Implementing MADDPG on Multi Particle Environments

Yongtao Qu

Northeastern University
360 Huntington Ave, Khoury College of Computer Sciences,
Boston, Massachusetts 02115 USA
qu.yo@northeastern.edu

Abstract

This project aims to implement a widely used multi-agent reinforcement learning algorithm known as MADDPG. The implementation offers two network options: MLP and RNN. To evaluate its performance, the implementation was tested on the Multi Particle Environments (MPE) and compared against a similar implementation from AgileRL, a well-established reinforcement learning library. The results of the comparison indicate that this project outperforms AgileRL in terms of both learning time and stability. Additionally, it was observed that using an MLP in this project yields better results than employing an RNN in competitive tasks. Several potential reasons for these findings were discussed during the experiment.The code of this project is at this link

Introduction

Multi-agent reinforcement learning (MARL) is an extension of reinforcement learning that focuses on analyzing group behavior while keeping the core component unchanged, i.e., how an agent optimizes its strategy to obtain higher rewards. However, In multi-agent environments, traditional reinforcement learning algorithms like Q-Learning or policy gradient may not be effective due to various reasons such as changing policies during training, non-stationary environments, and high variance in coordination between agents. This makes MARL a challenging yet interesting area to explore.

Multi-agent tasks exhibit a high degree of diversity compared to single-agent tasks. In single-agent settings, diversity is generally limited to the action space, reward function, and observation dimension. In contrast, multi-agent tasks feature a range of diversifying elements, including different tasks, agent types and other additional information. For example, the task modes in multi-agent reinforcement learning can be broadly classified into two types: Cooperative-like, where agents tend to work as a team towards a shared goal, and Competitive-like, where agents have adversarial targets and exhibit aggression towards other agents.

In order to solve different types of tasks, this project choose to utilize MADDPG(Lowe et al. 2017), which is an actor-critic method and follows Centralized Training, Decentralized Execution(CTDE) framework. Its centralized critic make it applicable to all types of multi-agent tasks, including cooperative, collaborative, competitive, and mixed tasks, as opposed to Value Decomposition methods like QMIX(Rashid et al. 2018) that only able to solve cooperative tasks. To evaluate the performance of MADDPG, the project examines its performance using the Multi Particle Environments (MPE)(Mordatch and Abbeel 2017). MPE offers multiple maps, each catering to distinct tasks. This selection allows for a comprehensive analysis of MADDPG's performance across different scenarios.

Background

Dec-POMDP In this work, each agent is modelled as a Decentralized Partially Observable Markov Decision Processes (Dec-POMDPs) (Oliehoek, Amato et al. 2016), which is formally defined as $G =< I, S, A, P, R, \Omega, O, n, \gamma >$, where $I = \{1, 2, \cdots, n\}$ is the finite set of agents. $s \in S$ is the state of the environment from which each agent i draws an individual partial observation $\Omega_i \in \Omega$ according to the stochastic joint observation function $O(s, i) : A \times S \rightarrow \Delta\Omega$. At each timestep, each agent i selects an action $a_i \in A$, forming a joint action $\mathbf{a} \in A^n$, resulting in a shared reward $\mathbf{r} = R(s, \mathbf{a})$ for each agent and the next state s_0 according to the transition function $T : S \times A \rightarrow \Delta S$ that determines observation emissions $Pr(\mathbf{o}|\mathbf{a}, s)$, and γ is the discount factor.

Dec-POMDP is a more general approach for modeling sequential decision-making problems, especially in multi-agent settings. For instance, in a complex cooperative task, a group of agents may not have access to the entire system information, making it impractical for a single agent to solve the task. Instead, a more practical approach is to provide each agent with local observations to make decisions based on the current situation.

The main challenge lies in helping the agent build its belief system such that its decision-making aligns with its teammates and the system state towards achieving the final goal. To tackle this challenge, one of the most commonly used techniques is Centralized Training Decentralized Execution (CTDE), which we will discuss later.

Deep Q-Networks(DQN) Q-Learning and DQN(Mnih et al. 2013) are popular methods in reinforcement learning

and have been previously applied to multi-agent settings.The core idea behind DQN is to approximate the Q value function using a deep neural network rather than a table. The network takes the environment state as input and predicts the Q values for each possible action. By selecting actions with highest Q values, the agent can make informed decisions to maximize long-term rewards.

In practice, DQN introduces the concept of experience replay, which stores agent experiences in a replay buffer. During training, a batch of experiences is randomly sampled from the buffer. This has two advantages: first, it improves the utilization rate of the data and avoids the waste of discarding it only once each time; second, it breaks the temporal correlation of the data and reduces the instability of the training.

With its ability to handle high-dimensional state spaces and its strong performance in various domains, DQN has achieved remarkable results in challenging RL tasks, including playing Atari games, controlling robotic systems, and solving complex control problems. Q Learning can also be directly applied to multi-agent settings by having each agent i learn an independently optimal function Q_i . 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.

Policy Gradient(PG) Policy Gradient methods(Sutton et al. 1999) take a gradient-based approach to optimize the policy parameters. The underlying idea is to search for policy parameters that maximize the expected cumulative reward. The policy is typically represented by a parametric function, such as a neural network, with the parameters determining the behavior of the agent. Formally, Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right]$$

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy. The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

The optimization process involves iteratively updating the policy parameters using gradient ascent, aiming to increase the expected reward. This is achieved by estimating the gradient of the expected cumulative reward with respect to the policy parameters and performing updates in the direction of the gradient. The updates are typically guided by stochastic gradient ascent, where samples of state-action trajectories are used to estimate the gradient.

Actor-Critic(AC) As the name Actor-Criric (Konda and Tsitsiklis 1999) suggests, there are two networks in AC methods: the actor(policy network) interacts with the environment, and use policy gradient to learn a better policy with the guidance of critic; the critic(value network) aims to learn a value function based on the data from actor and the environment, and forth guide the actor to choose better action. Both actor and critic use policy gradient to update its parameters. DDPG is a classic algorithm of Actor-Critic. The process is as illustrated in Algorithm 1:

Algorithm 1: Actor-Critic algorithm

1: Initialized actor network parameter θ and critic network parameter ω
2: **for** episode = 1 to E **do**
3: Sample a trajactory based on current policy $\{s_1, a_1, r_1, s_2, a_2, r_2, \cdots\}$
4: Calculate for each step:$\delta_t = r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)$
5: Update value parameter $\omega = \omega + \alpha_\delta \sum_t \delta_t \nabla_\omega V_\omega(s_t)$
6: Update policy parameter $\theta = \theta + \alpha_\theta \sum_t \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$
7: **end for**

DDPG Deep Deterministic Policy Gradient (DDPG)(Lillicrap et al. 2015) is a popular reinforcement learning algorithm that can handle continuous action spaces. Typically, if known the optimal action-value function $Q^*(s, a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving $a^*(s) = \arg\max_a Q^*(s, a)$. 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. 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.

Instead, DDPG concurrently learns a Q-function and a policy network using off-policy data and the Bellman equation. The Q-function(critic) estimates the expected cumulative reward for taking a specific action in a given state, and the policy network(actor) produces actions to maximize the Q-value. Specifically, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent: $\mathrm{L}(\phi, \mathcal{D}) = E_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s, a) - (\tau + \gamma(1-d)Q_{\phi_{\max}}(s', \mu_{\theta_{\max}}(s')))\right)^2\right]$ where $\mathcal{D}$ is the replay buffer, $\mu_{\theta_{targ}}$ is the target policy. 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

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

Note that the Q-function parameters are treated as constants here.

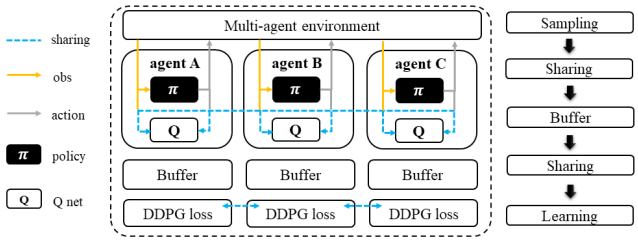


Figure 1: Workflow of MADDPG

MADDPG Multi-agent Deep Deterministic Policy Gradient (MADDPG)(Lowe et al. 2017) is one of the multi agent versions of DDPG algorithm. Instead of using DDPG for each agent independently, MADDPG extends the DDPG algorithm with a centralized Q function that takes observation and action from all agents. All agents now share a centralized critic network that simultaneously guides each agent's actor network during the training process, and each agent's actor network acts completely independently, which is called decentralized execution. The workflow of MADDPG is in Figure 1.

More detailedly, MADDPG algorithm has each agent trained with the Actor-Critic method, but unlike the traditional single agent, the Critic part of each agent in MADDPG can obtain the strategy information of other agents. Specifically, if we consider a game with N agents, each agent's policy parameter is $\theta = \{\theta_1, ..., \theta_N\}$, and $\pi = \{\pi_1, ..., \pi_N\}$ is the set of strategies for all agents, then we can write the policy gradient of each agent's expected return in the case of a random strategy:$\nabla_{\theta_i} J(\theta_i) = E_{s\sim p^{\mu}, a\sim\pi_i}[\nabla_{\theta_i} \log \pi_i(a_i \mid o_i) Q_i^{\pi}(\mathbf{x}, a_1, \ldots, a_N)]$. $Q_i^{\pi} = (\mathbf{x}, a_1, ..., a_N)$ is a centralized action value function. In general, $\mathbf{x} = (o_1, ..., o_N)$ contains the observations of all agents, and Q_i must also contain the actions of all agents at that moment, so the premise of Q_i is that all agents must give their own observations and corresponding actions at the same time. The centralized action value function can be updated with the following loss function:

$$\mathcal{L}(\omega_i) = E_{\mathbf{x},a,r,\mathbf{x}'}\left[\left(Q_i^{\mu}(\mathbf{x}, a_1, \ldots, a_N) - y\right)^2\right],$$

$$y = r_i + \gamma Q_i^{\mu'}(\mathbf{x}', a_1', \ldots, a_N')\Big|_{a_j'=\mu_j'(o_j)}$$

$\mu' = \left(\mu_{\theta_1}', \ldots, \mu_{\theta_N}'\right)$ is the set of target policies used in the update value function, which have parameters for deferred updates. The general process of MADDPG is in Algorithm 2

Algorithm 2: MADDPG algorithm

```
Randomly initialized actor and critic net for each agent
for episode = 1 to E do
    Initialize a random process N for action exploitation
    Get joint observation o
    for t = 1 to T do
        For each agent, choose action based on current policy a_i = μ_{θ_i}(o_i)
        Run action a, get reward r and new joint observation o'
        Store (o, a, r, o')into replay buffer D
        Sample experiences from replay buffer D
        For each agent, centralized training critic
        For each agent, decentralized training actor
        Soft update all actor and critic networks
    end for
end for
```

Related Work

CTDE The CTDE framework, which stands for Centralized Training Decentralized Execution, is a widely used approach in multi-agent reinforcement learning. In this setting, agents are trained together in a centralized manner where they can access all available information, including the global state, other agents' status, and rewards. However, during the execution stage, agents are forced to make decisions based on their local observations, without access to centralized information or communication.

CTDE strikes a balance between coordination learning and deployment cost, making it a popular framework in MARL. Since multi-agent tasks involve numerous agents, learning a policy that aligns with the group target requires incorporating extra information from other sources. Thus, centralized training is the preferred choice. However, after training, the delivery of centralized information is too costly during deployment, leading to delays in decision-making. Furthermore, centralized execution is insecure, as centralized information can be intercepted and manipulated during transmission.

Centralized Critic Centralized critic-based algorithms are applicable to all types of multi-agent tasks, including cooperative, collaborative, competitive, and mixed. These algorithms use a centralized critic to approximate the state-action value function, which enables agents to learn a policy that considers the actions of other agents and the global state. The classic centralized critic methods are MADDPG(Lowe et al. 2017), MAPPO(Yu et al. 2021), etc.

Value decomposition On the other hand, value decomposition-based algorithms can only be applied to cooperative and collaborative scenarios. These algorithms use a value decomposition technique to decompose the value function into individual value functions, one for each agent. The agents then learn their own policies based on their individual value functions. Since value decomposition-based algorithms do not use a centralized critic, they cannot be applied to competitive scenarios where the agents' objectives conflict. Therefore in this project, we choose to use the MADDPG to solve competitive envs. The classic value decomposition methods are VDN(Sunehag et al. 2017), QMIX(Rashid et al. 2018), etc.

Project description

This section provides a detailed explanation of the implementation of MADDPG in this project

Network The project offers two options for the base network used in both the actor and critic components: MLP and RNN. MLP is made of a three-layer simple neural network. and use relu as activation function, which is suitable for most tasks. RNN represents a recurrent neural network model, which utilizes a series of linear transformations and a recurrent cell, such as the GRUCell (used in this project), to generate the output. RNN excels at capturing temporal dependencies and are particularly beneficial in RL tasks that involve sequential information.

Replay buffer The implementation of the Replay Buffer in this project is crucial for effective data storage and retrieval in Reinforcement Learning. The basic principle behind the Replay Buffer is to store the agent's experiences in the environment within a buffer and randomly sample a subset of the data from the buffer during training.Since MADDPG is an off-policy algorithm, the utilization of a replay buffer becomes necessary. In this project, a circular queue of fixed size is implemented for the replay buffer. When the queue reaches its maximum capacity, the oldest data is automatically discarded to make room for new experiences. Each transition within the replay buffer is defined as a namedtuple '(state, action, reward, next_state, done, hidden_state)'. The 'state' represents the current joint state observed by the agent, 'action' denotes the action taken in response to that state, 'reward' signifies the immediate reward received, 'next_state' signifies the subsequent state, and 'done' indicates whether the episode has terminated. Additionally, for RNN-based networks, the 'hidden_state' is also stored within the transition tuple to retain the necessary information for RNN processing.

Gumbel-Softmax In MPE, each agent operates within a discrete action space. Even though MADDPG is intended for continuous action space, which can make the agent's actions derivable for its strategy parameter, it can be applied to discrete action space by using Gumbel-Softmax to get an approximate sample of the discrete distribution.

Suppose a random variable Z that follows a discrete distribution $K = (a_1, ..., a_k)$, with $a_i \in [0, 1]$ means $P(Z = i)$ and $\sum_{i=1}^{k} a_i = 1$. If we want to sample according to this distribution $z \sim K$, we can see that sampling this discrete distribution is not derivable.

One way to make discrete sampling controllable is heavy parameterization, and the Gumbel-Softmax technique is used here. Specifically, we introduce a heavy parameter factor g_i, which is a noise sampled from $Gumbel(0, 1)$:

$$g_i = -\log(-\log u), u \sim Uniform(0, 1)$$

Gumbel Softmax samples can be written as:

$$y_i = \frac{exp((loga_i + g_i)/\tau)}{\sum_{j=1}^{k} exp((loga_j + g_i)/\tau)}, i = 1, ..., k.$$

In this case, if the discrete value is calculated by $z = \arg\max_i y_i$, the discrete value is approximately equivalent to the value sampled by the discrete $z \sim K$. Furthermore, the gradient for a is naturally introduced into the sample result y. $\tau > 0$ is called the temperature parameter of the distribution, and by adjusting it one can control how similar the generated Gumbel-Softmax distribution is to the discrete distribution: the smaller τ, the more the resulting distribution tends to be the result $onehot(\arg\max_i loga_i + g_i)$; the larger τ, the more the resulting distribution tends to be uniformly distributed.

DDPG and MADDPG First we define a DDPG class, each with a actor, a critic, and two target networks based on the predefined network arch. Adam is chosen for optimizers as it is proven to be better than RMSProp in a research on the implementation tricks of SMAC environment(Hu et al. 2021). DDPG uses some tricks from DQN, like target network and soft-update,i.e. the target network is updated slowly and gradually approaches the network rather than directly updated, and its formula is

$$w^- \leftarrow \tau w + (1 - \tau) w^-$$

Usually τ is a relatively small number, and when $\tau = 1$ it is consistent with how the DQN is updated. The target network u also uses this soft update method. In addition, due to the problem of overestimating the Q value of the function Q, DDPG uses the technology in Double DQN(van Hasselt, Guez, and Silver 2015), which is to choose actions from the original network rather than the target network to update.

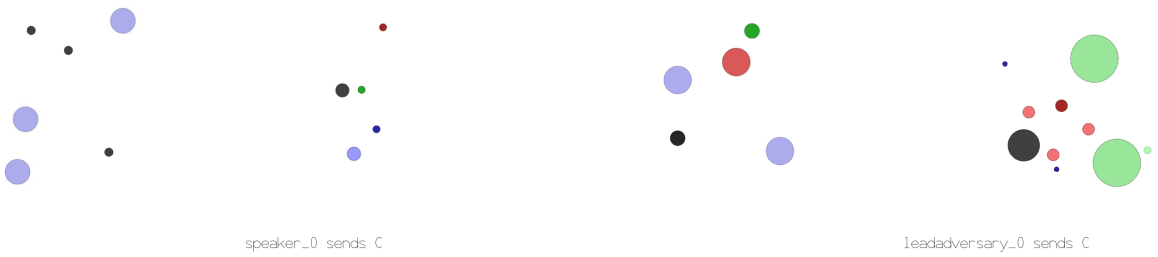
However, because DDPG uses a deterministic strategy, its own exploration is still very limited. In the DQN algorithm, its exploration is mainly generated by the behaviour of the greedy strategy. Also as an algorithm for offline strategies, DDPG introduces a random noise on the behavioural strategies to explore. When taking actions with a policy network, we add Gaussian noise to the actions for better exploration. In the original DDPG paper, the added noise corresponds to the Ornstein-Uhlenbeck (OU) stochastic process:

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

where u is the mean, W is random noise following Brownian motion, and θ and σ are scale parameters. It can be seen that $x(t - 1)$ when deviating from the mean, the value of x_t moves closer to the mean. the OU stochastic process is characterised by a linear negative feedback around the mean with an additional disturbance term.

Experiments

In this section, we provide experimental results comparing various implementations of MADDPG. The evaluation is conducted on the Multi Particle Environments(MPE) (Mordatch and Abbeel 2017) with four representative environments. MPE are a set of communication oriented environment where particle agents can (sometimes) move, communicate, see each other, push each other around, and interact with fixed landmarks. MPE have agents which are partially observable and have different types of environments including cooperative, competitive and mixed tasksThis variety enables comprehensive testing of MADDPG's performance across different types of multi-agent tasks.



(a)Simple Spread) (b)Simple Speaker Listener (c)Simple Adversary (d)Simple World Comm

Figure 2: Four environments used in project

Specifically, we choose four representative environments, which are shown graphically in Figure 2:

- Simple Spread, which is a cooperative environment with N agents, N landmarks and agents must learn to cover all the landmarks while avoiding collisions.
- Simple Speaker Listener, which is a cooperative environment with 2 agents and 3 landmarks of different colors. One agent called Listener wants to get closer to the target landmark, which is known only by the other agent, called Speaker.
- Simple Adversary, which is a competitive environment with 1 adversary, N good agent and N landmarks. One landmark is the 'target landmark'. Both teams are rewarded based on how close the closest one of them is to the target landmark, but negatively rewarded based on how close the adversary is to the target landmark. However, the adversary has no idea which landmark is the target landmark.
- Simple World Comm, which is a mixed environment with predator-prey settings: good agents are faster and receive a negative reward for being hit by adversaries. Adversaries are slower and are rewarded for hitting good agents. Also some landmarks with different utilities in the map.

Three implementations of MADDPG were evaluated in this study: the MLP version, the RNN version, and the implementation from AgileRL, which is a well-developed deep reinforcement learning library focused on improving development by introducing RLOps - MLOps for reinforcement learning, as the baseline. The experiment results are shown in Figure 3, 4, 5 and 6. The analysis of the results focuses on two aspects: comparing the performance of this project's implementations with AgileRL's implementation and assessing the impact of network architecture choices within this project.

Since two projects have different configurations, it is unfair and unreasonable to compare them based on the number of episodes. Therefore, the results are presented separately for this project's implementation (denoted as (a) in each figure) and AgileRL's implementation (denoted as (b) in each figure), with a grey line representing the converged value is included to facilitate comparison.

It is important to note that this project utilizes simpler frameworks, resulting in significantly shorter training processes. Despite the differences in training duration, both projects achieve convergence to approximately the same optimal solution across all four maps, indicating successful optimization. However, it is obvious that AgileRL's project is significantly more volatile, as evidenced by the wide range of standard deviation plotted. The agents experience large fluctuations in returns throughout the training process and even a large dip in Figure 5(b).

One possible explanation for this volatility is that the MPE tasks may not be particularly challenging, allowing simpler network architectures to perform well. On the other hand, AgileRL focuses on reducing the time taken for training models and hyperparameter optimization (HPO) by pioneering evolutionary HPO techniques for reinforcement learning, which is much more complex to train and converge with tremendous hyperparameters, but making it possible to solve much difficult tasks probably.

When comparing the MLP and RNN implementations, it is more informative to present them on the same figure since they share the same configuration except for the network architecture. Note that Figue 6 (a) only shows the curve of lead adversary instead of all adversaries to maintain clarity in the figure.

In cooperative environments such as simple spread and simple speaker listener, both MLP and RNN agents demonstrate fast and stable convergence. They perform equally well and reach the optimal value. In mixed environments like simple adversary and simple world comm, the RNN agents exhibit poor performance. Even fail to converge to the optimal value and experience significant fluctuations in simple adversary. In contrast, the MLP agent performs just as effectively as it did in the cooperative environments.

One possible explanation for this discrepancy is that in competitive scenarios, where agents aim to outsmart each other, the environment becomes more dynamic and unpredictable. The optimal action at any given time may depend not only on the historical actions but also on anticipating and reacting to the opponent's future moves. In such competitive scenarios, relying too heavily on past information, as done by the RNN, can be detrimental. Opponents may exploit predictable patterns learned by the RNN, leading to suboptimal

performance. MLPs, on the other hand, focus more on the current state and may be less susceptible to such exploitation. In addition, in this project, the implementation of RNN is to store hidden states in the replay buffer. This can lead to a problem known as 'Representational Drift'(Kapturowski et al. 2018), where, after a couple of epochs, the hidden state stored might no longer correspond to the hidden state the network would produce when feeding the same input. While this has little impact on cooperative environments where both agents work together towards the same goal, it can become a serious problem when adversaries taking unpredictable actions.

Conclusion

This project has implemented MADDPG with both MLP and RNN network architectures on the Multi Particle Environments. By comparing the results with the implementation in AgileRL, several possible reasons and guidelines for choosing the type of network have been identified.

In the future, that can be further developed: 1) Adding support for CNN network architecture: This would make the code compatible with other pettingzoo tasks, such as space invader. 2) Addressing the issue of representational drift in RNN implementations: Since storing hidden states can lead to representational drift, alternative approaches can be explored. For example, using a zero start state where the hidden state is reset to zero during training, or allowing the network a burn-in period where it can run for a certain number of time steps and produce its own hidden state before training begins. 3)Implementing an episodic replay buffer: This type of replay buffer would enable sampling of complete episodes rather than random transitions from different episodes. This is particularly useful when using zero start hidden state and histories, as it allows for preserving the temporal continuity of the episodes. 4) Comparing with more baselines, which was intended to but failed: the original MADDPG repository is archived; the AgileRL only supports for evolvable CNN and MLP networks; the MARLLib has its code too messy and coupled, making it impossible to edit; and the Xuance is too new that only supports for one MPE scenario. 5) Trying more challenging tasks like MAgent which involve a large number of agents on each side, support for global state, and the potential for heterogeneous actions among agents.

References

Hu, J.; Jiang, S.; Harding, S. A.; Wu, H.; and wei Liao, S. 2021. Rethinking the Implementation Tricks and Monotonicity Constraint in Cooperative Multi-Agent Reinforcement Learning.

Kapturowski, S.; Ostrovski, G.; Quan, J.; Munos, R.; and Dabney, W. 2018. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*.

Konda, V.; and Tsitsiklis, J. 1999. Actor-critic algorithms. *Advances in neural information processing systems*, 12.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv e-prints*, arXiv:1509.02971.

Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, P.; and Mordatch, I. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv e-prints*, arXiv:1706.02275.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, arXiv:1312.5602.

Mordatch, I.; and Abbeel, P. 2017. Emergence of Grounded Compositional Language in Multi-Agent Populations. *arXiv preprint arXiv:1703.04908*.

Oliehoek, F. A.; Amato, C.; et al. 2016. *A concise introduction to decentralized POMDPs*, volume 1. Springer.

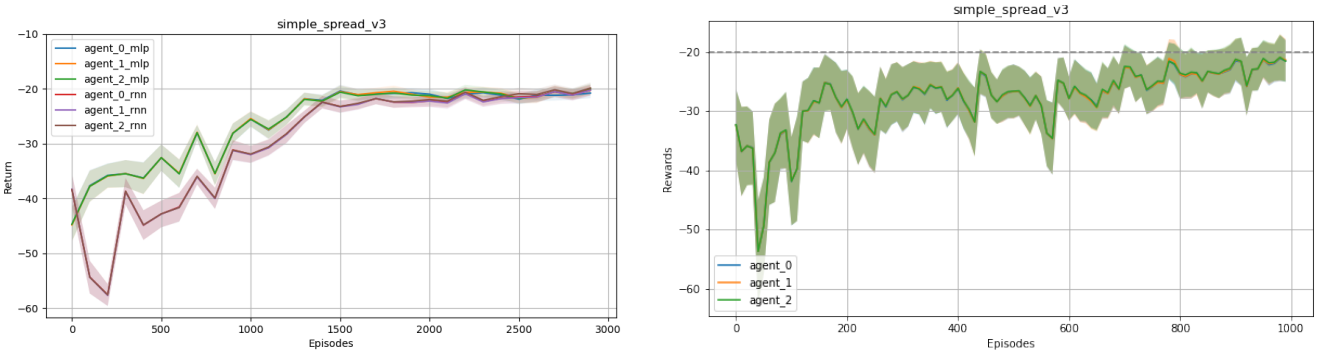
Rashid, T.; Samvelyan, M.; Schroeder de Witt, C.; Farquhar, G.; Foerster, J.; and Whiteson, S. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. *arXiv e-prints*, arXiv:1803.11485.

Sunehag, P.; Lever, G.; Gruslys, A.; Czarnecki, W. M.; Zambaldi, V.; Jaderberg, M.; Lanctot, M.; Sonnerat, N.; Leibo, J. Z.; Tuyls, K.; et al. 2017. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12.

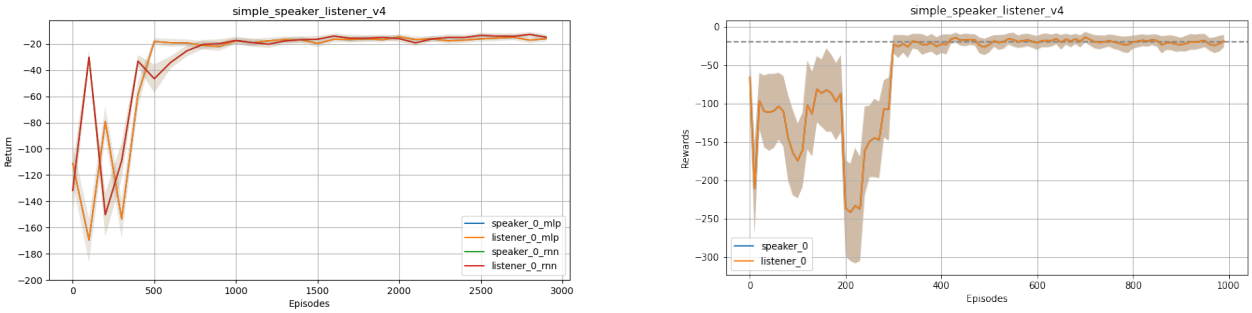
van Hasselt, H.; Guez, A.; and Silver, D. 2015. Deep Reinforcement Learning with Double Q-learning. *arXiv e-prints*, arXiv:1509.06461.

Yu, C.; Velu, A.; Vinitsky, E.; Gao, J.; Wang, Y.; Bayen, A.; and Wu, Y. 2021. The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games. *arXiv e-prints*, arXiv:2103.01955.



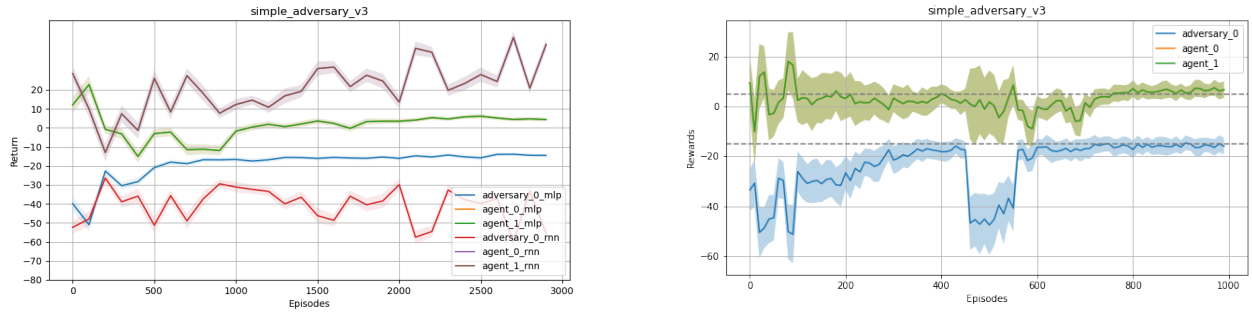
(a)This project (b)AgileRL

Figure 3: Performances of different MADDPG implementations in Simple Spread environment



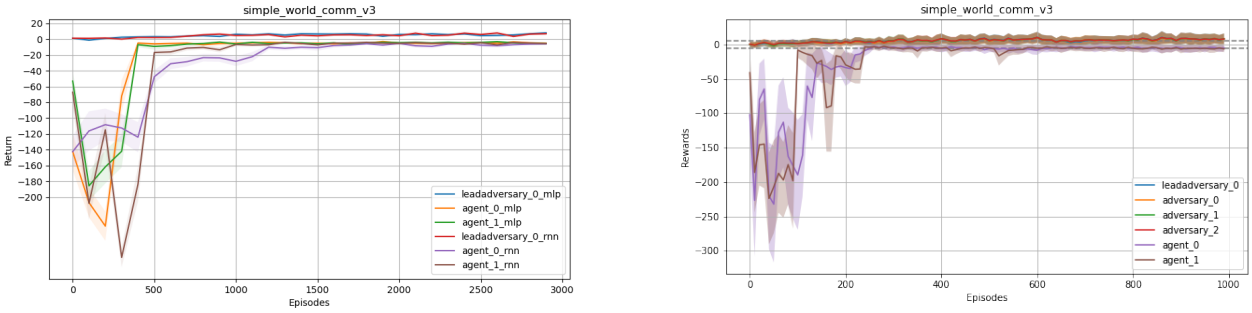
(a)This project (b)AgileRL

Figure 4: Performances of different MADDPG implementations in Simple Speaker Listener environment



(a)This project (b)AgileRL

Figure 5: Performances of different MADDPG implementations in Simple Adversary environment



(a)This project (b)AgileRL

Figure 6: Performances of different MADDPG implementations in Simple World Comm environment