Udacity Deep Reinforcement Learning

Multi-Agent Reinforcement Learning

Max Baugh

February 28, 2022

Abstract

Several experiments were performed with the Unity “Tennis" environment, attempting to use a multi-agent variant of DDPG to solve the environment. Two of the eight experiments demonstrated success, but the wildly varying performance with minor changes in the hyperparameters implies the strategy taken here is sub-optimal.

1 Introduction

The environment for this project is the Unity “Tennis" environment, in which two agents play a game of tennis. The reward structure has two components, a $+0.1$ for hitting it over the net and -0.01 for allowing the ball to hit the court on your side or to go out of bounds. The environment is considered “solved" when the 100-episode moving average of the maximum of the two agent’s scores crosses 0.5. This is both cooperative and competitive because each agent is attempting to score on the other but the ball going out of bounds ends the episode & truncates the score.
Each agent receives as an observation a 24 element vector corresponding to the trajectory of the ball & its own location on the field, as such these are *partial* observations of the environment, as opposed to complete environment observations as in previous projects.

2 Actor-Critic Deep Deterministic Policy Gradient

Actor-Critic methods are a class of Reinforcement Learning algorithms in which two separate function approximators are used to learn how to perform a particular task. The first approximator is the Actor, which is the thing that actually *takes actions* in the environment, while the second is the Critic, which evaluates the effectiveness of the Actor’s actions.

One of the earliest appearances of Deep Reinforcement Learning was the DQN model, which uses a neural network function approximator to estimate the action-value function for an environment with a discrete number of possible actions (see: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf). This worked extremely well, but would fail in the case of a continuous action space with a non-denumerable set of possible actions, as in the case of a robot determining how much torque to apply to keep its arm stable. The solution proposed by a team at Google Deepmind (see: https://arxiv.org/pdf/1509.02971.pdf) is to use a second neural network which takes the state as the input and directly outputs the optimal action, which then gets fed into the larger Q network along with the state.

In this environment, a single Agent contains two Actors and one Critic network, as well as a single replay buffer. At each time step, the two Actor networks see a partial environment & choose their actions, each generating an experience tuple (state, action, reward, next_state, done) which gets pushed to the replay buffer. During the learn step, a set of experiences is drawn from the replay buffer for each Actor. These are then combined in order for the Critic target network to calculate the target Q values, the Critic primary network then calculates its estimates, and the Critic is updated based on the loss. Each Actor then gets updated using the loss from the Critic primary network using both sets of experiences, so each Actor is able to learn from the experiences of the other.

Hyperparameter	Value
Buffer Size	10^5
γ	0.99
LR Actor	0.0001
μ	0
N_Updates	1
σ	0.2
θ	0.15
Update Every	1

Table 1: Core Hyperparameters

Recall the DQN agent from projects with discrete action spaces, in which a neural network estimated the Q-function and the policy was simply “take the arg-max action": here the Actor is the continuous-space version of that arg-max function, and the Critic is the Q-function. Because the policy is deterministic, it will not explore, in a given state it will always produce exactly the same output. In order to enable exploration, Ornstein-Uhlenbeck Noise is added; this noise turns out to have a *massive* impact on performance.

3 Experience Replay

Experience Replay is a method to significantly increase the sample efficiency of an agent by revisiting previous states & actions. This is done by storing experience tuples as (state, action, reward, next_state, done) in a buffer and randomly drawing from that buffer at regular intervals. Due to memory limitations, a deque was used with a size of 1,000,000 tuples. Though it is possible to use a more targeted, prioritized experience replay, constructing one that is *fast* is a more involved task & mine is not yet operational.

4 Ornstein-Uhlenbeck Noise

The Ornstein-Uhlenbeck process is an example of a stationary Gauss-Markov process, meaning that it is both a Gaussian & a Markov process and that it is homogeneous in time. It is similar to the random walk but with a tendency to revert to the mean value over time. In this project, OU noise with a mean value of zero is added to the action values as a form of regularization; physically, one can imagine mild friction in the joints of the robot arm or slight differences in the air currents surrounding it affecting the final orientation of the arm.

There are three hyperparameters controlling the amount of OU noise applied in our system: μ, σ, and θ. The first term, μ, is the mean value the noise reverts to over time, it is always set to zero because any non-zero value would simply constitute a bias that would be quickly compensated for (in the case of the robot arm, this would be something like wind blowing from a consistent direction). The second parameter, σ, determines the “strength" of the noise, while the third term, θ, is the strength of the reversion to the mean.

In this implementation, the amount of noise added to each action is:

$$\delta_i = \theta * (\mu - x_i) + \sigma * U(0, 1) \tag{1}$$

where x_i is the i^{th} element of the action and $U(0, 1)$ is the uniform random distribution from 0 to 1 inclusive & exclusive, respectively.

5 Experiments

Many experiments were done in this project, with the hyperparameters recorded in a JSON file. This environment, like the “Reacher" environment of the previous project, is rather sensitive to hyperparameter settings, especially the learning rate of the Critic and the soft update parameter.

The hyperparameters μ, σ, and θ were already explained in the section on Ornstein-Uhlenbeck Noise, the others are mostly self-explanatory. The term γ is the discount factor, which was not

Hyperparameter	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 6	Exp. 7	Exp. 8
Batch Size	128	128	128	128	128	256	128	128
LR Critic	0.0001	0.0005	0.0002	0.0002	0.0002	0.0001	0.0003	0.0002
Episodes	2500	2500	2500	2500	2500	4000	4000	4000
τ	0.06	0.06	0.01	0.01	0.03	0.05	0.03	0.01
Pass?	No	No	No	No	No	**Yes**	No	**Yes**

Table 2: Experimental Results

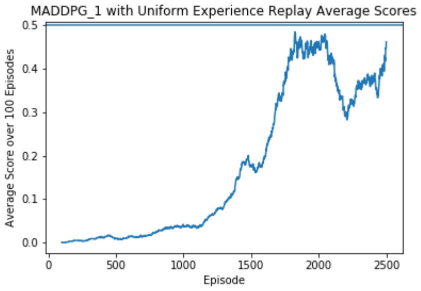

Figure 1: First Experiment

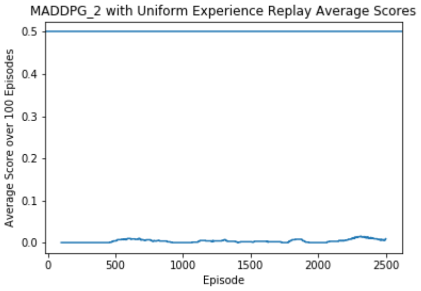

Figure 2: Second Experiment

experimented with at all in this project.
$N\_UPDATES$ is the number of learning steps to do every time it is time to learn; consulting the Udacity Knowledge forums found that this parameter can have a significant impact, but I did not experiment much with it in this project.
Update Every is how often to perform the soft update of the target network in the DDQN architecture.
Weight decay is the decay rate applied to the weights in the neural network, which could have been applied to the Critic although in this experiment was not actually implemented (a weight decay of zero is the same as no weight decay).
Window Size is the number of episodes to average over when doing the final performance evaluation.

All of the experiments were done using the same Actor & Critic network architectures. The Actor networks had two hidden layers, with 400 and 300 nodes, respectively. The Critic networks were somewhat larger, with three hidden layers of 256, 256, and 128 nodes each.
The Agents themselves contain two Actor networks, each with their own target network & optimizer. These Actors are initialized with different random seeds to further enhance exploration, as each one will take a slightly different action for the same input. Following this same logic, there is a single experience replay buffer that both Actors feed into and draw from during the learning step, so either one may learn directly from the experiences of the other. Similarly, there is only a single Critic network that takes in states & actions from either Actor network to learn and then dispenses its wisdom to both Actors.

6 Results

The success of the agent on this environment seems unusually dependent on the hyperparameter settings.

One additional experiment was done, attempting to use the “Twin-Delayed Deep Deterministic Policy Gradient" (TD3) algorithm, which experimentally has been found to be superior to DDPG in the single-agent setting. Unfortunately this experiment went rather poorly, and there was not enough time to correct it. The TD3 experiment was based on the code here https://github.com/sfujim/TD3/blob/master/TD3.py

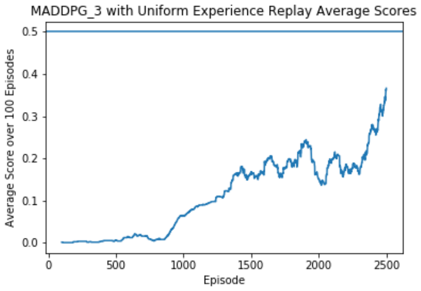

Figure 3: Third Experiment

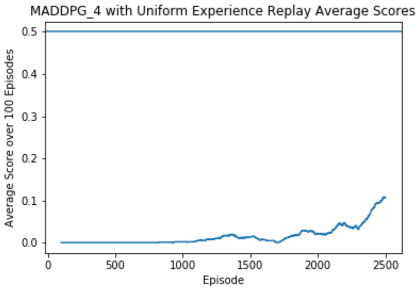

Figure 4: Fourth Experiment

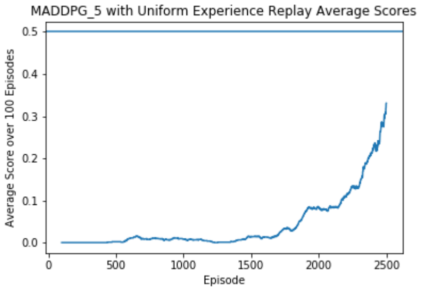

Figure 5: Fifth Experiment

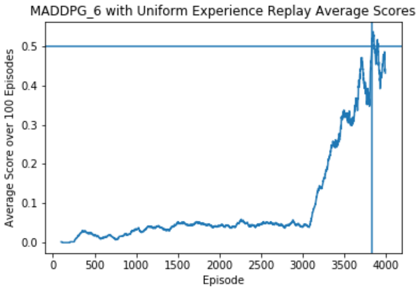

Figure 6: Sixth Experiment

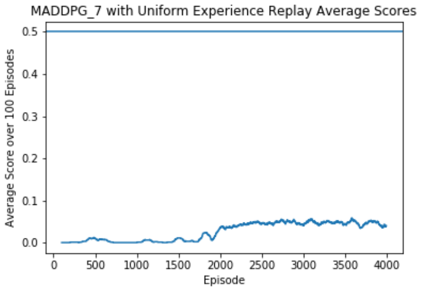

Figure 7: Seventh Experiment

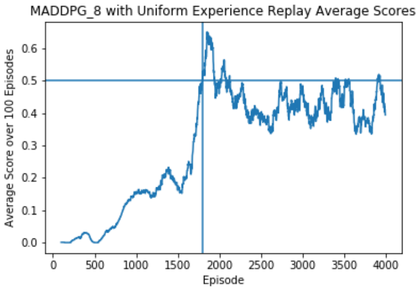

Figure 8: Eighth Experiment

7 Conclusions

Environments with continuous action spaces are significantly more complex than those with discrete action spaces. Nevertheless, proper utilization of a pair of relatively simple neural networks in an Actor-Critic architecture, where the Actor determines the exact actions to take and the Critic evaluates the performance, can solve even these environments in relatively short order.

8 Future Opportunities

There is ample opportunity to continue expanding this project. Hyperparameter Optimization was found to be especially important in this project, and many hours of further work could be spent optimizing the other hyperparameters like the soft-update factor τ, the frequency of updates, or the number of learn steps per update.

Implementing prioritized, as opposed to uniform, experience replay ought to enable the agent to learn much more quickly by preferentially using more informative experiences when learning. Unfortunately, my implementation of it is too slow to really use, upgrading it will be a priority for the future.

More interesting work would be to utilize a different style of RL architecture, such as Proximal Policy Optimization, Trust Region Policy Optimization, or Q-Prop, and then compare the results with those of this Deep Deterministic Policy Gradient agent.

Improvements that can be made using the DDPG architecture include the use of N-step TD updates or Generalized Advantage Estimation, although how exactly to *implement* either of those is something I still do not understand, particularly how to combine them with Prioritized Experience Replay.

Much to learn there still is!