Project Report: Project 3 - Collaboration and Competition

Christian Michler

April 26, 2019

1 Introduction

In this project, I am studying multi-agent actor-critic methods. To put this into context, let me recall that in the first assignment of Udacity's Deep Reinforcement Learning Nanodegree [1], I studied *value-based methods* such as Deep Q Networks (DQN) [2] for training an agent to learn the value function that maps each state-action pair to a value. The optimal policy is then obtained from this action-value function estimate by taking the action for each state that has the biggest value.

Policy-based methods on the other hand directly learn the optimal policy and therefore do not have to maintain a separate value function estimate. Policy-based methods are attractive for continuous action spaces and, moreover, they can learn stochastic policies. A subclass of such policy-based methods are *policy gradient methods* that estimate the weights of an optimal policy by using gradient ascent.

Actor-Critic methods combine the respective strengths of policy-based methods and value-based methods. The actor uses a policy-based approach in which it learns to act by directly estimating the optimal policy and maximizing reward by gradient ascent. In contrast, the critic uses a value-based approach in which it learns to estimate the value of different state-action pairs. Actor-critic methods combine these respective approaches which makes them more stable than value-based methods and they require fewer training samples than policy-based methods. Deep Deterministic Policy Gradients (DDPG) [3] can be considered an actor-critic method and an extension of the DQN Method [2] to continuous action spaces.

When it comes to *multi-agent environments*, conventional methods such as Q-Learning or policy gradient methods face difficulties, as discussed in [4]: As each agent's policy is evolving as training progresses, this renders the environment non-stationary as perceived by any individual agent and precludes the immediate use of experience replay which is essential for making Q-learning stable. Policy gradient methods, on the other hand, face difficulties from a variance in gradient estimates that increases with the number of agents. This has motivated the extension to Multi-Agent Actor-Critic Methods [4]. The nature of interaction between agents can either be cooperative (all agents must maximize a shared return), competitive (agents have conflicting goals), or both.

In the first assignment [1], the setting was a *discrete* action space, i.e. an action space with a finite number of discrete action values, and I solved it using Deep Q Networks [2]. In the second assignment, the setting was a *continuous* action space, i.e. an action space that has an infinite number of possible action values, and I used DDPG [3]. In this third assignment, the setting involves multiple agents. In particular, using the *Tennis* Environment [1] involving two players, the goal of this project is to train a multi-agent Reinforcement Learning method to play tennis for as many time steps as possible. The exercise uses the Unity ML Agents Toolkit [5] to train an agent. I will use and study an extension of DDPG to multi-agent environments henceforth referred to as Multi-Agent DDPG (MADDPG). Similar to [4], I implement a centralized training / critic with decentralized execution / actor, where the critic is learning from the combined information of both agents, while the actor has access only to local information. As pointed out in [4], the main motivation behind this approach is that, if one knows the actions taken by all agents, then the environment is stationary even if the policies change.

The outline of this report is as follows: Section 2 describes the Multi-Agent DDPG method. Section 3 first describes the problem setting of the environment in more detail. This section also

contains the numerical results obtained using this environment and discusses the findings. Section 4 discusses some improvements and future work.

2 Multi-Agent DDPG

To extend DDPG to multiple agents, I followed the outline given in Udacity's Benchmark discussion, i.e. each agent receives its own, local observation which facilitates adapting the code to simultaneously train both agents through self-play. Each agent uses the same actor network to select actions, and the experience was added to a shared replay buffer. As was also pointed out in Udacity's Knowledge Hub, this allows to treat this problem as a simple continuous-control problem with two parallel agents, similar to the multi-agent reacher problem (assignment 2). Similar to [4], I then extended my implementation to a centralized training / critic with decentralized execution / actor, where the critic is learning from the combined information of both agents, while the actor has access only to local information. This requires a shared replay buffer collecting the experiences of all agents, to which all critics have access; this has implications for the critic network architecture as detailed below.

As a starting point, I therefore take my DDPG implementation of assignment 2 (originally based on the pendulum example) as a starting point, and use two fully-fledged DDPG actor-critic models, each with their own actor and critic. Each agent acts on its own behalf, but the critic can also leverage the experience gained by the other agent. I will refer to this extension of DDPG to multiple agents with shared replay buffer as Multi-Agent DDPG (MADDPG) which is similar to [4] (see also the algorithm included in the appendix of this reference).

Extending the DDPG framework to two agents requires that functions such as `act()` and `learn()` are implemented on the top / MADDPG level, which then call `act()` and `learn()`, respectively, of the individual DDPG agents. It is the MADDPG level that is now `step()`'ed forward, coordinates the overall process and the individual DDPG agents and manages the replay buffer. It also involves accommodating more than a single agent's information in the replay buffer. Hence, on the MADDPG level, information resulting from actions and states need to be concatenated / flattened when it is passed into the replay buffer and extracted when it is retrieved; cf. also the discussions on Udacity's Knowledge Hub. This is done so that the replay buffer can be re-used in its current, single-agent form; see also [6].

Note that the critic uses `next_action` information from all agents which requires access to `actor_target` of the individual agents (cf. the Bellman Equation). Hence this needs to be done on the MADDPG level, see also the implementation in `maddpg_agent.py`. As pointed out in [4], the main motivation behind this approach is that, if one knows the actions taken by all agents, then the environment is stationary even if the policies change.

Both actor and critic use a deep neural network as nonlinear function approximator where the network weights are determined such that the neural network best approximates the respective functions. In this project, the neural network is implemented in `model.py`. Both actor and critic have two hidden layers of 256 nodes.

Since the critic is learning from combined action and states of both agents, the size of the critic's and actor's input layers are, respectively,

1. Size of actor input layer = state size
2. Size of critic input layer = number of agents * (state size + action size)

as discussed in the Knowledge Hub. Consequently, for the critic input layer the states and actions are concatenated right at the input layer rather than adding the actions in the first hidden layer as was done in Assignment 2 [6].

The actor (policy) network maps states to actions and has the following structure:

1. Input nodes (24 states)
2. First hidden layer (256 nodes) with ReLU activation
3. Batch normalization (activated by default)

4. Second hidden layer (256 nodes) with ReLU activation
5. Output nodes (2 actions) with tanh activation[1]

The critic (value) network maps (state, action) pairs to Q-values and has the following structure:

1. Input nodes (2 agents * (24 states + 2 actions) = 52 nodes)
2. First hidden layer (256 nodes) with ReLU activation
3. Batch normalization (activated by default)
4. Second hidden layer (256 nodes) with ReLU activation
5. Output node (1 Q-value), no activation

As a side note, MADDPG has two DDPG agents corresponding to the two players. Each DDPG agent has an actor and a critic, and each actor and critic has a local and a target network. So in total I have here eight neural networks.

Multi-agent DDPG is an off-policy method and uses Experience Replay, i.e. rather than learning from sequential batches of experience tuples directly, these batches are stored in a replay buffer from which the agent then randomly samples. This mitigates the risk of potential correlations in a sequence of experience tuples and, moreover, enables the agent to learn from individual tuples multiple times, recall rare occurrences, and in general make better use of the generated samples. Experience Replay is implemented in `maddpg_agent.py`.

As mentioned above, the MADDPG implementation also invokes a target network besides the regular network, i.e. weights from a separate network are used in approximating the function value. While the local / regular network is updated after each time step, the target network is '*soft updated*'; see `maddpg_agent.py`.

Adding noise usually helps exploration. Over time the noise is then reduced as the agent gradually shifts focus from "exploration" to "exploitation". Commonly a decay rate < 1 is used such that over time the noise added decreases to zero. Here, I have found that even without adding noise to the action space the agent's exploration space is sufficiently rich for the agent to achieve its target score of +0.5, as shown in the next Section. I have, however, done numerous runs with adding various ways of time-decaying noise as well, and the agent likewise achieved the target score.

3 Problem Setting, Results and Discussion

In the Tennis environment [1], 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 in order to solve the environment, the agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically, after each episode, the rewards that each agent received are added up (without discounting), to get a score for each agent. This yields two (potentially different) scores. The maximum of these two scores is then taken. 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. The above description is also given in the README file of the assignment [1].

The focus of this assignment is on extending a Reinforcement Learning algorithm to multiple agents. While this also involved some parameter tuning by trial and error, fortunately, the settings

[1] The tanh activation serves to scale the output action values to be a number between -1 and 1.

from the continuous-control problem (assignment 2) appeared to be good choices also for the current assignment. The chosen parameter settings are given in Table 1. This list of hyperparameters and additions (such as batch normalization, gradient clipping, etc.) is based on the previous assignment 2 as well as on discussions / suggestions in Udacity's Student Hub and Knowledge area and on the papers [3, 4]. For a more detailed discussion of these hyperparameters see my report of Assignment 2 [6]. The corresponding numerical result is given in the jupyter notebook `Tennis.ipynb` for an agent that succeeds in performing the given task and attain the target score of +0.5.

From Fig. 1 it is apparent that quite a long training time is required - In the first 1100 episodes the average score is not increasing much, and only around episode 1100 it starts to increase, then rapidly goes up and reaches 0.514 after 1226 episodes.

Parameter	**Value**
`fc_unit` #nodes in hidden layers of Actor/Critic network	256
`LR_ACTOR, LR_CRITIC` learning rate actor, critic	5e-4, 5e-4
`BATCH_SIZE` size of mini-batches	256
`BUFFER_SIZE` Size of Replay buffer	1e5
`UPDATE_EVERY` how often update network	1
`UPDATE_M_TIMES` how many times update network	1
`sigma` OU noise Std. Deviation	0.20
`add_noise` if noise to be added	no
`use_BatchNorm` Batch normalization	yes
`use_clipping` gradient clipping for critic	no
`WEIGHT_DECAY` L2 weight decay	0
`TAU` for soft update of target parameters	1e-3
`GAMMA` discount factor	0.99

Table 1: Hyperparameters settings for the Multi-Agent DDPG (MADDPG) Agent that successfully attains the target average score of +0.5 after 1226 episodes; see also `Tennis.ipynb` and Fig. 1.

4 Ideas for Future Work

For future work, the hyperparameters of the Multi-Agent DDPG agent could be better tuned to possibly achieve better performance. It also seems worthwhile to implement *Prioritized* Experience Replay [7] rather than the plain vanilla Replay Buffer only. Moreover, I would consider the extension of Proximal Policy Optimization (PPO) [8, 9] to multi-agent settings, assess its performance on the environment considered, and see how it compares to the approach taken here.

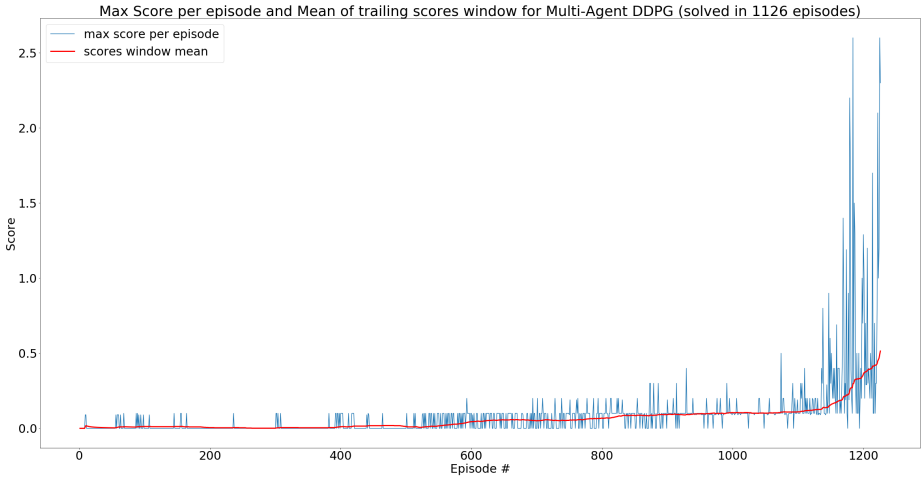

Figure 1: Max score per episode and mean score over a trailing-score-window of the last 100 episodes for the MADDPG Agent with parameters set according to Table 1. Achieved a trailing-score-window mean of +0.5 after 1226 episodes. If solved is counted as number of episodes - 100, then this corresponds to 1126 episodes as indicated in the figure.

References

[1] https://eu.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg and Demis Hassabis, Human-level control through deep reinforcement learning, Nature (518), 529–533, 2015. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[3] 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, ICLR Conference paper (2016). https://arxiv.org/pdf/1509.02971.pdf

[4] Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments, Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel and Igor Mordatch, Neural Information Processing Systems (NIPS) (2017). https://arxiv.org/pdf/1706.02275.pdf

[5] https://github.com/Unity-Technologies/ml-agents

[6] https://github.com/christian-m02/Udacity-Deep-RL-Project-2-Continuous-Control

[7] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, Prioritized Experience Replay, ICLR Conference paper (2016). https://arxiv.org/pdf/1511.05952.pdf

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov, Proximal Policy Optimization Algorithms. https://arxiv.org/pdf/1707.06347.pdf

[9] https://openai.com/blog/openai-baselines-ppo/