Report for Project 3: Collaboration and Competition

Udacity Deep Reinforcement Learning Nanodegree

Bernhard Häfner

April 27, 2020

1 The Implementation

The main part of the code is written in the Jupyter Notebook `Tennis.ipynb`, which guides the user when executing the code. The intelligent agent has been implemented in the `agent.py` file. This Multi-Agent DDPG (MADDPG) agent consists of two DDPG agents defined in `ddpg.py`, each of which uses two neural networks, for its actor and its critic, as defined in `model.py`. The prioritized experience replay buffer is defined in `buffer.py`, while the Ornstein-Uhlenbeck noise process is defined in `OUnoise.py`.

1.1 The Neural Networks

The networks as defined in `model.py` by default have two hidden fully-connected layers with 256 and 128 neurons, respectively. The goal of the actor is to provide the best action given a state s: $a = \mu(s; \theta_\mu)$. The critic's goal is to estimate Q for the best-believed action, i.e. $Q(s, \mu(s; \theta_\mu); \theta_Q)$. For the critic, the actions are fed into the network as inputs to the second hidden layer, while states are directly input to the first layer. Here, always the whole state space for both agents is used. For the critic, also two action sets are used, namely

- own actions and own next actions for the critic update and experience weights for the Replay Buffer
- own actions and expected actions for the actor update.

Important to notice here is the fact that the order of these to sets had to be swapped for the two agents, since they are playing against each other (see the `if self.index:` decision in `ddpg.learn`.

Replay Buffer	Value	Agent Hypers	Value
Buffer Size	100k	γ	0.99
Batch Size	128	τ	0.06
α	0	$\text{LRate}_{\text{actor}}$	0.001
β	$0 \rightarrow 1$	$\text{LRate}_{\text{critic}}$	0.001

Table 1: The hyperparameters used for the learning algorithm.

1.2 The DDPG Class

The agent class is capable of learning according to the Deep Deterministic Policy Gradient (DDPG) algorithm as discussed in class and in this paper.

Within its `__init__` method, several internal variables are defined: state size, action size, and the `PrioritizedReplayBuffer` for experience replay. Besides, the network weights (local w and target w^{-}) and optimizer to be used (Adam) are defined for the agent's actor and critic network. As action noise, an Ornstein-Uhlenbeck process is defined.

The `step` method adds the current experience to the replay buffer and triggers a learning step. To do this, it triggers the `learn` method, which backpropagates the errors made from sampled experiences through the network to update weights, once for the critic, and once for the actor. The actual (soft) update is done in `soft_update`. Since we are using experience replay, when adding samples in the `step` function, we have to calculate the importance weights (priorities) for the samples. For us, this is the quadratic error between expected Q (based on local critic) and target Q (based on target critic). Similarly, when learning from samples, we need to update the weights (priorities) for the sampled tuples, by checking their current expected Q value against the current target Q value (again, quadratic error).

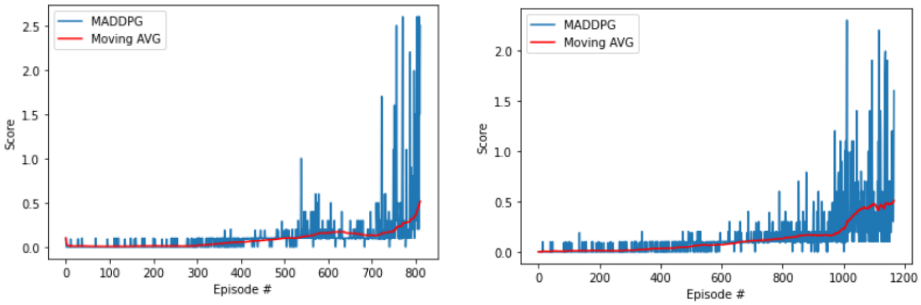
Hyperparameters used are mentioned in Table 1.

The `act` method is used to return the action suggested by the actor for the current state.

The `save` and `load_weigths` functions are used to save trained agents for further use and to load previously-trained agents, respectively. They are quite straightforward, only some checks for stringent dimensions of the networks are done in the `load` method.

1.3 The Prioritized Experience Replay Buffer

In order to use samples more often that we currently *deem* to be more important, we included a Prioritized Experience Replay Buffer. In contrast to the normal replay buffer, we additionally store priorities for each sample, reflecting how high we value each sample. These priorities determine how likely an experience is sampled. After each learning step, we have to update the priorities in order to be consistent with an *updated* view on the value of the experience. This assessment of the experience value is based on the critic networks' Q values for the experience samples.



(a) Original without normalization. (b) After adding normalization layers.

Figure 1: The scores reached by the intelligent agent.

The implementation is based on this paper and has been inspired by PHRABAL's implementation. It comprises an initialization method as well as methods for adding experiences and corresponding weights, sampling from the stored experiences, updating priorities, and a function for returning the current length of the buffer. The value β, which controls the importance of sampling weights, is increased linearly in the training loop from 0 to 1.

1.4 The MADDPG class

For interacting with the environment, the MADDPG agent is a convenient wrapper class. It comprises the methods `act`, `step`, `reset`, `save`, and `load_weights` and is used to spawn the respective methods of the comprising DDPG agents in a standardized way.

2 Results

As Figure 1a shows, the agent is capable of learning, the environment has been solved within **711 episodes**. During a search for good combinations of hyperparameters, several options were tried. This included, among others, adding additional batch normalization layers for actor and critic, the outcome of which is displayed in Figure 1b. However, the model performed worse for adding normalization (slower training and longer execution times).

3 Experiences and Insights

This project was quite challenging. Most notably, it the learning is facing quite big instabilities in this environment. Due to the very time-consuming runs (even in the GPU-enabled Udacity work space), it is hard to find suitable hyperparameters. After trying different model sizes, buffer sizes, with and without experience replay, and with several configurations of the models (completely independent regarding states or not, with or without normalization, with our without dropouts, etc.), I finally got a hold on

the environment. Both DDPG agents share now state spaces for the actor (input size = 2×state size), while they use own and next actions for the critic.

Technically, I experienced that while for organization and clarity, it is good to have separate files for the different involved classes, it is much more efficient to include everything into the Jupyter Notebook during programming and debugging. The reason being that it is necessary to restart the kernel every time something in imported modules is changed. For the Udacity workspace, this means that also the installation command taking around a minute needs to be executed, again. So having half a minute delay only for small bugfixes sums up quite a lot, especially during early stages of coding.

4 Ideas for Future Improvements

While the setup currently works sufficiently well, it may be interesting to re-implement some of the features tried on the way which have not found their way into the final solving version. This could be

- Adding dropouts to the critic
- Adjust hyperparameters, especially the batch size and prioritization parameters.

I already started doing this, but only without success. For example, adding batch normalization layers

Additionally, a mentor suggested to use parameter noise instead of action space noise as described here (video here).

It would be interesting to find a better configuration which potentially could solve the environment even faster.