DRL-Collaboration and Competition - MADDPG Learning Algorithm - Description of the Implementation.

This project was done as part of the Udacity Deep Reinforcement Learning Nanodegree. MADDPG (Multi-Agent Deep Deterministic Policy Gradient) algorithm was used to teach two agents to collaborate and play tennis in a Unity Tennis environment.

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

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

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

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

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

Introduction

Goal of AI is to solve problem of intelligence. Intelligence is the result of interaction with other agents (which include humans). As Gerhard Weiß put it: “Intelligence is deeply and inevitably coupled with interaction" [94]. The multi agent scenario is a very complex environment, because agents trying to achieve their goals while learning **simultaneously** and **interacting** with **one-another**. The interaction between agents consists of computation, communication, coordination, competition, negotiation, prediction and so on. This motivates/leads to expand our horizon from using single agent to multi-agent systems. One of benefits/advantages of multi-agent systems, that can be exploited, is the fact that agents can **share** their **experience,** making each other smarter, in a **parallel** manner. This requires communication capabilities through software and hardware. Another benefit would be, **robustness** through redundancy. And, since redundancy is easily achieved through cloning/copying in case of failure, as a by-product, **scalability** is also achieved. Most multi-agent systems allow insertion of new agents easily, which inevitably leads to a more complex system than before. These are the benefits/advantages that can be exploited

To solve the Unity Tennis environment challenge, the MADDPG algorithm seemed a good choice from its resemblance with DDPG (Deep Deterministic Policy Gradient), acting as an extension for multi-agent environments.

A ‘Tennis.ipynb’ solution notebook was added using the DDPG algorithm implemented in the DRL-Continuous-Control project repository. The reason DDPG works well in this Unity Tennis environment, without many changes, is that the two agents in the environment are trying to learn the **same** exact **policy** and have the **same reward structure**. Thus, the two agents, can **share** the replay buffer and run DDPG locally to learn the best policy possible.

The Cooperation-Competition Challenge

Some of the main challenges surrounding MARL (Multi-Agent Reinforcement Learning) environments are:

- While single RL agents adapt their behavior to their own actions and feedback from the environment, MARL agents need to also adapt to other agents’ learning speed and actions.

- A balance between an agent’s current knowledge exploitation and knowledge base improvement, or exploration, must be reached. Too much exploration can destabilize the agents, making the learning task more difficult.
- Non-stationarity arises because all agents are trying to learn simultaneously, and each agent is faced with a moving target learning problem.
- MARL agents do not always have a complete view of the environment and cannot normally predict the actions of other agents
- Deciding which agent is responsible for the success of failure of a multi agent system is difficult, also known as the ‘credit assignment problem’.
- In most of the cases there is no stable Nash equilibrium, where MARL agents have no incentive to deviate from the initial strategy/policy.
- The existence of this intrinsic pressure causes agents to constantly adapt their policies, get smarter and not necessarily collaborate.
- MARL methods scale poorly with problem size

The challenge with traditional reinforcement learning methods in multi-agent scenarios has to do with the centralized approach to training and policy evaluation.

Experience Replay stores experience (s_t, a_t, r_t, s_{t+1}, a_{t+1}) tuples in memory to remove the correlation between data samples and consequently stabilize training and improve training efficiency. In a MARL environment, each agent is changing with training, causing the environment to become non-stationary, at least, from the perspective of each agent. In a non-stationary environment, past experience might no longer reflect the current environment dynamics, in which case the accumulated experience becomes obsolete.

This threatens the learning stability without a proper incorporation of experience replay in MARL.

Learning Algorithm

MADDPG (as it extends the principles of another reinforcement learning algorithm called DDPG to multi-agent settings), the OpenAI algorithm, allows agents to learn from their own actions as well as the actions of other agents in the environment. It is like a wrapper algorithm adopting a framework of **centralized training** and **decentralized execution**.

The actor evolves a strategy by taking advice from a critic, to decide which actions to reinforce at training time. Without a critic, it tries to increase or decrease the probability of actions that lead to good or bad outcomes respectively. The critic, as the episode unfolds, tries to predict the future rewards. When a critic is added to an actor, the agent is better able to make the distinction between situations and select an action better, improving its performance. This method adds some stability over time compared to conventional RL (Reinforcement Learning) methods, as the actual rewards can exhibit high variance when coordinating multiple agents.

Since, observations are symmetric (at least in this 'Tennis' environment example), experiences can be stored in a replay buffer and reused by all agents collectively, to access the experiences (i.e. observations and actions) of all participant agents. This is a key improvement over the DDPG approach. This is also an immediate, short-lived benefit as this approach can increase short term training stability, decrease training time and increase training performance. But, the down side is that, it suffers from the non-stationarity problem, mentioned above, where the accumulated experience becomes obsolete, and renders unstable training over longer periods of time (see Fig.1 as an example).

Note that the selected action from the critic is used only during the **training** phase. That extra information used during training effectively goes away. During **execution**, the actor's policy network doesn't need the critic's advice, and returns the action for any given state, based on its own observations and predictions of other agents. In 'Tennis-Viewer.ipynb', the agent only loads the 'best_actor_model.pth' **actor model** learned during the training phase. The critic network is ignored.

Core concepts (i.e. experience replay, actor-critic networks):

- The Actor network takes state as input and returns the action to take in the environment
- The Critic network, Deep Q-Network, takes the state and action of all agents as input and returns the action-value (Q-value)
- The Critic and Actor networks are trained by sampling experiences from the replay buffer.
- The action-value from Critic is used to teach the Actor to choose better actions.

Hyper-Parameters

Parameter	**Value**	**Description**
BUFFER_SIZE	1e6	replay buffer size
BATCH_SIZE	256	minibatch size
GAMMA	0.99	discount factor
TAU	1e-3	soft update of target parameters
LR_ACTOR	2e-4	learning rate of the actor
LR_CRITIC	2e-3	learning rate of the critic
ALT_UPDATE_EVERY	10	parameter for switching between 'updating' and 'learning' mode
NUM_UPDATES	4	number of samples to learn in 'learning' mode
ALPHA	0.1	prioritization level (ALPHA=0 is uniform sampling, no prioritization)
EPS_START	1.0	Epsilon start – coefficient for noise at beginning of the training
EPS_END	0.01	Epsilon end - coefficient for noise at end of the training
EPS_DECAY	2e-7	Epsilon decay per learn step

Training and Results

Training the MADDPG did show some unstable training behavior (see Fig 1). At this point the networks for the actor were using 48 nodes for the fully connected layers and 24 nodes and for critic.

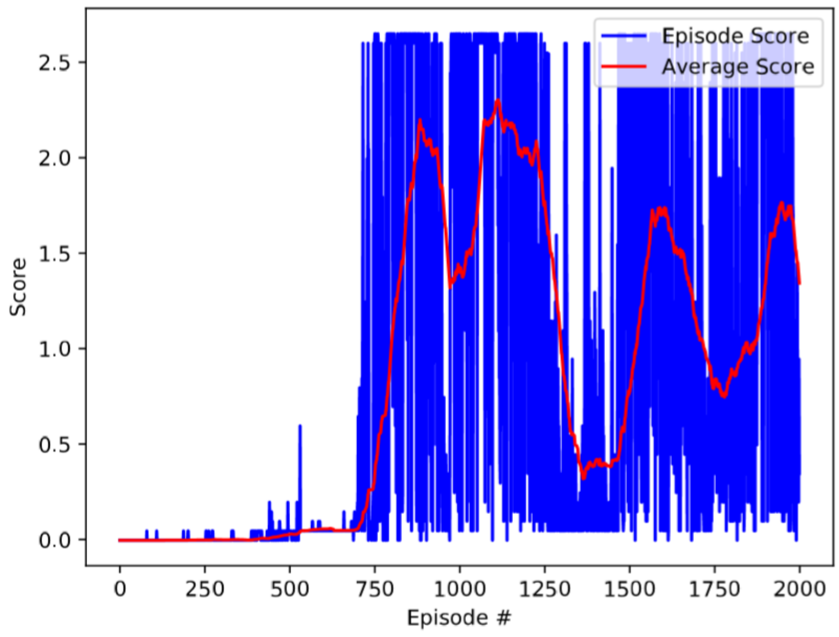

Figure 1 MADDPG unstable training

After increasing these numbers, several times, to 128 and 128 respectively, the training performance continued to be volatile, but overall more stable (see Fig. 2).

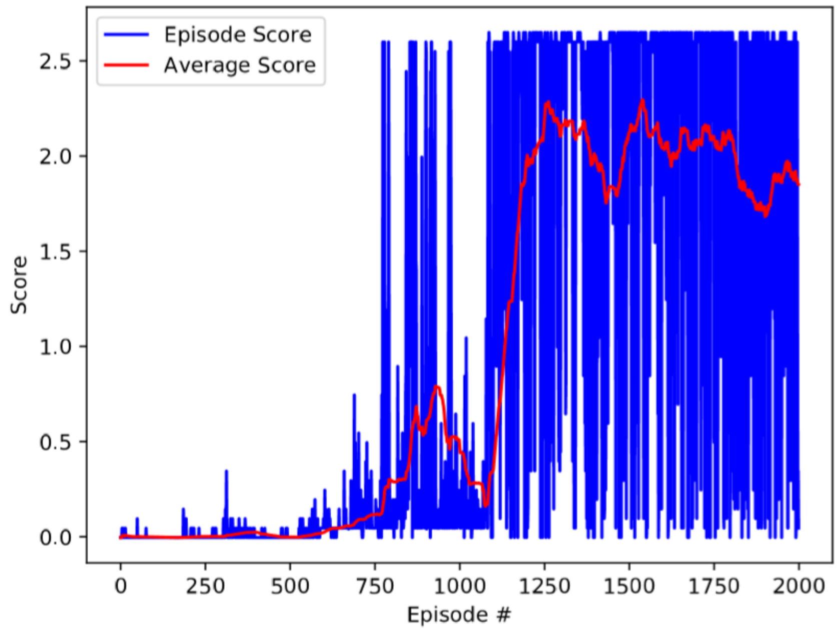

Figure 2 MADDPG Improved training performance and stability

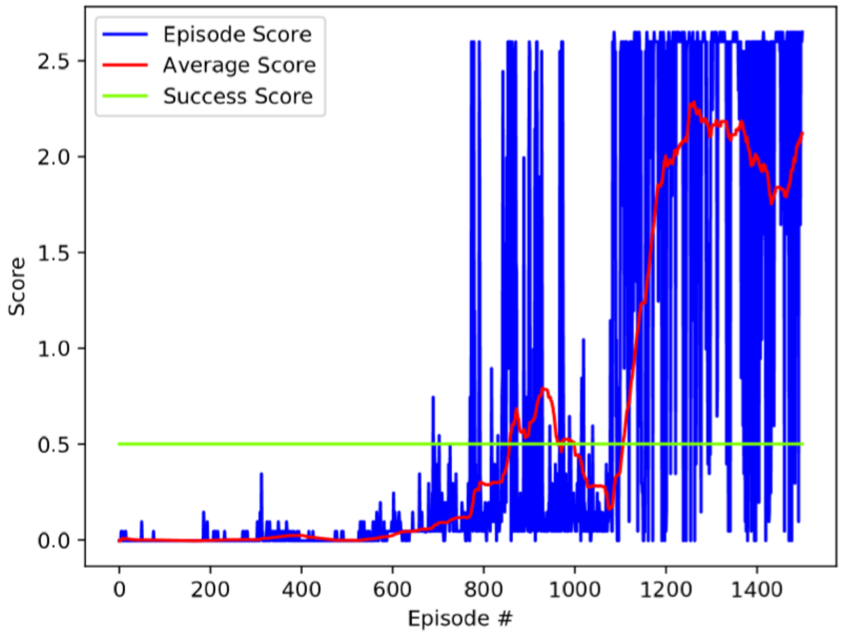

Figure 3 MADDPG training stopped at episode 1500, above 2.0 score

The training process is very sensitive to the hyperparameters and took a good amount of trial and error to finally get to some meaningful results.

Even though the training was a little unstable, the agent achieved a rolling score of 2.12, well above the 0.5 required to solve this environment.

One interesting fact to note is that, the optimal solution was found without exploratory noise added to the agent's chosen action. Adding the noise or its magnitude was interfering with the learning, therefore no noise was eventually added.

Future Work Ideas

Searching for a better hyper-parameters configurations could help, but adding back the exploratory noise and dropout layers for a more stable and generic model would be the first step. The exploratory noise could follow a very low magnitude sinusoidal signal (e.g. see Fig. 5 and Fig. 7), that fluctuates between some very small positive number (e.g. 0.083 or 0.05, see Fig. 4 and Fig. 6) and 0. These are examples, but tweaking the amplitude and/or wavelength of the sinusoidal wave would be necessary. This would basically allow for some timid exploration or exploitation (as seen so far in this report).

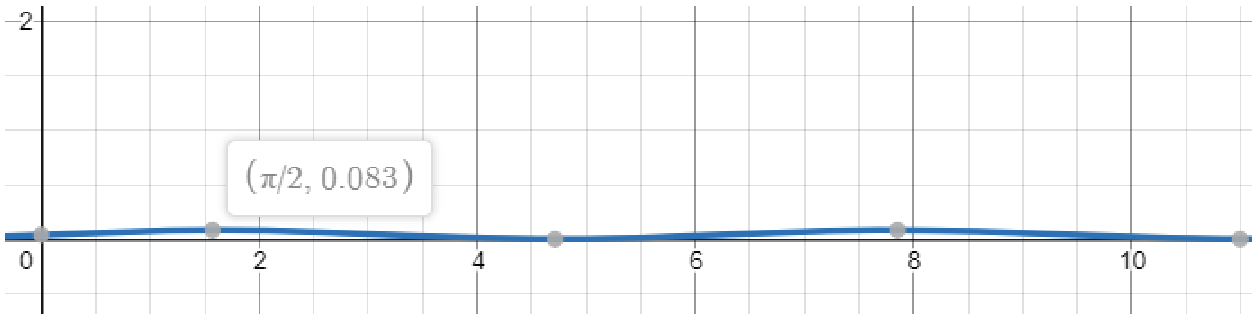

Figure 4 Noise signal cap

$$y = \left(\frac{\sin(x)}{6} + \frac{1}{6} \right) \frac{1}{4}$$

Figure 5 Noise signal cap formula

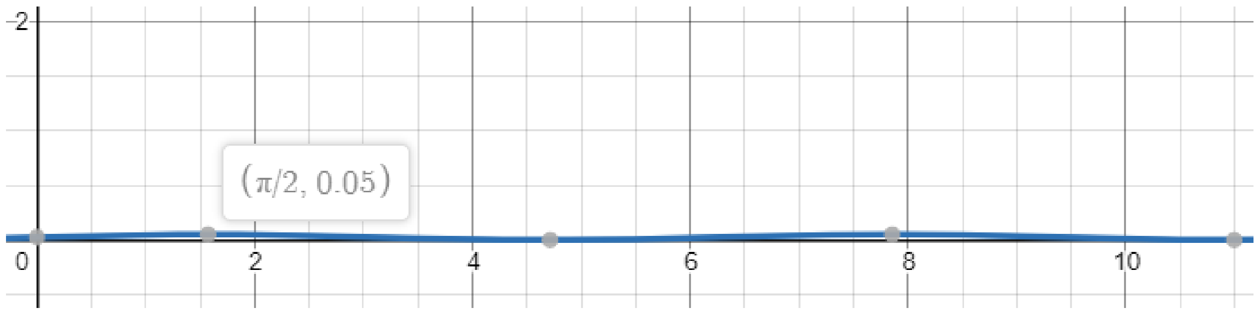

Figure 6 Noise signal cap

$$y = \left(\frac{\sin(x)}{10} + \frac{1}{10} \right) \frac{1}{4}$$

Figure 7 Noise signal cap formula

Additionally, since an Experience Replay buffer is used, replacing it with a Prioritized Experience Replay buffer would help learn from less frequently seen experiences, and/or with high error, causing the training to be unstable, as seen in Fig. 1.

Next, adapting the MADDPG to a more challenging environment, like the Unity Soccer environment, with different reward structures could also be a good idea.