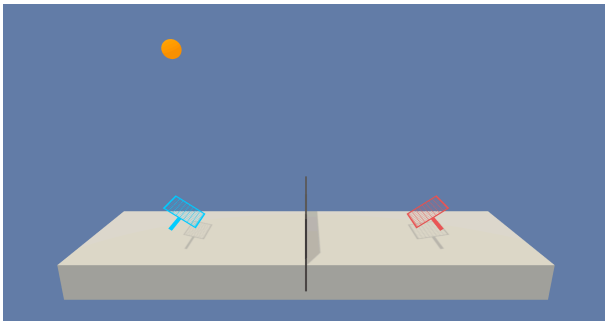
Project 3 Report: Collaboration and Competition

TAN H. CAO*

September 8, 2020

1 Project Goal

In this project, we will work with the Tennis environment, which is not the same one on the ML-Agents GitHub page.



In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, our agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single score for each episode.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

*Department of Applied Mathematics and Statistics, SUNY (State University of New York) Korea, Yeonsu-Gu, Incheon, Korea (tan.cao@stonybrook.edu).

2 Learning Algorithm

To solve the final challenge, we have studied and implemented the *multi-agent deep deterministic policy gradient* (MADDPG) as described in the paper [1].

2.1 Introduction

One well-known issue arising in multi-agent settings when using independently learning agents is that each agent's policy changes during training, resulting in a non-stationary environment and preventing the naive application of experience replay. The authors of the paper proposed an idea of using policy gradient methods with a centralized critic. Their method requires explicitly modeling decision-making process of other agents. They incorporate the robustness of the decision making process of other agents by requiring that agents interact successfully with an ensemble of any possible policies of other agents, improving training stability and robustness of agents after training.

2.2 Background

Consider a multi-agent extension of Markov decision processes (MDPs) called partially observable Markov games. A Markov game for N agents is defined by

- a set of states $\mathcal{S}$: describes the possible configurations of all agents,
- a set of actions $\mathcal{A}_1, \ldots, \mathcal{A}_N$ for each agent,
- a set of observations $\mathcal{O}_1, \ldots, \mathcal{O}_N$ for each agent.

Actions: To choose actions, each agent i uses a *stochastic policy* $\pi_{\theta_i} : \mathcal{O} \times \mathcal{A} \mapsto [0, 1]$ producing the next state according to the state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_N \mapsto \mathcal{S}$.

Rewards: Each agent i obtains rewards $r_i : \mathcal{S} \times \mathcal{A}_i \mapsto \mathcal{S}$ and receives a private observation correlated with the state $\mathrm{o}_i : \mathcal{S} \mapsto \mathcal{O}_i$.

The initial states are determined by a distribution $\rho : \mathcal{S} \mapsto [0, 1]$.

Goals: Each agent i aims to maximize its own total expected return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where γ is a discount factor and T is the time horizon.

Some Issues with Q-Learning and DQN

Q-Learning makes use of an action-value function for policy π as

$$Q^\pi(s, a) = \mathbb{E}[R|s^t = s, a^t = a].$$

This Q-function can be recursively rewritten as

$$Q^\pi(s, a) = \mathbb{E}_{s'}[r(s, a) + \gamma \mathbb{E}_{a' \sim \pi}[Q^\pi(s', a')]].$$

DQN learns the action-value function Q^* corresponding to the optimal policy by minimizing the lossL

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

where $\bar{Q}$ is a target Q function whose parameters are periodically updated with the most recent θ, which helps stabilize learning.

There are some issues when applying Q-Learning to multi-agent settings. First the environment is non-stationary, violating Markov assumptions required for convergence of Q-Learning. Second, the experience replay buffer cannot be used in such a setting since in general

$$P(s'|s, a, \pi_1, \ldots, \pi_N) \neq P(s'|s, a, \pi'_1, \ldots, \pi'_N)$$

for any $\pi_i \neq \pi'_i$.

Policy Gradient (PG) Algorithms

Policy gradient methods are used in a variety of RL tasks. Its main idea is to adjust the parameters θ of the policy in order to maximize the objective function $J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[R]$ by taking steps in the direction of $\nabla_\theta J(\theta)$ given by

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)],$$

where ρ^π is the state distribution.

PG algorithms are known to exhibit high variance gradient estimates, especially in multi-agent settings: since an agent's reward usually depends on the actions of many agents, the reward conditioned only on the agent's own actions exhibits much more variability, thereby increasing the variance of its gradients.

Deterministic Policy Gradient (DPG) Algorithms

We can extend the policy gradient framework to deterministic policies $\mu_\theta : \mathcal{S} \mapsto \mathcal{A}$, where the gradient of the objective $J(\theta) = \mathbb{E}_{s\sim p^\mu}[R(s, a)]$ is given by

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}[\nabla_\theta \mu_\theta(a|s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}].$$

Deep deterministic policy gradient is a variant of DPG where the policy μ and critic Q^μ are approximated with deep neural networks. DDPG is an off-policy algorithm, and samples trajectories from a replay buffer of experiences that are stored throughout training.

2.3 Methods

Multi-Agent Actor Critic

The authors proposed an algorithm that works well in multi-agent settings under the following constraints:

(i) the learned policies can only use local information (i.e. their own observations) at execution time,

(ii) the assumption of a differentiable model of the environment dynamics is not imposed,

(iii) any particular structure on the communication method between agents is not assumed.

This can be considered as a general-purpose multi-agent learning algorithm that could be applied not only to cooperative games with explicit communication channels, but also competitive games and games involving only physical interactions between agents.

The authors can achieve their goal by adopting the framework of **centralized** training with **decentralized** execution. Thus the policies are allowed to use extra information to ease training, so long as this information is not used at test time. It is not possible to do this with O-learning since Q-function generally cannot contain different information at training and test time. So this algorithm proposes a simple

extension of actor-critic policy gradient methods where the critic is augmented with extra information about the policies of other agents.

Consider a game with N agents with policies parametrized by $\theta = \{\theta_1, \ldots, \theta_N\}$, and let $\pi = \{\pi_1, \ldots, \pi_N\}$ be the set of all agent policies. Then the gradient of the expected return for agent i, $J(\theta_i) = \mathbb{E}[R_i]$ is given by

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^\mu, a_i\sim\pi_i}[\nabla_{\theta_i} \log \pi_i(a_i|o_i) Q_i^\pi(\mathbf{x}, a_1, \ldots, a_N)],$$

where $Q_i^\pi(\mathbf{x}, a_1, \ldots, a_N)$ is a *centralized action-value function* that takes inputs as actions of all agents $a_1, \ldots, a_N$ and outputs the Q-value function for agent i.

In the simplest case, $\mathbf{x}$ may consists of the observations of all agents $\mathbf{x} = (o_1, \ldots, o_N)$ (we may include additional state information if possible). Since each Q_i^π is learned separately, agents can have conflicting rewards in a competitive setting.

The above idea can be extended to work with deterministic policies. Consider N continuous policies μ_{θ_i} w.r.t parameters θ_i (abbreviated as μ_i), the gradient can be written as

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{\mathbf{X},a\sim\mathcal{D}}[\nabla_{\theta_i}\mu_i(a_i|o_i) \nabla_{a_i} Q_i^\mu(\mathbf{x}, \mathbf{a_1}, \ldots, \mathbf{a_N})|_{a_i=\mu_i(o_i)}],$$

where the experience replay buffer $\mathcal{D}$ contains the tuples $(\mathbf{x}, \mathbf{x}', a_1, \ldots, a_N, r_1, \ldots, r_N)$, recording experiences of all agents.

The centralized action-value function Q_i^μ is updated as

$$\begin{cases} \mathcal{L}(\theta_i) = \mathbb{E}_{\mathbf{x},a,r,\mathbf{x}'}[Q_i^\mu(\mathbf{x}, a_1, \ldots, a_N) - y]^2 \\ y = r_i + \gamma Q_i^{\mu'}(\mathbf{x}', a_1', \ldots, a_N')|_{a_j'=\mu_j'(o_j)}, \end{cases}$$

where $\mu' = \{\mu_{\theta_1'}, \ldots, \mu_{\theta_N'}\}$ is the set of target policies with delayed parameters θ_i'. It turns out that the centralized critic with deterministic policies works very well in practice. This is called *multi-agent deterministic policy gradient* (MADDPG).

One advantage of MADDPG is that the environment is stationary even the policies change. This is not the case if we do not explicitly condition on the actions of other agents. This assumption is not restrictive: if we aim to train agents to exhibit complex communicative behavior in simulation, this information is often available to all agents. However, this assumption can be relaxed by learning the policies of other agents from observations. Another approach to strengthen the robustness of multi-agent policies to changes in the policy of competing agents is to train a collection of K different sub-policies, as mentioned in paper [1].

2.4 Code Implementation of MADDPG

Our own code here is the modification of our code in project 2 "Continuous Control" as the baseline from "Deep Reinforcement Learning Nanodegree". The code consists of the following files:

(i) **model.py**: This file includes all information about the Network Architecture.

- Network input: Recall that the observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own local observation. Each agent receives 3 stacked vector observation from the environment, representing the current observation together with the last two in the past. As a result, all the networks in this project receive as input 24 variables. Each agent has its own **actor** and **critic** networks.
- The actor network receives an input of 24 variables $= 3 \times 8$ representing the observed state-space and outputs 2 continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

- The critic network, on the other hand, receives an input of 48 variables representing the observed state-space of both agents (24 variables each). This network will take stack the result of its first hidden layer and the actions from the actor network and pass them as inputs for its second hidden layer. The critic network returns the prediction of the optimal action-value function $Q(s, a)$ by using both agents' observations and their best estimated actions. This is an idea of the so-called **centralized critic**.
- The numbers of units in the first and second hidden layers of both networks are 256 and 128 respectively.
- Using batch normalization we address the issue of running computations on large input values and model parameters by scaling the features to be within the same range throughout the model. Therefore, we use batch normalization to the output of the first fully-connected layers of both the actor and critic models.

(ii) **maddpg_agent.py**: this file contains the MADDPG-agent, the replay buffer implementation, and the exploration implementation (QUNoise)

- MADDPG hyperparameters:

```
_buffer_size = int(1e6) # replay buffer size
_batch_size = 128      # minibatch size
_lr_actor = 1e-3      # learning rate of the actor
_lr_critic = 1e-3      # learning rate of the critic
_weight_decay = 0      # L2 weight decay
_learn_every = 1       # learning timestep interval
_learn_num = 1         # number of learning passes
_gamma = 0.95          # discount factor
_tau = 8e-3            # for soft update of target parameters
_ou_sigma = 0.2        # Ornstein-Uhlenbeck noise parameter, volatility
_ou_theta = 0.15       # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
_eps_start = 1.0       # initial value for epsilon in noise decay process in
    Agent.act()
_eps_ep_end = 300      # episode to end the noise decay process
_eps_final = 0         # final value for epsilon after decay
```

- The DDPG agent class has the following methods:
  - `__init__()`: would initialize an agent object.

    + Randomly initialize critic network $Q_{loc}(s, a|\theta^Q)$ and actor network $\mu_{loc}(s|\theta^\mu)$ with weights θ^Q and θ^μ.
    + Initialize the target network Q_{tar} and μ_{tar} with weights

    $$\theta^{Q_{tar}} \leftarrow \theta^{Q_{loc}}, \quad \theta^{\mu_{tar}} \leftarrow \theta^{\mu_{loc}}$$

    by making two copies of the actor and critic networks from the model.

    + Implement Adam optimization algorithm using `optim.Adam`.
    + Initialize the memory: `self.memory = ReplayBuffer(random_seed)`
    + Initialize the noise process by setting: `self.noise = QUNoise(action_size, random_seed)`

  - `step()` would save experience in replay memory and extract a sample from the buffer with the batch size 128 to learn.
  - `act()` would return actions for both agents per current policy, given their respective states.

    + We first pass the states to the actor-local networks.
    + Get the actions for each agent and concatenate them.

+ Add noise to these actions in order to explore the environment:
`action = action + self.noise_decay*self.noise.sample()`
Here we decay the noise in order to exploit later on.
`self.noise_decay *= self.noise_decay`
Note that the noise here is created from the class `OUNoise` with the method `sample()`. This kind of noise is made by perturbing the state by `state = state + dx`, where `dx` is specified in the class `OUNoise`.
Finally we need to clip the action to ensure it takes the values in the interval $[-1, 1]$.

– `learn()` would update the policy and value parameters using the given batch of experience tuples for the defined agent.

+ We update the critic by doing the following tasks:
  * Extract the tuples `(states, actions, rewards, next_states, dones)` from the `experiences`.
  * Get the predicted next-state actions from the actor-target network.
  * Construct `actions_next` vectors relative to the agent.
  * Compute the Q-targets for current states:

$$y^j = r_i^j + \gamma Q_i^{\mu'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \mu'_k(o_k^j)}$$

  * Compute and minimize the critic loss function measuring the gap between Q-expected and Q-targets:

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2.$$

  Here we implemented the technique **gradient clipping** to avoid the exploding/vanishing gradients, which is also a possible improvement for our second project "Continuous Control".

+ We update the actor by using the sample policy gradient method:
  * Get the `actions_pred` from the actor-local network.
  * Construct the action prediction vectors for the relative agent.
  * Compute and minimize the actor loss function.

– `soft_update()` would update the model parameters softly with parameter `tau = 0.008`.

+ Finally we update the noise decay parameter.

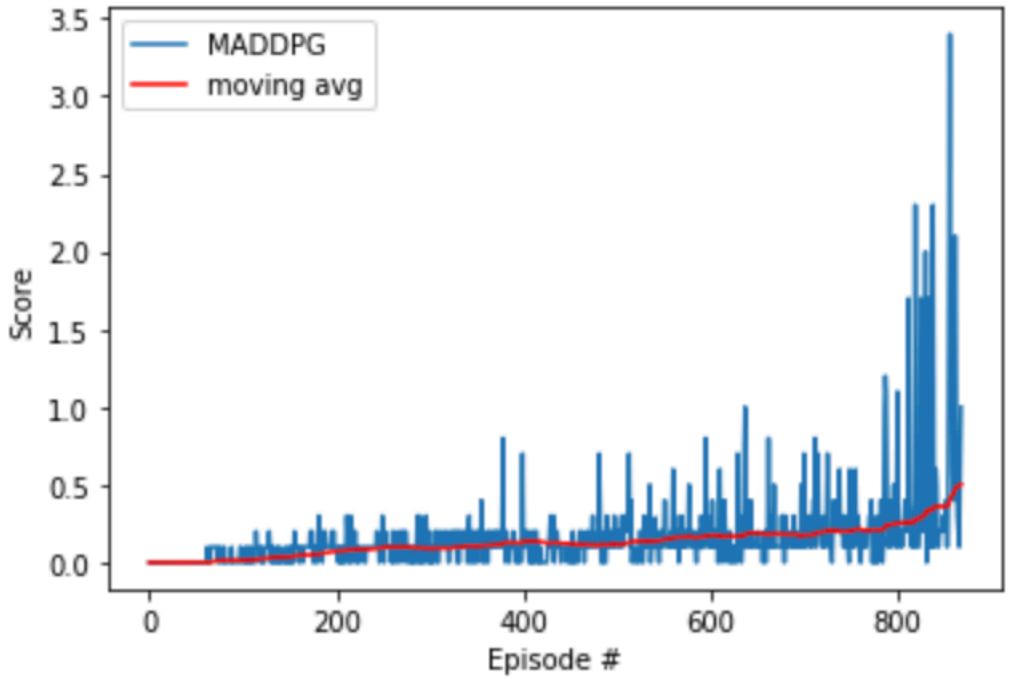
(iii) **Tennis.ipynb**: this Jupyter notebook has the reference project starter code and the code that we have implemented to train the agent.

3 Plot of the Rewards

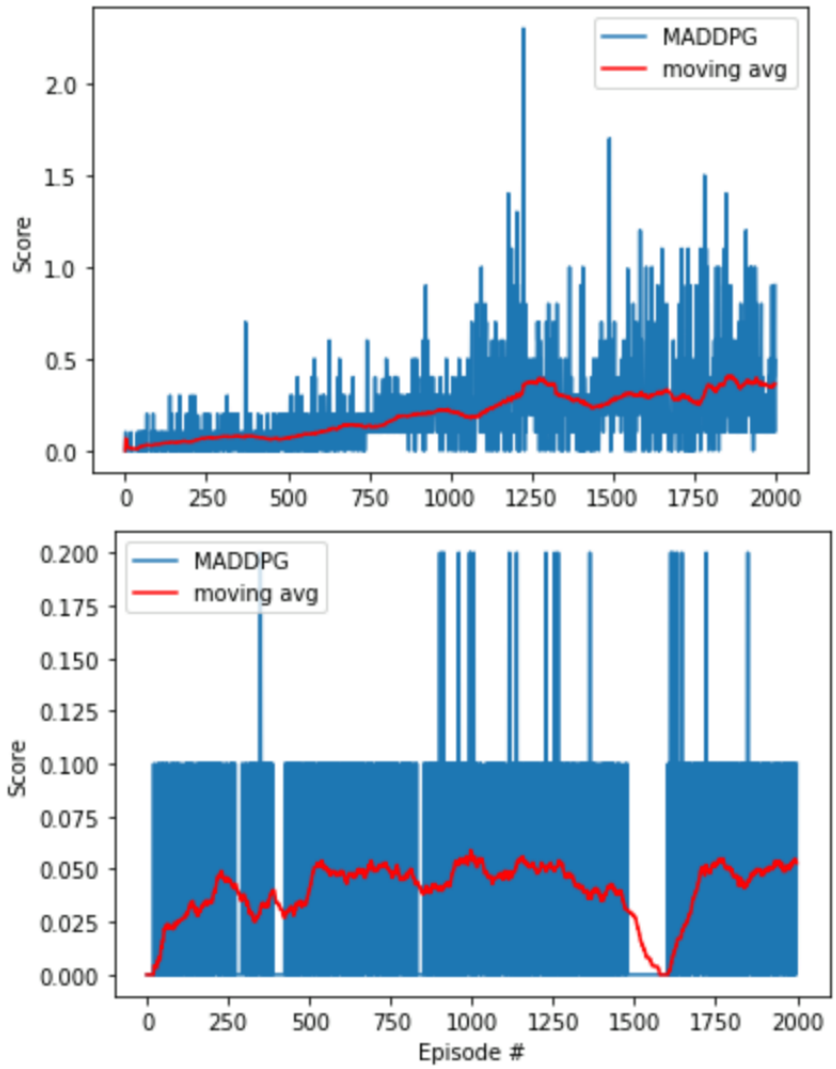
The graph below shows the rewards per episode within the training phase, as well as the moving average. The best-performing agents were able to solve the environment in 769 episodes with a top score of 3.40 and a top moving average of 0.98.

```
Episode number 630      Within  10 episodes     Mean score:  0.13 (H:  0.70 / L:  0.00)        Moving Average: 0.171
Episode number 640      Within  10 episodes     Mean score:  0.30 (H:  1.00 / L:  0.00)        Moving Average: 0.187
Episode number 650      Within  10 episodes     Mean score:  0.17 (H:  0.40 / L:  0.10)        Moving Average: 0.189
Episode number 660      Within  10 episodes     Mean score:  0.10 (H:  0.20 / L:  0.00)        Moving Average: 0.182
Episode number 670      Within  10 episodes     Mean score:  0.24 (H:  0.80 / L:  0.10)        Moving Average: 0.185
Episode number 680      Within  10 episodes     Mean score:  0.13 (H:  0.30 / L:  0.00)        Moving Average: 0.180
Episode number 690      Within  10 episodes     Mean score:  0.16 (H:  0.30 / L:  0.10)        Moving Average: 0.183
Episode number 700      Within  10 episodes     Mean score:  0.16 (H:  0.50 / L:  0.10)        Moving Average: 0.174
Episode number 710      Within  10 episodes     Mean score:  0.22 (H:  0.70 / L:  0.10)        Moving Average: 0.179
Episode number 720      Within  10 episodes     Mean score:  0.35 (H:  0.80 / L:  0.10)        Moving Average: 0.196
Episode number 730      Within  10 episodes     Mean score:  0.21 (H:  0.70 / L:  0.10)        Moving Average: 0.204
Episode number 740      Within  10 episodes     Mean score:  0.26 (H:  0.60 / L:  0.00)        Moving Average: 0.200
Episode number 750      Within  10 episodes     Mean score:  0.22 (H:  0.60 / L:  0.00)        Moving Average: 0.205
Episode number 760      Within  10 episodes     Mean score:  0.22 (H:  0.60 / L:  0.10)        Moving Average: 0.217
Episode number 770      Within  10 episodes     Mean score:  0.12 (H:  0.20 / L:  0.00)        Moving Average: 0.205
Episode number 780      Within  10 episodes     Mean score:  0.17 (H:  0.40 / L:  0.00)        Moving Average: 0.209
Episode number 790      Within  10 episodes     Mean score:  0.48 (H:  1.20 / L:  0.00)        Moving Average: 0.241
Episode number 800      Within  10 episodes     Mean score:  0.25 (H:  0.50 / L:  0.10)        Moving Average: 0.250

Episode number 810      Within  10 episodes     Mean score:  0.29 (H:  1.10 / L:  0.10)        Moving Average: 0.257
Episode number 820      Within  10 episodes     Mean score:  0.59 (H:  2.30 / L:  0.10)        Moving Average: 0.281
Episode number 830      Within  10 episodes     Mean score:  0.45 (H:  1.70 / L:  0.10)        Moving Average: 0.305
Episode number 840      Within  10 episodes     Mean score:  0.77 (H:  2.30 / L:  0.00)        Moving Average: 0.356
Episode number 850      Within  10 episodes     Mean score:  0.26 (H:  0.60 / L:  0.10)        Moving Average: 0.360
Episode number 860      Within  10 episodes     Mean score:  0.98 (H:  3.40 / L:  0.10)        Moving Average: 0.436
<-- Environment solved in 769 episodes!
<-- Moving Average: 0.503 over past 100 episodes
```



Before getting the best result, we tried to run the code with different hyperparameters but we encountered several issues such as exploding gradients, long training time, and unsuccessful learning agents. But after all, we were able to find the best hyperparameters that gives us the best performance after multiple attempts.



4 Ideas for Future Work

We will consider the following for our future work:

- Implementing the method of "Inferring Policies of Other Agents " proposed in the paper [1] to relax the assumption of knowing the observations and policies of other agents.
- Implementing the prioritized experiences in our MADDPG algorithm.
- In order to ensure the robustness of multi-agent policies to changes in the policy of competing agents, we may train a collection of different sub-policies as proposed in the paper [1].

5 Acknowledgments

We are gratefully indebted to Nilson Chapagain for his useful remarks and discussions on how to select good hyperparameters so that the agents could achieve their highest performance.

References

[1] Ryan Lowe, Yi Wu, Aviv Tammar, Jean Harb, Pieter Abbeel, & Igor Mordatch: Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments.