Train simultaneously all agents for 25 000 timesteps
marl_sys.learn(env, nb_timesteps=25000)
```

1 Trainable agents

A trainable agent is composed of the following attributes:

- a **name**
- a **policy** based on :
  - a **model** (e.g. neural network model, table, etc)
  - an **observation space** (Gym like space, e.g. Discrete, Box, MultiDiscrete,..)
  - an **action space** (Gym like space, e.g. Discrete, Box, MultiDiscrete,..)
- an **experience memory** (e.g. buffer of experience replay)
- an **exploration process** (e.g. greedy, ϵ-greedy, Ornstein–Uhlenbeck process)
- some **training parameters**:
  - a **learning rate**, a **discount factor** γ, a **batch size**

The package is done such that we can easily declare each component (models, experiences, exploration processes, policies and agents). It follows some tricks of Gym repository [3] to being able to easily and adequatly declare objects. An example to declare an exploration process (ϵ-greedy process for instance) is shown below.

```
import marl
import marl.exploration

# See all available exploration processes
print(marl.exploration.available())

# Create an exploration process from an available process
exploration = marl.exploration.make("EpsGreedy-cst01")

# Create an exploration process using additional parameters
exploration = marl.exploration.make("EpsGreedy", eps_deb=1.0, eps_fin=0.05,
                                                 deb_expl=0.2, fin_expl=0.9)
# Using the same way we can easily declare other components
# --> name_type include in {experience, agent, policy, exploration, model}
my_type_obj = marl.{name_type}.make("available_class", additional_params)
```

Part III

Experiments

The experiments where made using the environments described in [2]. More especially in two of the available environments which are the "Simple" environment and the "Speaker-Listener" environment. These environments are described below. They are adapted for algorithms with continuous action space. This kind of setting implies some difficulties. For instance, the authors did not explain how they faced the question of exploration. It is obvious that exploration stay a very important question, even more when dealing with several agents with a continuous action space. Another particularly recurrent problem was that the gradient was very unstable.

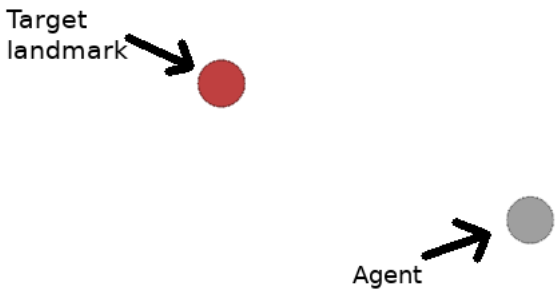
1 Environments description

Because training several agents with continuous action space is not easy and is very sensitive to a lot of parameters, I choose to stay focused on two of the simplest environments provided. One typical situation in the *multi-agent particles* environments is the case of agents who need to reach a landmark. In this case the agent's observation can be his own location, the location of the other agents and the location of the landmarks. As for the actions it typically consists of the amount of velocity in each of the four directions (typically between near to 0 means low velocity and near to 1 means high velocity).

1.1 *"Simple"* environment

This environment is, as is name, a simple environment with only one agent. The agent start at a random position in the environment and have to reach a landmark which is initially positionned randomly on the map. As explain in their repository [2], the agent sees landmark position and is rewarded based on how close it gets to landmark.

If we consider MADDPG algorithm in this single agent's setting, the algorithm is nothing but the DDPG algorithm. This is a good starting environment for understanding challenges of learning continuous action space policies.



1.2 *"Speaker-Listener"* environment

This second environment is more complex and involves two agents and three landmarks. One agent is the *speaker* that does not move (observes the goal of other agent), and other agent is the *listener* (cannot speak, but must navigate to correct landmark). The correct landmark is the landmark with the same color that the agent's color.

In the above exemple, the listener must reach the green landmark but he cannot know it by himself. A good strategy would be for the speaker to output his input and for the listener to learn to reach the landmark of the same color than the speaker output.

2 Important tricks for training

2.1 Ornstein–Uhlenbeck process

For the exploration, we used the Ornstein–Uhlenbeck process. The process is a stationary Gauss–Markov process, which means that it is both a Gaussian and a Markov process. It was originally applied in physics as a model for the velocity of a massive Brownian particle. We thus can easily understand that it is an adequate choice in order to model exploration in our case. This process is used in the single agent algorithm DDPG.

Formally, the Ornstein–Uhlenbeck process x_t with additional drift term μ is defined by the following stochastic differential equation:

$$dx_t = \theta \times (\mu - x_t)\, dt + \sigma dW_t$$

Where W_t is a Wiener process, θ and σ are stricly positive parameters. In our algorithm we will approximate this process by the following formulation:

$$x_{t+1} - x_t = \theta \times (\mu - x_t) + \sigma W_t$$

With $W_t \sim \mathcal{U}_{[0,1]}$.

2.2 Gradient clipping

In order to avoid that the gradient become too large and make too big changes in the policy, we applied another important trick which is to clip the gradient between two values. This can be essential to use gradient clipping especially in a competitive setting because when the policy changes with to wide amount, it results to a learning unstable and so it is not tractable.

3 Results

Even with the two previous tricks and using scenario with only few agents it was difficult to get good results due to the continuous action space. In fact, the continuous action space lead to difficulties for controlling action. One consequence that we noticed, the agent will usually take actions with any risk to get large negative reward. The consequence is that it is a better choice, in term of reward, to stay at the initial position than trying to move. The reason for this is that stop moving is not easy in these kind of environment and so it is likely that the agent loose more by trying something than without. To face this difficulty, in addition to the continuous action space, it it very important that the exploration process is efficient (explore a enough states to generalize well) and to keep exploring for a long time in order to test many action in many configuration.

It was sad because we didn't find any way to reproduce the same results than the paper. We should need more time to make more experiments and explore better ways to do exploration efficiently.

Conclusion

This project was an ambitious but facinated project. The idea to start developping an entire multi-agent reinforcement learning package came from the fact that the studied paper involved to reimplement several state-of-the-art deep reinforcement learning algorithms. In fact, we had to reimplement the entire DQN algorithm for the critic, with all its tricks (i.e. experience replay, target network, etc). Moreover, because it is an actor-critic based method, we also had to implement the actor part including the implementation of DDPG algorithm and finally extend this to multi-agent actor-critic case.

We will continue to implement the package with the objective that it can be reusable in future works in various environments. For instance, we would like to try multi-agent reinforcement learning methods in the context of 3D environments and make simple real world application with external agents possible.

Finally, this project teach us that dealing with agents that have continuous action space can be hard and that it involves to take a lot more precautions while training than with finite action space.

Code

- **Code 1** - Multi-Agent Reinforcement Learning API
  https://github.com/blavad/marl

Bibliography

[1] *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments* [Lowe et al., 2017]
https://arxiv.org/abs/1706.02275

[2] *Multi-Agent Particle Environment* [Lowe et al., 2017]
https://github.com/openai/multiagent-particle-envs

[3] *OpenAI Gym* [OpenAI]
https://github.com/openai/gym