Deep Reinforcement Learning Nanodegree
Project 3 – Multi-Agent RL

Chris Cadonic
chriscadonic@gmail.com

April 3, 2021

1   Introduction

As part of this the deep reinforcement learning nanodegree program, this report discusses my work in solving a mult-agent environment through the use of an algorithm that leverages the idea of multiple interacting agents in a learning environment.

1.1   Environment Overview

The environment, called *Tennis*, is a multi-agent learning situation wherein a pair of agents are playing against each other in a game of virtual table tennis. That is, each agent controls a virtual racket that can hit a ball back and forth over a net, if Thus, the task involves two agents that are simultaneously learning how to play tennis against an opponent, which introduces the issue of coordination and competition between agents.

For this task, the environment can be summarized as follows:

MDP property	characteristics
observation space	8-vector of continuous values representing position and velocity of the ball
action space	2-vector of continuous values with one value representing amount of horizontal movement for the racket, and how high to jump
rewards	+0.1 if the agent hits the ball over the net, and -0.01 if it hits the ball out of bounds or the ball hits the ground on its side of the table

Table 1: Characteristics of the MDP in the Reacher environment.

With these characteristics, each agent can learn how to move appropriately so as to reliably hit the ball over the net before it hits their side of the table, all while not hitting the ball too far and having it go out of bounds. Thus, the agent must learn how to adapt from the observation space of the ball and its current location to know how to move towards the ball at the right moment to hit it back to the other agent. A quick example image showing the game being played is shown below in Figure. **??**, where one agent is controlling the blue racket and the other agent is controlling the red racket.

Figure 1: A snapshot of the multi-agent Unity-ML tennis environment. The blue racket and red racket agents are tasked to hit the orange ball over the net to the other agent as reliably as possible to keep the ball in play together.

Since this is a multi-agent RL environment, there are two agents simultaneously learning from *independent* representations and action computations (i.e., internal policies). Each agent must be

able to keep the ball in play for as long as possible, which translates into an episode score that is represented by the *maximum score accumulated by either agent* over that episode. Since this is in effect a somewhat coordinated environment, being able to ensure both agents learn effectively is captured by the goal:

- achieving a 100-episode average max agent score of +0.5.

2 Approach

The overall approach used herein was to utilize the *Multi-Agent Deep Deterministic Policy Gradient* (MADDPG) algorithm, which is an extension of the DDPG algorithm to a multi-agent environment. This algorithm is explained in more detail in the following section.

2.1 MADDPG

This algorithm is an actor-critic algorithm that adapts Q-learning for a continuous action space, wherein both a Q-function approximation is learned alongside a serially adapted policy. This is similar to the DDPG base algorithm wherein target networks are utilized in learning both an *actor* network and a *critic* network by drawing from experiences in an expanding experience replay buffer. The primary extension of the DDPG algorithm into the multi-agent space is by now training agents using *centralized planning with decentralized execution*. That is, each agent takes in global information about what all agents are experiencing and what actions they take, but each agent has an actor that only leverages its own local information to actually make actions, which is particularly true throughout testing [1]. This helps extend DDPG to multi-agent learning environments in a way that minimizes the non-stationarity that occurs when utilizing DDPG for each agent independently without modification.

Similar to the baseline DDPG algorithm, the *actor* network is responsible for leveraging the approximation of the Q-function to determine an optimal action per time step. This is done by selecting actions as a continuous maxima on the Q and utilize gradient ascent.

$$\max_{\theta} \mathbb{E}_{s \sim D} \left[Q_\phi \left(s, \mu_\theta(s) \right) \right] \tag{1}$$

Additionally, the *critic* network is only concerned with improving the Q-function approximation, whereby the network evaluates the value of taken actions against the value of predicted actions suggested by the *actor* network. This is very similar from the DQN algorithm, and thus follows a valuation from:

$$Q^*(s, a) = \mathbb{E}_{s' \sim P} \left[r(s, a) + \gamma \max_{a'} Q^*(s', a') \right] \tag{2}$$

for evaluating a Q-function utilizing the Bellman equation for capturing reward and current max estimations on states and actions from the *critic* and *actor* networks.

Similar to DDPG, noise values are added to the actions of each agent to maintain a level of exploration throughout training, specifically by adding Ornstein-Uhlenbeck noise [2]. During application of the trained models, however, not only are actors now selecting actions using only local information but noise is also removed.

2.2 Implementation

The specific implementation of MADDDPG completed in this work largely takes on the same form as the pseudocode in Lowe et al. [1].

The implementation takes on the following characteristics:

- both *actor* and *critic* networks utilized a baseline and target network pairing during training, where target networks are updated using a per-step soft-update schedule with ratio τ,

- the *actor* network seemed to benefit from a simpler architecture in comparison to the *critic* network,

- using a repeatable update schedule per step (i.e., the networks can be updated m times per step) along with learning rate α,

- the *critic* for each agent learned from global observation and action information, whereas the *actor* network for each agent only utilized locally available information to that agent.

Using these implementation details on top of the MADDPG algorithm, I settled on the following hyperparameters:

hyperparameter	utility	value
α_{actor}	*actor* learning rate	0.001
α_{critic}	*critic* learning rate	0.001
τ	target network soft-update rate	0.001
γ	discount factor on future returns	0.99
t update	number of steps to take before updating networks	2
num updates	number of updates to complete each time networks are updated during training	5
μ	regression mean for Ornstein-Uhlenbeck noise	0.0
θ	factor for weighing the delta of value from mean	0.15
σ	factor for weighing the Weiner (Gaussian) noise process	0.2

Table 2: Hyperparameters experimented with to train an agent using MADDPG.

Values were either determined experimentally, such as by varying learning rate α, τ, γ, and the integers for the update schedule, or utilized from the first description of leveraging Ornstein-Uhlenbeck noise in the DDPG algorithm [2] for μ, θ, and σ.

Lastly, smaller architecture sizes seemed to produce better results for learning, where larger networks tended to show more tendency for residual performance dropoff once solved. Though the smaller networks tended to show this same pattern to some degree, they were able to learn how to consistently hit the ball nearly as well with more stability upon reaching that $+0.5$ solved average. For a bit more detail on the architectures utilized in these models, they are outlined below in Figure 2.
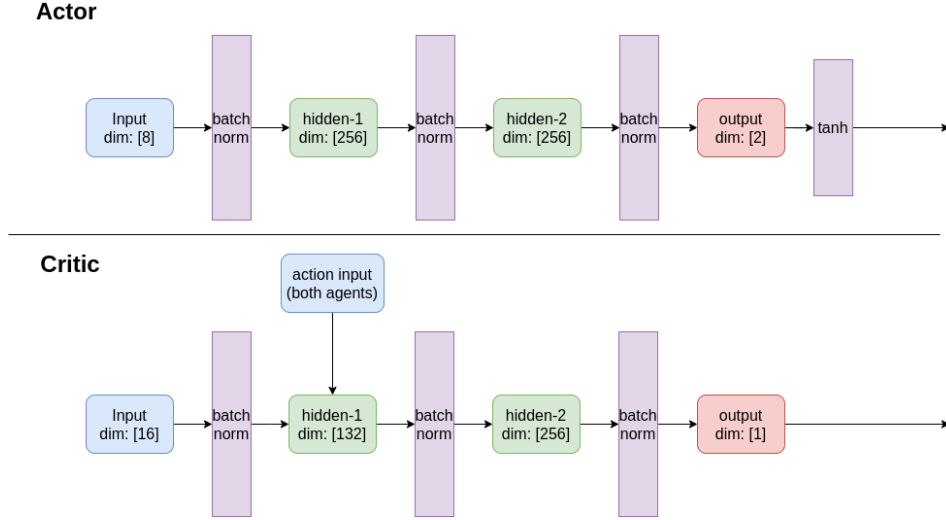
Figure 2: The architectures used in the final *actor* and *critic* networks in the MADDDPG algorithm. Though not specified, all activation functions are ReLU unless otherwise noted. Critic networks utilized global information and thus took in observation and action data from both agents, where actor networks only utilized the observations of an agent itself.

3 Results and Discussion

3.1 Learning Performance

After determining an optimal set of parameters by running hyperparameter search on $\alpha \in \{0.0001, 0.001, 0.05\}$ for both *actor* and *critic* learning rates, determining stable architectures for hidden layers across both *actor* and *critic* networks, and manually hyperparameter tuning the number of episodes to skip before training and the number of updates per update episode, the following results show training performance over a maximum 4000 episode training session on different major versions of experiments.
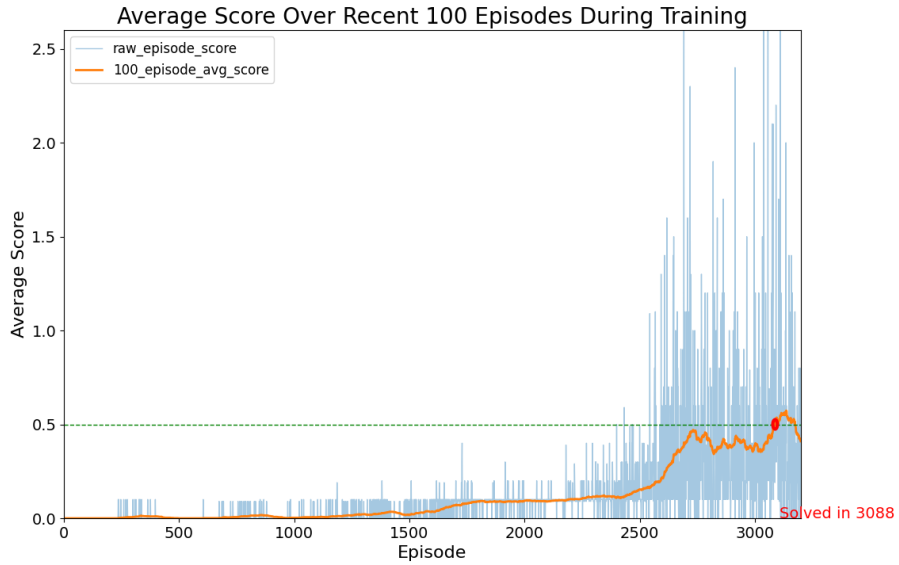

Figure 3: Per episode and 100-episode average score for episode scores across both agents (max of both agents for an episode). The environment was solved by both agents by achieving a 100-episode average score of +0.5 over 3088 episodes of training.

The results in Figure 3 illustrate the ability of the MADDPG algorithm to learn to solve the tennis environment in 3088 episodes when leveraging experiences from both agents for *critic* updates and only local agent information for *actor* updates. After all hyperparameter experimentation and other types of modifications to the algorithm, some improvements that seemed to matter most for achieving more stability in training as shown in Fig. 3 were simpler architectures, and maintaining a static Ornstein-Uhlenbeck noise process throughout training (i.e., decaying the noise actually resulted in more variable training).

Though the agents were able to learn the task well, the stability of learning still left some room for improvement, however, as training proceeded quite rapidly after a certain number of episodes were reached (2400 epsiodes). This may be due to the nature of the experience replay buffer cycling out low-reward episodes from early in training. Unclear, however, is why training shows a fair amount of instability around the target episode average of $+0.5$. Once the agents achieve a score of approximately $+0.4$, training tends to see more diminishing returns and some regression at times. This may be due to the changes in reward values that the updates to *critic* and even *actor* networks are seeing in later episodes, particularly when there are sporadic high reward episodes ($+2.6$). Despite these observed patterns, training appears to continue to generally improve throughout. More rigorous hyperparameter tuning and/or experimentation with other multi-agent algorithms may prove for more avenues to show better training performance against the MADDPG results shown herein.

3.2 Next Steps

Some obvious next steps were to complete an intended comparison of MADDPG to other multi-agent potential algorithms such as a multi-agent Q-learning agent useful for coordination tasks EAQR [3]. Beyond this, some additional next steps that were thought of to possibly introduce improvement if experimented with appropriately included:

- experimenting with different noise processes, specifically examining how effective an explicit Gaussian noise process would be,

- experimenting with different parameters for the Ornstein-Uhlenbeck noise process, particularly for θ and σ for controlling the variance of injected noise,

- experimenting with the idea of introducing differences between the *actor* and *critic* network architectures, thereby introducing more fine-tuned abilities of the networks to capitalize on more quickly learning a policy by learning from actions and values somewhat independently, this may be important especially for capturing aspects of learning from global information (i.e., *critic* learning) than only local information,

- some level of prioritization and weightings by using a prioritized experience replay buffer [4], which could help ascertain if issues with stability were in fact due to the nature of experiences sampled from the buffer and how more consistent higher reward experiences affected training stability.

With these potential improvements, there are many avenues to still experiment further with MADDPG before moving on to another algorithm. Improvements that are relevant for DDPG may also be feasible for exploration in this logical extension of DDPG into MADDPG as well.

4 Conclusion

With the results shown herein, agents were trained to coordinate their actions to keep a ball passing back and forth for as long as possible using the MADDPG algorithm. This algorithm was able to extend the DDPG algorithm to help agents leverage global information to uncover value, but only act on using local information available to each agent. Though the agents were able to learn the

task quite well, there did appear to be some issues with stability as well, and thus show that there are other areas that can be experimented with further to gain some more improved training results.

References

[1] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275*, 2017.

[2] 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.

[3] Zhen Zhang and Dongqing Wang. Eaqr: A multiagent q-learning algorithm for coordination of multiple agents. *Complexity*, 2018, 2018.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.