ECE 6563 Project: Formation Control and Collision Avoidance using Muti-Agent Policy Gradient

Qian Luo, Yaru Niu, Sicong Jiang

Abstract

Recently, the formation control and obstacle avoidance of multi-agent systems become a research hotspot. To achieve better performance in multi-agent control, we train learning-based models based on the Deep Deterministic Policy Gradient (DDPG) and Multi-Agent Deep Deterministic Policy Gradient (MADDPG) to achieve multi-agent formation control and obstacle avoidance. We apply the DDPG algorithm in the Robotarium simulation environment to achieve a basic fixed point control which enables the robots to achieve fixed targets while avoiding collision with other robots. Then we employ the MADDPG in the OpenAI Multi-agent environment to perform more complex formation control and generalize the trained policy to unseen situations. Through experiments, we find that both DDPG and MADDPG can realize the formation control and obstacle avoidance, while the MADDPG algorithm is more powerful in multi-agent formation tasks.

1 Introduction

In recent years, cooperative control of multi-agent systems has been widely studied due to the development of advanced theory of complex systems and its broad applications in many fields including biology, physics, robotics, vehicles and control engineering. Problems of consensus, flocking, formation, rendezvous, coverage and swarming are some important research issues, and the common task is to develop distributed schemes or protocols to ensure the realization of complicated global goals. In particular, the formation control problem is to find a coordinated scheme for networks of multiple agents such that they would reach and maintain some desired, possibly time-varying formation or group configuration. Applications of formation control can be found in the automotive and aerospace areas, ranging from assembling structures, exploration of unknown environments, navigation in hostile environments, to cooperative transportation tasks.

In a multi-agent system, finding a feasible policy of formation without collision for each agent is important and challenging. And it can be difficult to model and optimize the formation strategy using traditional control theory. In recent years, the development of deep reinforcement learning (RL) techniques provides the robot agent with the strong abilities of learning policies by exploring and interacting with the environments itself. The policy gradient methods will model and optimize the policy directly instead of choosing actions based on the Q-state value function, which may suffer when dealing with continuous spaces. The DDPG [1] network combines the advantages of Actor Critic and DQN [4]. It doesn't output the probability of behavior, but the specific behavior, which is used for the prediction of continuous action.This improves its stability and accelerates its convergence.

However, when it comes to multi-agent system, traditional policy gradient algorithms, such as DDPG and TRPO [6], may have issues with increasing variance with the number of agents growing. In this project we will explore a policy gradient method suitable for the formation control in a multi-agent system.

2 Related Work

The simplest way to apply RL to the multi-agent system is to use independently-learning agent [7]. At first, Q-learning was used to train the agent. However, Q-learning did not perform well. Also, independent RL performs bad because of the changing of each agent's policy during training. Therefore, people turned to research on the multi-agent policy gradient method.

Ryan Lowe et al. proposed the MADDPG method which enables the agents to finish tasks in simulation such as cooperative communication, cooperative navigation (formation), physical deception and so on [3].

In the work of [2], four robots realize the formation control in both simulation and real-world environments by leveraging the multi-agent actor-critic algorithm.

3 Preliminaries

3.1 Brief Introduction of DDPG

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function Q*(s,a), then in any given state, the optimal action a*(s) can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a). \tag{1}$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted specifically for environments with continuous action spaces? It relates to how we compute the max over actions in

$$\max_a Q^*(s,a). \tag{2}$$

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

3.2 The Q-learning Method of DDPG

First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \mathrm{E}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right] \tag{3}$$

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set D of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi,\mathcal{D}) = \mathrm{E}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right] \tag{4}$$

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function.

3.3 Calculating the Max Over Actions in the Target

As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a target policy network to compute an action which approximately maximizes $Q_{\phi_{targ}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \mathrm{E}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))\right) \right)^2 \right], \tag{5}$$

where $\mu_{\theta_{targ}}$ is the target policy.

3.4 The Policy Learning Side of DDPG

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \mathrm{E}_{s\sim\mathcal{D}} \left[Q_\phi(s, \mu_\theta(s)) \right]. \tag{6}$$

And the Q-function parameters are treated as constants here.
The pseudocode of DDPG is shown in Algorithm 1. [H] Deep Deterministic Policy Gradient [1] Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$ Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ} \leftarrow \phi$ Observe state s and select action $a = clip(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$ Execute a in the environment Observe next state s', reward r, and done signal d to indicate whether s' is terminal Store (s, a, r, s', d) in replay buffer $\mathcal{D}$ If s' is terminal, reset environment state. it's time to update however many updates Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$ Compute targets y(r,s',d) = r + $\gamma(1-d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$ Update Q-function by one step of gradient descent using $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$ Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$ Update target networks with $\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi$
$\theta_{targ} \leftarrow \rho\theta_{targ} + (1-\rho)\theta$ convergence

3.5 The Framework of MADDPG

Multi-Agent Deep Deterministic Policy Gradient(MADDPG) is an improved DDPG network. It requires the training of separate agents, and the agents need to collaborate under certain situations (like don't let the ball hit the ground) and compete under other situations (like gather as many points as possible). Just doing a simple extension of single agent RL by independently training each agent does not work very well because the agents are independently updating their policies as learning progresses. And this causes the environment to appear non-stationary from the viewpoint of any one agent. While we can have non-stationary Markov processes, the convergence guarantees offered by many RL algorithms such as Q-learning requires stationary environments.

The primary motivation behind MADDPG is that if we know the actions taken by all agents, the environment is stationary even as the policies change, since $P(s'|s, a1, a2, 1, 2) = P(s'|s, a1, a2) = P(s'|s, a1, a2,'1,'2)$ for any $i'i$. This is not the case if we do not explicitly condition on the actions of other agents, as done by most traditional RL algorithms.

In MADDPG, each agent's critic is trained using the observations and actions from all the agents, whereas each agent's actor is trained using just its own observations. This allows the agents to be effectively trained without requiring other agents' observations during inference (because the actor is only dependent on its own observations). Here is the pseudocode of maddpg. [H] episode = 1 to M Initialize a random process $\mathcal{N}$ for action exploration Receive initial state $\mathbf{x}$ $t = 1$ to max-episode-length for each agent i, select action $a_i =_{\theta_i} (o_i) + \mathcal{N}_t$ 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 $\mathcal{D}$ $\mathbf{x} \leftarrow \mathbf{x}'$ agent $i = 1$ to N Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$ Set $y^j = r_i^j + \gamma Q_i^{'}(\mathbf{x}'^j, a_1', \ldots, a_N')|_{a_k'=_k'(o_k^j)}$ Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$ Update actor using the sampled policy gradient:

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

Update target network parameters for each agent i:

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

3.6 Review of Formation Control

In formation control, the velocity change of each agent is determined by the its relative position with other agents. This can be represented as

$$\dot{x_i} = -\sum_{j \epsilon N^i} (x_i - x_j) \tag{7}$$

In DDPG, the training process is still based on the velocity and the distance between agents and goals, The velocity and the distance decide the reward of each agent's action. We will discuss this in detail in the following sections.

4 Fixed Point Control and Collision Avoidance using DDPG

4.1 Introduction to Robotarium

The Robotarium is a remotely accessible swarm robotics research platform where we are able to see our algorithms deployed on real robots [5]. In this section, we will test the performance of DDPG algorithm in multi-agent system to accomplish fixed point control while implementing collision avoidance. The environment we use is robotarium-python-simulator, a online simulator which is highly similar to the real robot system of Robotarium.

GRITSBot is used for multi-agent experiments of Robotarium. The robot runs a nonlinear velocity controller and a linear position controller. More specifically, the GRITSBot can be modeled using unicycle dynamics.

$$\dot{x} \quad = v\cos\theta \tag{8}$$
$$\dot{y} \quad = v\sin\theta \tag{9}$$
$$\dot{\theta} \quad = \omega \tag{10}$$

By controlling a point d in front of the robot offset by length l, the linear velocities of point are mapped to linear and rotational velocities(v, w) of the unicycle model:

$$v \quad = \cos(-\theta)v_x - \sin(-\theta)v_y \tag{11}$$
$$\omega \quad = \frac{1}{l}(\sin(-\theta)v_x + \cos(-\theta)v_y) \tag{12}$$

The transformation is calculated while $\dot{x}$ and $\dot{y}$ are input to single robot, and the robot will move according to the calculated linear and rotational velocities.

4.2 Experiment Setup

In this experiment, four identical robots were used to conduct fixed target movement. In the beginning, four agents were placed in random places of the Robotarium platform. The dimension of the platform is 3.2*2, and the coordinate of the center is (0,0). Then, we drived the four agents to four angles of a square: the target of agent1,agent2,agent3,agent4 is (0,-0.25), (0.25,0), (0,0.25), (-0.25,0). Also, we let the agents facing to the center of the square.

The targets of agent1,2,3,4 are (0,0.2), (-0.2,0), (0,-0.2), (0.2,0), both trying to find a optimal way to the opposite side of the square, as is shown in figure 1(a). The initial positions and targets are marked by circles of different color. Every episode contains at most 50 iterations, which means each agent will be given 50 control signals if there are no collision in the episode, otherwise the episode ends immediately. Our goal is to let the agents learn an optimal policy to both get closer to the target and avoid colliding with others.

4.3 Learning Protocols

To generalize the target-oriented policy of each agent, we use one DDPG network for the training of four agents. Every agent will store their experience in the memory batch, and they will share one set of network parameters while choosing actions based on certain observation.

The observation(current state) of an agent(Agent0) contains 8 variables: $d_1, a_1, d_2, a_2, d_3, a_3, a_{target}, a_{current}$, where $d_1, a_1,\ d_2, a_2$,d_3, a_3 represent the distances and direction angles between Agent0 and other agents, a_{target} represents the direction angle between the target position and Agent0 and $a_{current}$ represents the direction angle of Agent0 relative to world coordinates.

As for the action variable, we use only one action variable a_{action} as the output of DDPG, which represent the difference between navigation angle and a_{target}, ranging from $-\pi$ to π. The velocity of the agent is determined by the distance between the target and Agent0:

$$v \quad = \sqrt{(x_0 - x_{target})^2 + (y_0 - y_{target})^2} \tag{13}$$

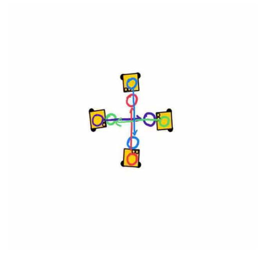
where x_0, y_0 represent the coordinates of Agent0, and x_{target}, y_{target} represent the coordinates of target. The direction of the velocity is determined by a_{action}.

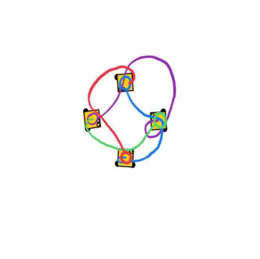
It is obvious that the reward is larger if a_{action} is closer to 0. However, if a_{action} is too close to 0, it may cause collision with other agents, which results in much less reward in the long term. Considering both cases, we set the reward function as follows: For Agent0, if the distance between Agent0 and any other agent is less than 0.13(slightly larger than the agent's diameter 0.11) the collision reward adds -5000, and the episode ends. The goal reward is given by:

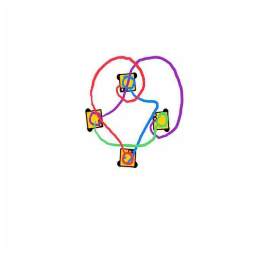
$$r_{goal} \quad = 1000 * (\pi/2 - |a_{action}|) \tag{14}$$

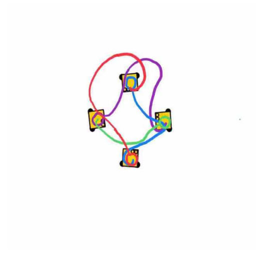
By adding collision reward and goal reward together, we can get final reward for single agent.

4.4 Experiment Results


(a) Agents and Goals


(b) Simulation 1


(c) Simulation 2


(d) Simulation 3

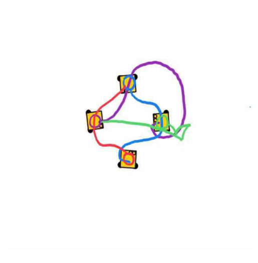
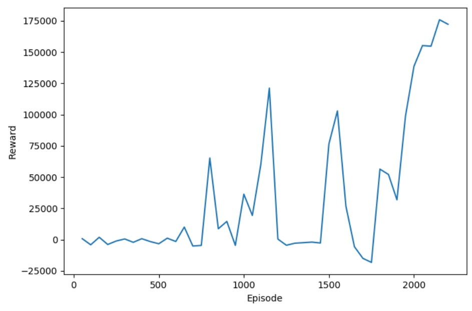
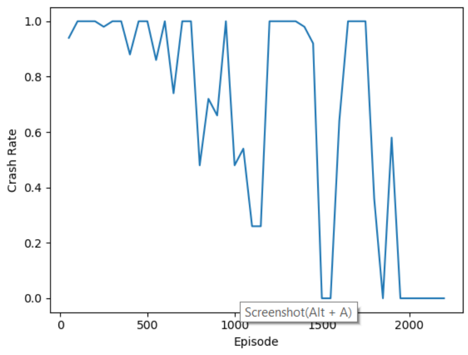

(e) Simulation 4

Figure 1: Simulation of four agents

After training of 2200 episodes, we got the results shown in Figure 1 (b),(c),(d),(e). The moving trajectories of each agent were depicted in lines of different color. The simulation process were recorded in a video, where all the agents could move towards the target point successfully without any collision. In Figure 2 (a),(b), the rewards of one episode grow larger with the passage of time, and get stable in the end. Also, the collision rate goes down to 0 finally.



(a) Rewards



(b) Crash Rate

Figure 2: Rewards and Crash Rate

4.5 Drawbacks of DDPG

By using DDPG, we could successfully accomplish the goal of fixed point control and collision avoidance. However, the case we have studied on is not that complicated. The position of the agents are fixed in the beginning rather than randomly distributed and we only conducted fixed target control rather than more complex control protocols. In terms of the performance of this experiment, as is shown in Figure 2, there are huge flunctuations during the learning process, which might take it long to reach higher reward. More importantly, the uncertainty in the training process would also have negative influence on the convergence of final results in the long term. Actually, in this experiment, it is likely that the reward drops after it gets 'stable' for a short time.

The limitations of DDPG might be caused by changing environment and the uncertainty of other agents' actions. DDPG could definitely behave well while the environment is stable, in this experiment, however, the agents have much 'misunderstanding' about the current situation. That's to say, the agents cannot adjust to the changing environment quickly. To enhance the performance of original DDPG algorithm, we have to take other agents' actions into account and make certain prediction of others' actions, which is the core idea of MADDPG.

5 Formation Control and Collision Avoidance using MADDPG

As we have found DDPG algorithm is challenged by multi-agent tasks, and the formation control may suffer from different initialized positions of the agents and avoiding obstacles in the environments, MADDPG is considered to be applied in our formation control and collision avoidance tasks.

5.1 Task Description

In this part we design four multi-agent tasks regarding formation. The first one is to let the agents to occupy the landmarks of equal number in the environment without colliding with each other. The positions of the agents and the landmarks are initialized randomly in each episode. The second one is to add obstacles in the first task setting, which means the agents need to reach the landmarks without collision with other agents or obstacles. The agents do not have to occupy a specific landmark and can choose their own.

The third task is to enable agents to form a shape, which is randomly initialized by specifying the edges between agents in each episode, and likely, the agents cannot bump into each other. The fourth task adds obstacles to the third task setting and the agents should also avoid those obstacles. Meanwhile the agents should not go out of the boundary.

5.2 Shaped Reward Function

In this project, we use the shaped rewards and propose the reward functions as follows:

$$\mathrm{R}(k) = \mathrm{T}(k) + \mathrm{C}_a(k) + \mathrm{C}_0(k) + \mathrm{B}(k) \tag{15}$$

where k represents the agent, $\mathrm{R}(k)$ is the reward function, $\mathrm{T}(k)$ is a task-specified cost function, $\mathrm{C}_a(k)$ represents a agent collision cost function activated when agents collide, $\mathrm{C}_0(k)$ represents a obstacle collision cost function activated when agents and obstacles collide, and B(k) is a boundary cost function activated when agents are close to the boundary.

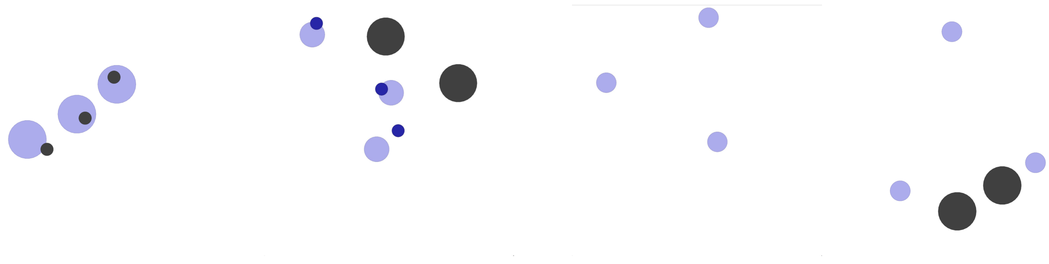
Specifically, the task-specified cost function for task 1 and 2 is:

$$\mathrm{T}_s(k) = -\sum_{l} min(\{d_{lk}\}_k^K), \tag{16}$$

where l is the landmark, k is the agent, K is the number of agent, and d_{lk} is the distance between landmark l and agent k. Then the task-specified cost function for task 3 and 4 is:

$$\mathrm{T}_f(k) = -\frac{1}{N}\sum_{j \epsilon N^k} |(||x_k - x_j|| - ||e_{kj}||)|, \tag{17}$$

where $||x_k - x_j||$ is the distance between two connected agents and $||e_{kj}||$ is the expected distance between them.



(a) Task 1 (b) Task 2 (c) Task 3 (d) Task 4

Figure 3: Experiment environment for the tasks

5.3 Experiment Setup

Here we leveraged the multi-agent environment engine from OpenAI. We created our own task scenario as shown in Figure 3. In Figure 3 (a), the blue ones are agents, and the black ones are landmarks. Figure 3 (b), the big blue ones are agents, the small ones are landmarks, and the black ones are obstacles. In Figure 3 (c), the blue ones are agents. In Figure 3 (d), the blue ones are agents and the black ones are obstacles.

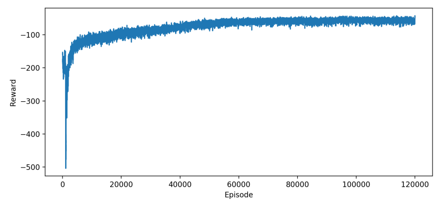
Here we chose three agents and two obstacles because they can well represented similar problems in a more understandable way, and they only require acceptable computational resources for a short-term project. Based on the reward function we designed and the MADDPG algorithm, we trained models for each task on a Macbook Pro with the cpu setting.

5.4 Results and Analysis

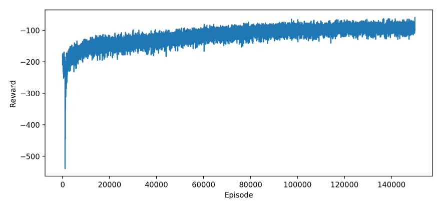
Figure 4 shows the rewards during training for each task. Task 1 was trained for 120000 episodes and the rewards were going up obviously. Task 2 was trained for 150000 episodes and the rewards also had a remarkable upwards trend. However, compared to rewards for task 1 training, those for task 2 developed with more fluctuations, and this should be due to the more complex environment and huger state space in task 2. For task 3 and task 4, both curves had deep troughs at the start of the training, and then grew slowly afterwards. The growing trends for task 3 stopped at an early stage and converged. However, we found that the policy for task 3 might converge sub-optimally, and we would like to explore the reasons in our future work.
As can be seen from Figure 5 (a) and (b), the learned policy for task 1 enables the agent to reach the goal very close, whereas the learned policy for task 2 is not that perfect. In Figure 5 (c), the blue line and orange line are consistent with the red lines after about 40 steps, while the red line has an obvious deviation from the target. In the case depicted in Figure 5 (d), the green one is more close to its own target than the blue one, but it oscillates around the red line.

6 Conclusion and Future Work

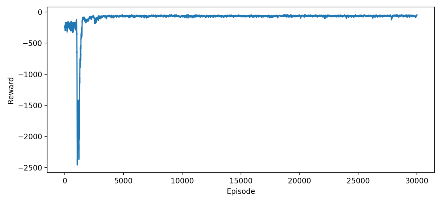
In this project, We designed formation control and collision avoidance tasks with shaped reward functions, and explored two policy gradient methods which can be applied in the multi-agent environment. The



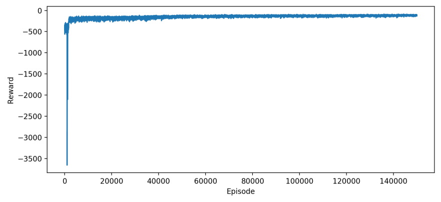
(a) Rewards for task 1



(b) Rewards for task 2

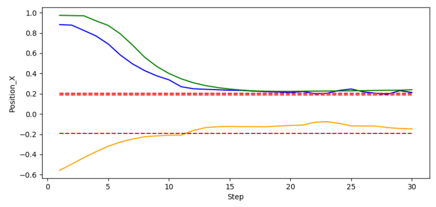


(c) Rewards for task 3

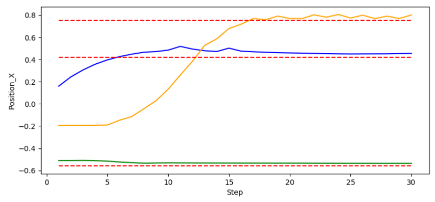


(d) Rewards for task 4

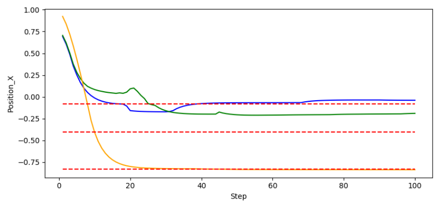
Figure 4: Rewards during training for the 4 tasks



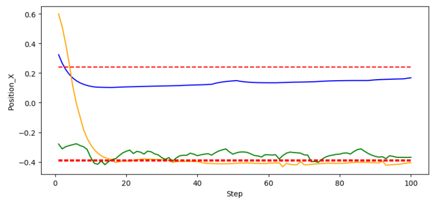
(a) Situation 1 for task 1



(b) Situation 2 for task 1



(c) Situation 1 for task 2



(d) Situation 2 for task 2

Figure 5: The trends of X positions of agents in one execution of learned policy for task 1 and 2. Red dash lines represent the target positions and the solid lines represent the X position of each agent.

experiments were implemented in two simulation environments. We found that DDPG can be used in a fixed point control task, however, it may work in limited situations and be of poor capacity of generalizing the policy to new environments. Designed for multi-agent system, MADDPG is more powerful in cooperative or competitive tasks like formation control and collision avoidance with several agents. However, MADDPG also has its drawbacks. For example, existing MADDPG techniques and applications may rely heavily on the rewards shaped by the human, and designing a good reward function can be challenging. Also, gym's simulation environment is much simpler than real world environment, such as Robotarium. So our future work can be focused on improving the performance of MADDPG and how to learn in a sparse-reward environment, and we could test our algorithm in Robotarium environment.

References

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[2] Qishuai Liu and Qing Hui. The formation control of mobile autonomous multi-agent systems using deep reinforcement learning. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–8. IEEE, 2019.

[3] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1699–1706. IEEE, 2017.

[6] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[7] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *the tenth international conference on machine learning*, page 330–337, 1993.