Udacity Deep Reinforcement Learning

Continuous Control

Max Baugh

January 24, 2022

Abstract

A Deep Deterministic Policy Gradient Agent with a uniform experience replay was constructed to solve the 20 arm "Reacher" environment provided by Unity. Several experiments were done with different hyperparameter settings before one was found which could solve the environment. The apparent sensitivity to hyperparameters has inspired a stronger desire to learn how reinforcement learning works at a deeper level.

1 Introduction

The environment for this project is the Unity "Reacher", which simulates a set of 20 robot arms anchored at the base. The reward function is $+0.1$ per time step in which the arm is in the goal location, and zero at other times. The observation space per robot arm consists of 33 variables corresponding to position, rotation, velocity, & angular velocity measurements. The action space consists of four real numbers between -1 and $+1$ corresponding to torques applied on each of two joints on the robot arm.

The environment is considered "solved" if the average score across all 20 and over 100 episodes exceeds 30. Within a training episode, there is a limit of 1000 allowed time steps, which is necessary because otherwise one can achieve a higher score by merely letting it run longer.

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, at each time step the agent gains an experience tuple (state, action, reward, next_state, done) and the action is stored in the replay buffer. Periodically, the agent conducts a set number of learn steps, in which it draws a batch of experiences from the buffer, calculates the TD target & loss values, and updates the Actor & Critic model weights accordingly. 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 (ex: the torque on the i^{th} joint) 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. The Actor & Critic models were both fixed from the beginning based on the Udacity solution to the OpenAI Gym "Bipedal Walker" environment, which is similar enough to the Reacher to be have comparable requirements, with refinement inspired by fellow Udacity student Daniel Diegel (`https://github.com/DiegelD/Deep-Reinforcement-Learning-ND/blob/main/p2_continuous-control/Continuous_Control.ipynb`). It turns out this environment is rather sensitive to hyperparameter settings, especially batch size, learning rates, and update frequencies.

The settings that passed the environment for the first time are presented in 1.

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 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.

Hyperparameter	Value
Batch Size	256
Buffer Size	1,000,000
γ	0.99
Actor Learning Rate	0.0002
Critic Learning Rate	0.001
μ	0.0
$N\_UPDATES$	10
seed	37
σ	0.10
τ	0.001
θ	0.15
Update Every	20
WEIGHT DECAY	0
Window Size	100

Table 1: Hyperparameter settings

6 Results

The success of the agent on this environment seems unusually dependent on the hyperparameter settings. The single greatest example of this was the transition from version 7 **??** to version 8 2: the *only* hyperparameter that was changed was the σ value of the OU noise, which was reduced from 0.20 to 0.10, and this tiny change caused the agent's 100-episode-average at level-off to jump from 27.5 to 37.5, and to reach the level-off point in many fewer episodes. Indeed, version 8 first crossed the 30 point threshold after only 80 episodes, and had a 100-episode-average above 30 at only 143 total episodes.

Before reaching version 8, the hyperparameters that were experimented with were the batch size and the learning rates. Increasing the batch size did seem to improve the performance per episode, at the cost of increasing the time per episode. Specifically, it appears that doubling the batch size increases the average episode time by about 25% and the performance per episode by roughly 45%, so larger is generally more efficient (hence why the later versions were all using a batch size of 256). Reducing the learning rate of the Critic from 0.001 to 0.0005 hampered performance significantly, as did doubling it. Doubling the learning rate of the *Actor* on the other hand, from 0.0001 to 0.0002, improved performance rather significantly, producing versions 7 & 8, which were the most performant of all those tested in this project.
After version 7, it was clear that the models were appropriately architectured and that it was just a matter of adjusting the hyperparameters a little bit. Given the observed sensitivity, and the known fact that learning rate decay often improves performance, three possibilities struck me as obvious: 1) decay the learning rate of the actor, 2) decay the strength of the noise (which was done by fellow Udacity student Daniel Diegel), or 3) reduce the strength of the noise for the whole run. I chose option 3 because it was by-far the simplest to implement, requiring only one character change in a single cell in the notebook; it worked, so the other possibilities were not tested. From a theoretical perspective, learning rate decay would probably improve things a bit more, but it would require additional complexity.
The GPU was found to be almost twice as fast as the CPU, a significant improvement to be sure but not quite as exciting as had been hoped. My recommendation to future students using the Udacity workspace for this project would be to use the active_session module from workspace_utils to run long CPU jobs until an adequate set of hyperparameters is found.

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.

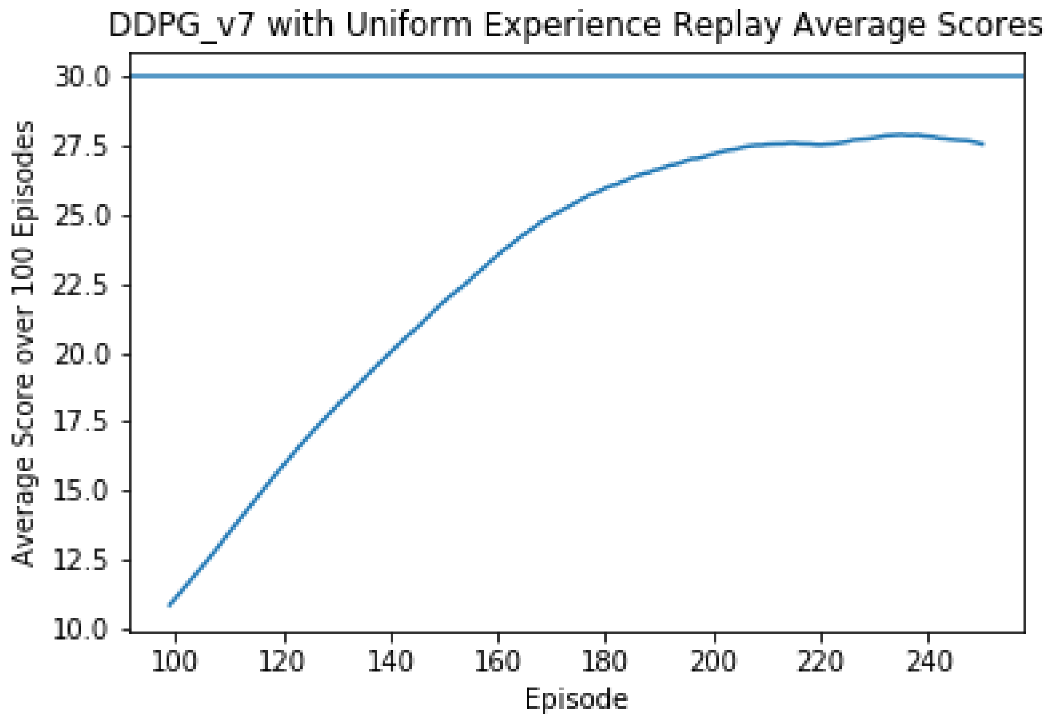

Figure 1: 100-episode-average scores for version 7

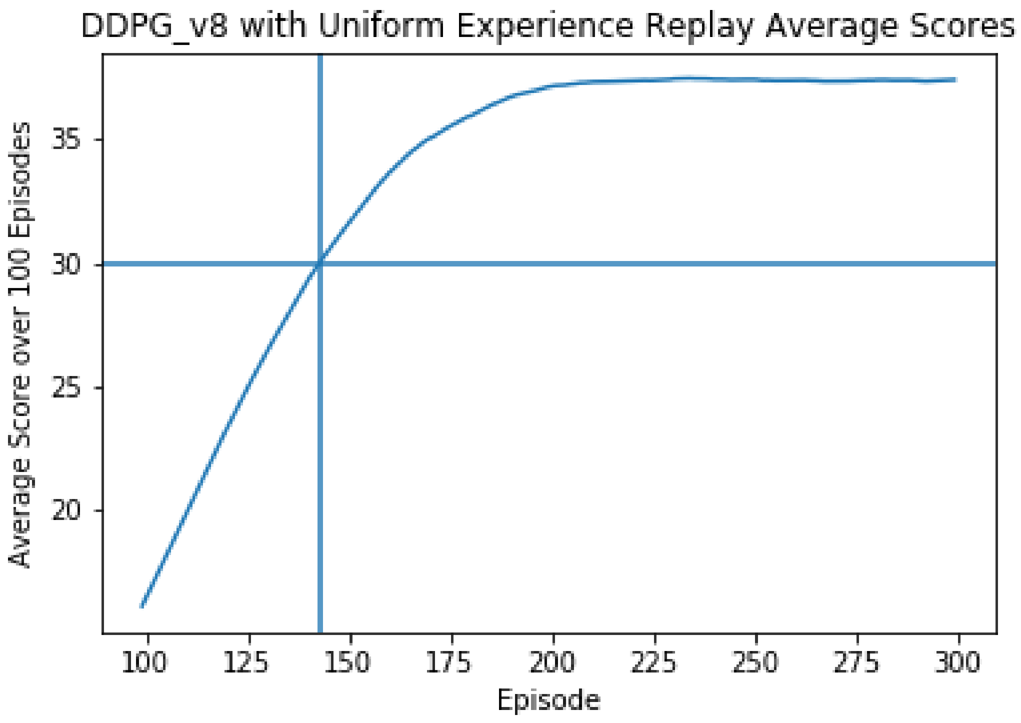

Figure 2: 100-episode-average scores for version 8

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.
The fact that the results changed so dramatically from version 7 to version 8, with such a small change, tells me there's still something almost arcane about this process, if I had a better understanding of how this actually works the proper hyperparameter values would have flowed directly from the form of the environment. Much to learn there still is!