Multi-Agent Actor-Critic for Cooperative Robot Arm Environments

Daniel Chernis
260707258
McGill University
December 2019

Abstract

The Multi-Agent Deterministic Deep Policy Gradient (MADDPG) is a reinforcement learning algorithm that tackled scenarios with two-dimensional agents in cooperative and competitive environments. I explored this algorithm in a robotics setting with a single and two robot arms side by side where both arms perform independent actions until either reaches the goal. After 300 000 steps, the first environment appears to learn very slowly and the reward value oscillates frequently; the latter shows impressive learning abilities in the cooperative environment with a steady convergence after 100 000 steps. This demonstrates that MADDPG requires more investigation into complex observation and action spaces especially in a robotic scenario.

1. Introduction

Reinforcement Learning (RL) provides an agent with learning through its own experience in a given environment faster than a human ever could. Many environments have been suggested including AlphaGo which taught itself to become the best Go player with 99.8% winning rate against other Go programs and defeated the human European Go champion by 5 games to 0 [5]. AlphaStar is rated at Grandmaster level for all three StarCraft races and above 99.8% of officially ranked human players [8]. StarCraft is known for a large action space, imperfect information and requires long term planning to build micro and macro-strategic skills. OpenAI Five plays 180 years worth of Dota 2 games against itself every day using a reward function to optimize its performance [OA]. The question becomes how to improve the learning and adapting it to new environments.

Using a policy, a mapping from perceived states of the environment to actions to be taken, an agent will try to maximize the total reward, generated by the environment, it receives in the long run. The learning rate can be optimised through a value function, which targets the expected value of the reward using deterministic policies [10].

Alternatively, policy methods use a stochastic policy to explicitly represent its own function approximator independent of value function [11]. Using a policy gradient, a policy can be updated incrementally thought a defined step size toward a local optimal policy. Additionally, learning a value function and using it to reduce the variance of the gradient estimate appears to be essential for rapid learning.

Unfortunately, such traditional RL approaches such as Q-Learning or policy gradient are poorly suited for multi-agent environments. Since the policy of each agent changes as the training progresses, the environment becomes non-stationary from the perspective of any agent [1]. Hence, one needs to adjust the policy of one agent to explain the policy of the other agent.

[9] introduces MADDPG, a multi-agent deep deterministic policy gradient learning algorithm that learns policies with only local observations at execution time. It assumes the same model of environment dynamics or structure on communication method between agents and tests on many cooperative and competitive environments.

My experiment consists of setting up a one and two robot arm environment and exploring MADDPG on both environments to analyse further the effects of this RL algorithm.

2. Related Work

The most naive approach to solve the multi-agent reinforcement learning problems is to treat each agent independently such that each such agent considers the rest of the agents as part of the environment [6]. This algorithm corresponds to independent Q-Learning (IQL).

REINFORCE algorithm is one of the first successful policy-gradient algorithms which proposed in 2000 [5]. It applies Monte Carlo method as an approximation for the Q-value. The Actor-Critic (AC) model adds to the REINFORCE algorithm the approximator Critic, to learn the Q-values. The critic receives states and returns the approximation.

Going beyond that is the MADDPG [9], that combines the naive multi-agent RL learning with Policy Gradients to derive an algorithm that works well in multi-agent settings with the following constraints: (1) learned policies can only use their own observations, (2) there are no assumptions of a differentiable model of the environment dynamics, and (3) there is no particular structure on communication between agents. The model uses centralised critics. Their approach learns a separate centralised critic for each agent and is applied to competitive and cooperative environments with continuous action spaces. Essentially, additional information is provided to the value function during training in order to reduce the variance of the policy gradients and allow the value function to provide more meaningful feedback for improving the policy. MADDPG outperforms DDPG with the correct behavior for both cooperative communication and physical deception scenarios.

Later works use both DDPG and MADDPG on new environments like a Unity Tennis Game [2] and Robot Soccer with 4 player-agents [3]. [2] finds that learning is unstable for his environment and modifies the algorithm by dividing the input into state (for keeping the environment stationary for that particular agent, which includes state of all agents) and actions of all other agents except the acting agent.

I will experiment with MADDPG in a robotic arms environment to analyse the algorithm in a robotics setting.

3. Background

Markov Decision Process: It is a partially observable Markov game. A Markov game for N agents contains a set of states S, set of actions $A_1, ..., A_N$ and set of observations $O_1, ..., O_N$. Each agent i chooses actions using a policy $\pi_{\theta_i} : O_i \times A_i \rightarrow [0, 1]$ which produces the next state with a state transition function $T : S \times A_1 \times ... \times A_N \rightarrow S$. Additionally, each agent i receives rewards $r_i : S \times A_i \rightarrow R$ as a function of its state and action, and a private observation correlated with the state $o_i : S \rightarrow O_i$. Initial states are determined by a distribution $\rho : S \rightarrow [0, 1]$. Each agent i aims to maximise its own expected return value over time:

$R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where $\gamma \rightarrow [0, 1]$ is a discount factor and T is the total time interval.

The discount factor determines how valuable a reward is immediately versus in the long run [10].

Q-Learning and Deep Q-Networks (DQN): These are popular RL methods, previously applied to multi-agent settings [9]. Q-Learning has a uses a recursive action-value (*Q*) function for policy π as $Q^{\pi}(s, a) = E_{s'}[r(s, a) + \gamma E_{a' \sim \pi}[Q^{\pi}(s', a')]]$.

DQN learns the optimal action-value function Q^* corresponding an optimal policy by minimising loss:

$$L(\theta) = E_{s,a,r,s'}[(Q^*(s, a|\theta) - y)^2] \text{ , where } y = r + \gamma max_{a'} \overline{Q^*}(s', a') \quad (1)$$

where $\overline{Q}$ is a target Q function whose parameters are periodically updated with the most recent θ, which helps stabilize learning. Another crucial component in stabilizing DQN is the use of an experience replay buffer *D* containing tuples (*s, a, r, s'*).

Q-Learning can be directly applied to multi-agent settings by having each agent learn the optimal function Q_i independently [6]. However, because agents are independently updating their

policies as learning progresses, the environment appears non-stationary from the view of any one agent, violating Markov assumptions required for convergence of Q-learning. Another difficulty observed in [9] is that the experience replay buffer cannot be used in such a setting since in general, $P(s|s, a, \pi_1, ..., \pi_N) \neq P(s'|s, a, \pi'_1, ..., \pi'_N)$ where any $\pi_i \neq \pi'_i$.

Policy Gradient (PG) Algorithms: These are alternate RL methods which differ in how they compute Q^{π}. You directly adjust θ of the policy to maximize the objective $J(\theta) = E_{s \sim p^{\pi}, a \sim \pi_{\theta}}[R]$ by taking steps in the direction of the gradient $\nabla_{\theta} J(\theta)$. Using the Q function defined previously, the gradient of the policy can be written as:

$$\nabla_{\theta} J(\theta) = E_{s \sim p^{\pi}, a \sim \pi_{\theta}}[\nabla_{\theta} log\, \pi_{\theta}(a|s)\, Q^{\pi}(s, a)] \ , \tag{2}$$

where p^{π} is the state distribution.
Learning a value function and using it to reduce the variance of the gradient estimate appears to be essential for rapid learning [11].

The policy gradient theorem spread the path to many algorithms, which often differ in how they estimate Q^{π}.

For example, sampling the return $R^t = \sum_{i=t}^{T} \gamma^t r_i^{\ t}$, leads to the REINFORCE algorithm [5]. Alternatively, one could learn an approximation of the true action-value function $Q^{\pi}(s, a)$ by temporal-difference learning [10], where $Q^{\pi}(s, a)$ is called the critic and leads to a variety of actor-critic algorithms [10].

Policy gradient methods are known to exhibit high variance gradient estimates. This is exacerbated in multi-agent settings: since an agent's reward usually depends on the actions of many agents, it is conditioned only on the agent's own actions (when the actions of other agents are not considered in the agent's optimization process), and exhibits much more variability, thereby increasing the variance of its gradients [9].

Consider a simple setting where the probability of taking a gradient step in the correct direction decreases exponentially with the number of agents. Assuming binary actions $P(a_i = 1) = \theta_i$ where $R(a_1, ..., a_N) = 1_{a_i = ... = a_N}$ and an uninformed scenario where agents are initialized to $\theta_i = 0.5 \forall i.$, the gradient of the cost J with policy gradient is $P(\langle \widehat{\nabla} J, \nabla J \rangle > 0) \propto (0.5)^N$, where $\widehat{\nabla} J$ is the policy gradient estimator from a single sample and ∇J is the true gradient [9].

Deterministic Policy Gradient (DPG) Algorithms: The above policy gradients can be extended using deterministic policies $\mu_{\theta} : S \rightarrow A$. Under certain conditions the gradient of the objective $\nabla_{\theta} J(\theta) = E_{s \sim p^{\mu}}[R(s, a)]$ can be written as:

$$\nabla_\theta J(\theta) = E_{s\sim D}\left[\nabla_\theta \mu_\theta(a|s)\ \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right] \quad (3)$$

where action space A must be continuous due to the nature of gradients.

Deep deterministic policy gradient (DDPG): This is a variant of DPG where the policy and critic Q^μ are approximated with deep neural networks. DDPG is an off-policy algorithm i.e., it learns the value of the optimal policy independently of the agent's actions. Furthermore, it samples trajectories from a replay buffer of experiences that are accumulated during training. DDPG also uses of a target network, as DQN [9].

Multi-Agent Deterministic Deep Policy Gradients (MADDPG) [9]: To derive an algorithm that works well in a multi-agent settings, a few constraints were followed: (1) the learned policies can only use local information (i.e. their own observations) at execution time, (2) no assumption a differentiable model of the environment dynamics, and (3) no assumption any particular structure on the communication method between agents (don't assume a differentiable communication channel). The above would apply to both cooperative games with explicit communication channels, and competitive games involving only physical interactions between agents. This allows the policies to use extra information to ease training, as long as this information is not used at test time. It is unnatural to do this with Q-learning, as the Q function generally cannot contain different information at training and test time.

Henceforth, the critics of the actor-critic policy gradient methods are given extra information about the policies of other agents. Assuming policies for N agents $\pi = \{\pi_1, ..., \pi_N\}$ parametrized by $\theta = \{\theta_1, ..., \theta_N\}$, the gradient of the expected return $J(\theta_i) = E[R_i]$ for agent i becomes:

$$\nabla_{\theta_i} J(\theta_i) = E_{s\sim p^\mu,\ a_i\sim\pi_i}\left[\nabla_{\theta_i} log\ \pi_i(a_i|o_i)\ Q_i^\pi(x, a_1, ..., a_N)\right], \quad (4)$$

where $Q_i^\pi(x, a_1, ..., a_N)$ is a centralized action-value function taking as input the actions of all agents, $a_1, ..., a_N$, some state information consisting of observations of all agents $x = o_1, ..., o_N$, and any additional state information. Since each Q_i^π is learned separately, agents can have arbitrary reward structures, including conflicting rewards in a competitive setting (e.g. collisions).

For deterministic policies, considering N continuous policies μ_{θ_i} (abbreviated as μ_i), the gradient can be written as:

$$\nabla_{\theta_i} J(\mu_i) = E_{x,a\sim D}\left[\nabla_{\theta_i} \mu_i(a_i|o_i)\ \nabla_{a_i} Q_i^\mu(x,\ a_1, ..., a_N)|_{a_i=\mu_i(o_i)}\right] \quad (5)$$

The experience replay buffer D contains tuples of information state, actions for every agent, and corresponding reward for every agent. The Q_i^μ is updated as:

$$L(\theta_i) = E_{x,a,r,x'}[(Q_i^{\mu}(x, a_1, ..., a_N) - y)^2]; \quad y = r_i + \gamma Q_i^{\mu'}(x', a'_1, ..., a'_N)|_{a'_j = \mu'_j(o_j)} \quad (6)$$

where $\mu' = \{\mu_{\theta_1}, ..., \mu_{\theta_N}\}$ is the set of target policies with delayed parameters θ. Finding the centralized critic with deterministic policies is MADDPG.

Inferring Policies of Other Agents: To remove the assumption of knowing other agents' policies as required in the update equation above, each agent i can additionally maintain an approximation $\hat{\mu}_{\varphi_i^j}$ to the true policy μ_i of agent j (where φ are the parameters of the approximation). This approximate policy is learned by maximizing the log probability of agent j's actions, with an entropy regularizer (abbreviate $\hat{\mu}_{\varphi_i^j}$ to $\hat{\mu}_i^j$):

$$L(\varphi_i^j) = -E_{o_j, a_j}[log\ \hat{\mu}_i^j(a_j|o_j) + \lambda H(\hat{\mu}_i^j)], \quad (7)$$

where H is the entropy of the policy distribution. With the approximate policies, y in Eq. 6 can be replaced by an approximate value *ŷ* calculated as follows:

$$\hat{y} = r_i + \gamma Q_i^{\mu'}(x', \hat{\mu}_i'^1(o_1), ..., \mu'_i(o_i), ..., \hat{\mu}_i'^N(o_N)) \quad (8)$$

where $\hat{\mu}_i'^j$ denotes the target network for the approximate policy $\hat{\mu}_i^j$.
Eq. 7 can be optimized: before updating Q_i^{μ}, the centralized Q function, the latest samples of each agent j is taken from the replay buffer to perform a single gradient step to update φ_i^j.
*Note also that, in the above equation, the action log probabilities of each agent is inputted directly into Q, rather than sampling.

Agents with Policy Ensembles: A recurring issue with multi-agent RL is that the environment is non-stationarity due to the agents' changing policies. This is particularly true in competitive settings, where agents can derive a strong policy by overfitting to the behavior of their competitors. Such policies are undesirable as they are fragile, and may fail when competitors alter strategies.

To obtain multi-agent policies that are more robust to changes in the policy of competing agents, [9] proposes to train a collection of K different sub-policies. At each episode, 1 particular sub-policy is randomly selected for each agent to execute. Suppose that policy μ_i is an ensemble of K different sub-policies with sub-policy k denoted by $\mu_{\theta_i^{(k)}}$ (denoted as $\mu_i^{(k)}$. For agent i, the ensemble objective is maximised: $J_e(\mu_i) = E_{k\sim unif(1,K),\ s\sim p^{\mu}, a\sim\mu^{(k)}}[R_i(s,a)]$.

Since different sub-policies will be executed in different episodes, a replay buffer $D_i^{(k)}$ is maintained for each sub-policy $\mu_i^{(k)}$ of agent i. Accordingly, the gradient of the ensemble objective with respect to $\theta_i^{(k)}$ is derived as follows:

$$\nabla_{\theta_i^{(k)}} J_e(\mu_i) = \frac{1}{K} E_{x,a\sim D_i^{(k)}}[\ \nabla_{\theta_i^{(k)}} \mu_i^{(k)}(a_i|o_i) \nabla_{a_i} Q_i^{\mu_i}(x,\ a_1, ..., a_N)|_{a_i=\mu_i^{(k)}(o_i)}\ ] \quad (9)$$

You can see the full algorithm in the *appendix*.

Nevertheless, even with optimized policies, MADDPG was only tested with a limited number of environment consisting of circular agents with simplistic action and observation spaces. This algorithm needs more evaluation of higher complexity spaces to verify the efficiency of the training agent.

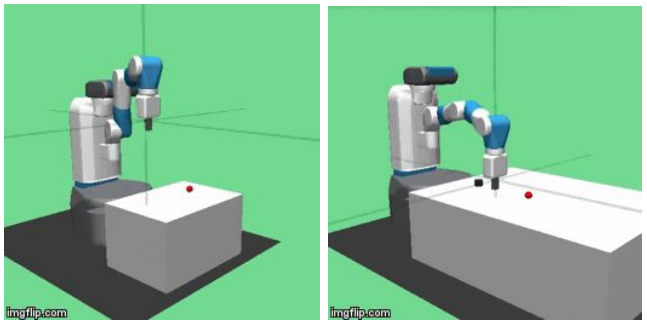
4. Experiments

The following experiment consists of applying MADDPG to a single-arm and two-arm robot environment. See full code, data, and results at:
https://github.com/dch133/MADDPG-With-Robot-Arms

4.1 One-Arm Environment

The first step was setting up the coding environment compatible with MADDPG experiment. After going through the available scenarios, and replicating the results, I set up a robotics related environment. My first attempts included making that code compatible with Gazebo for Robot arm simulations, then robot soccer and finally Open AI Gym Robotics environment.

Here are the available single arm Fetch environments offered by OpenAI. All of which require Mujoco Licenses to run:



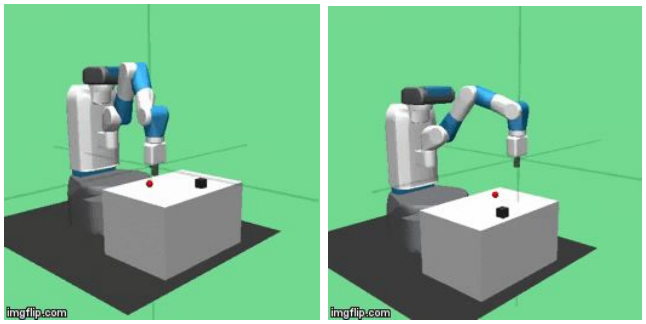


Figure 1: Illustrations of the experimental environments going from left to right, top to bottom: (a) *FetchReach-v0* (b) *FetchSlide-v0 (*c) *FetchPush-v0* (d) *FetchPickAndPlace-v0*

FetchReach-v0: Fetch has to move its end-effector to the desired goal position

FetchSlide-v0: Fetch has to hit a puck across a long table such that it slides and comes to rest on the desired goal.

FetchPush-v0: Fetch has to move a box by pushing it until it reaches a desired goal position.

FetchPickAndPlace-v0: Fetch has to pick up a box from a table using its gripper and move it to a desired goal above the table.

This experiment only operates using *FeachReach* environment for both one-arm and two-arm. This is a similar behaviour to the '*Simple' Scenario* offered by MADDPG team, where an agent (a circle in their case and the tip of the arm in my case) attempts to reach goal coordinates.

4.2 Two-Arm Environment

The following was tested to set up a two-arm environment: (1) Parallelising the same arm environment [4], (2) creating a new Gym environment with 2 arms and Robot.env, (3) replicate the MultiAgentEnv class from MADDPG code based on gym.Env but with Robot arms, (4) simulating two arms in Mujoco directly, the library providing the graphics for Robotics simulations in Gym, (5) making two copies of the single-arm environment that each move towards the same goal, and (6) modifying the parameters of the agent directly in the Mujoco environment source code (*gym/envs/robotics/assets/hand/reach.xm*l).

I settled on (5) due to its simplicity and comprehensibility in code. The goal of that environment is to reach the Red Dot (Goal) in the least amount of steps. In order to modify this for two arms, I simultaneously operate on two instances of this single arm environment with modifications to one of the arms: given the left arm as a reference frame, I set coordinates of the goal to the same relative position on both arms. For one arm, that goal appears more on the left to simulate the reference frame of an arm on the right of the other.

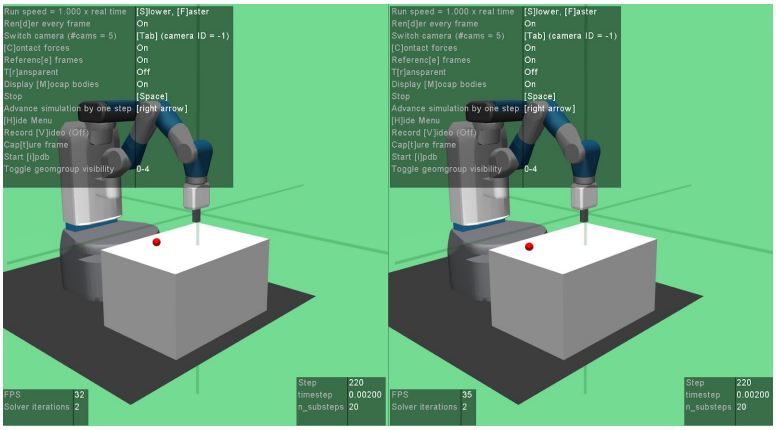


Figure 2: Simulation of Open AI Robotics Environment for FetchReach-v1 for two robot arms where the left image is the *Right-Arm*, and the right image is the *Left-Arm*

4.3 Setting Up the Arm Environment

In order to make both single and two arm environment compatible with the existing MADDPG code, I did the following:

(1) Set the environment.unwrapped.reward_type to an empty string (‘’): the default was *‘sparse’* which sets the reward value to 1 if Goal was reached, and -1 if Goal was not reached. The reward value needs to be more explicit.

(2) Instead of creating a variation of *MultiAgentEnv* [9], I call the ‘*create_2_arm_env()*’ which creates two instances of the 1-arm environment sharing the same relative goal position, and returns a list containing the left and right arm environment respectively. I would pick the left arm only for the single arm setting or use both otherwise.

(3) Set the environment’s *action_space* to a ‘*list type*’ to create the trainer object, and reverted back from list afterwards.

(4) The rest stays the same assuming ‘*env.n = 1*’

Note that the Robotics environment has much more parameters than the scenarios offered by MADDPG team code . The scenarios have a limited two-dimensional action and observations space. For Fetch-Environment, the first 3 dimensions of the action space are an offset in Cartesian space from the current end effector position and the 4th dimension is the state of the parallel gripper in joint space.

4.4 Running MADDPG on Single Arm Environment

After setting up the environment, the reward gets updated based on the 3-dimensional distance from the tip of the arm to the red point goal. The arm then takes 1000 steps (called episodes) per epoch for a total of 300 epochs.

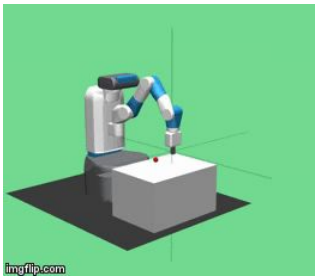


Figure 3: Sample simulation of *FetchReach-v1* for One Arm Robot using MADDPG for Learning

Core training parameters (passed as input):

- *--lr* : learning rate (default: 1e-2)
- *--gamma* : discount factor (default: 0.95)
- *--batch-size* : batch size (default: 1024)
- *--num-units* : number of units in the Multi Layer Perceptron (MLP) (default: 64)

The discount factor was increased and decreased to test the learning accuracy. With a lower discount factor, the agent appears to get closer to the goal quicker but then quickly appears to forget that, and its reward value seems to oscillate. On the other hand, as the discount factor approaches 1, the agent's reward function grows slower. Thus, for 300 000 episodes, the default parameters were considered for best reward values in *Figure 4* and *5*.

Increasing the number of units in the MLP, makes the model more complex, i.e. more accurate in the long run, but it takes a longer time to reach that accuracy. For that reason it wasn't tested in this experiment, but should be considered with more powerful computers with strong GPUs.

The learning rate and batch size were kept at default.

4.5 Running MADDPG on Two Arm Environment

As described in section 4.3, this environment was run for for 300 000 episodes and default training parameters. It is displayed as two instances *FetchReach* (see *Figure 2*) with the goal being in the same relative position.

Furthermore, there were two options when it comes to rewards:

(1) The reward at an iteration corresponds to the reward of the arm closest to the goal. Additionally, you need to make sure the arms don't collide i.e occupy the same *observation_space['achieved_goal']* value. Once either the left or the right arm reached the goal, stop.

(2) Make one arm mirror the other, based on which arm has the highest reward out of the two in the previous iteration.

For this experiment, the first option only was considered for the reward. The learning time per 1000 episodes was twice as slow as for the single arm setting, due to the fact that MADDPG was running on 2 agents. Unfortunately, the collision of both arms was not penalised in the reward function which could be responsible for a higher reward values. This still demonstrates that MADDPG is working on a multi agent environment but requires accounting for complex spaces not only in the setup, but also in learning.

5. Results

As the results show, both environments demonstrate learning capabilities and a rather good adaptability of the MADDPG algorithm to new environments. Yet, the two-arm environment is more efficient at learning compared to the one-arm environment most probably due to the fact that it has twice as much limbs and can cover twice as much ground.

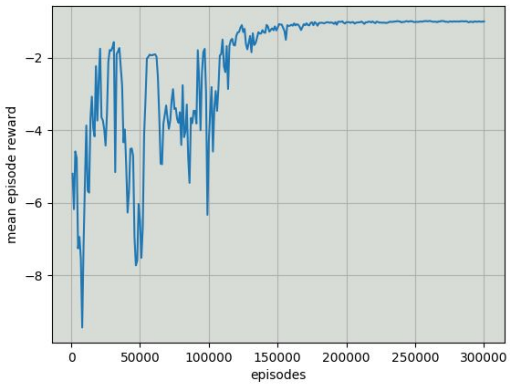


Figure 4: Agent reward on cooperative communication after 300 000 episodes for 2-arm environment

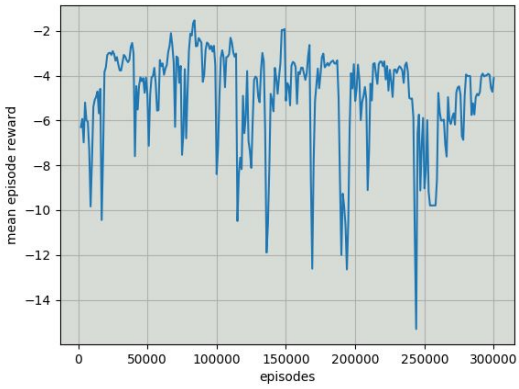


Figure 5: Agent reward on cooperative communication after 300 000 episodes for 1-arm environment

Based on *Figure 4*, I believe it takes the first 100 000 episodes to explore the environment then afterwards the accuracy increases until convergence. There appears to be a slow and steady learning over time for the two-arm environment. *Figure 5* demonstrates a faster running algorithm with highly ineffective learning for the 1-arm environment. See the *appendix* for the

time it took to run 1000 episodes in both environments. It takes twice the amount of time to run the same number of episodes as for the two-arm environment.

I hypothesise that such an algorithm still needs more analysis on complex observation and action spaces (like robot arms provide), which tend to have different patterns that should be formalised. For the simple environments with circles for agents offered by [9], the reward value converges after a mear 25 000 episodes. In the robotics setting, especially for the single arm environment, there are still a lot of oscillations after 100 000 episodes, probably due to the nature of the spaces which slow down convergence. Additionally, the choice of training parameters mentioned in *section 4.4* definitely affects the performance of the algorithm.

To improve convergence speed in the two-arm scenario, I enforced that these cooperative agents should update their policy at each episode and the reward of that state would be assigned based on the arm closest to the goal. A next step would be to allow the arms to communicate and guide each other rather than functioning virtually independently.

6. Conclusion and Future Work

I applied MADDPG algorithm to a robotics environment. While the tests were few, one can conclude that while the one-arm requires more investigation, the two arm environment shows great promise as a learning environment. The core training parameters need to be tested out thoroughly individually, as well as in different combinations of each other to see how they affect the learning speed, final accuracy and reward value.

Moreover, only two-arms cooperation scenario was experimented with. There are other similar OpenAI robot arm environments that could be suitable for different types of cooperation as well as competition, as described in *Figure 1*. In other words, a more polymorphic environment should be set up allowing to choose the type of robotic environment along with the number of arms. On top of that, there were many interesting scenarios portrayed for the original MADDPG code. They should also be translated, in a way, to robotic arms. For example, in a competitive, predator prey scenario, using *FetchSlide-v0*, two or more robotic arms can push pucks towards a goal, while other arms can push their pucks to interfere by causing collisions.

Additionally, in *section 4.2* and *4.5* different set ups and reward types were discussed but not tested. They should be considered, seeing as they could potentially have promising results.

Finally, the arms were assumed next to each other at a certain distance apart. Another approach would place the arms at a right angle or in front of each other yielding a different joint observation space which would impact all the potential environments mentioned above.

7. Acknowledgements

I thank professor Dave Meger at McGill university for guidance, interesting discussions related to this paper, and for comments on the paper draft. I thank Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch for providing the code base that was used for the MADDPG algorithm as a whole and for some early experiments associated with this paper. Finally, I'd like to thank OpenAI, and Mujoco for fostering an engaging and productive research environment for robotics.

References

[1] A. Jadid and D. Hajinezhad, A Review of Cooperative Multi-Agent Deep Reinforcement Learning, 2019, https://arxiv.org/pdf/1908.03963.pdf

[2] A. Kushwaha. Improving OpenAI Multi-Agent Actor-Critic RL Algorithm. Medium, Brillio Data Science, 2019. medium.com/brillio-data-science/improving-openai-multi-agent-actor-critic-rl-algorithm-27719f3cafd4, https://github.com/abhismatrix1/Tennis-MultiAgent.

[3] D. Barbosa. Multi-Agent Deep Deterministic Policy Gradient (MADDPG) to train four agents to play Soccer, 2018. https://github.com/danielnbarbosa/soccer_twos

[4] D. Reddy. Efficient Multiple Gym Environments. Squadrick, 2018. https://squadrick.github.io/journal/efficient-multi-gym-environments.html

[5] D. Silver, A. Huang, C. Maddison, et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016) doi:10.1038/nature16961

[6] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Proceedings of the tenth international conference on machine learning, pages 330–337, 1993. http://web.media.mit.edu/~cynthiab/Readings/tan-MAS-reinfLearn.pdf

[7] OpenAI. OpenAI Five, 2018. https://openai.com/blog/openai-five/

[8] O. Vinyals, I. Babuschkin, W.M. Czarnecki, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature 575, 350–354 (2019), doi:10.1038/s41586-019-1724-z

[9] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments.arXiv preprintarXiv:1706.02275, 2017.

[10] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction, volume 1. MIT press Cambridge, 1998.

[11] [11] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems, pages 1057–1063, 2000

Appendix

For completeness, I provide the MADDPG algorithm below [9]:

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

Results from 300 000 Iterations for 1-Arm and 2-Arm Environment:
See https://github.com/dch133/MADDPG-With-Robot-Arms