Project 2: Continuous Control

Timothy Bernard

July 2022

1 Problem Description and Algorithm Overview

The purpose of this project was to train a double jointed arm to reach a (moving) target goal. A reward of +0.1 is given for each step where the agent is in the goal location. The observation space is 33 continuous variables consisting of position, orientation and their respective velocities. The action space of the torques applied to the arm and is a continuous 4-tuple. Because this is a problem of continuous control, I used the Deep Deterministic Policy Gradient (DDPG) [1] algorithm which is suitable for continuous action spaces. With DDPG, the idea is to use an "actor-critic like" strategy that builds off of DQN to address non-discrete action spaces. It does not learn a state-value baseline like other actor-critic methods. The question is: how would we get the maximal action (like in DQN) for a continuous action space? In DDPG, the actor directly approximates the optimal policy deterministically (the currently best believed action at a certain state), and not a distribution over actions. The critic then learns to evaluate the optimal action-value function using this result from the deterministic actor network (essentially taking as input (s,a) pairs). DDPG continues to make use of the replay buffer as in DQN to not have correlated data. This is done to stably train the neural networks, which usually depends on independently and identically distributed data. DDPG also uses target networks so that it's not chasing a constantly moving target.

2 Implementation

This algorithm was implemented using Python3 and PyTorch, to train a policy according to DDPG to achieve the goal of +30.0 score across 100 consecutive episodes. The implementation builds off of the ddpg-pendulum and ddpg-bipedal algorithms from the Udacity Deep Reinforcement Learning GitHub repo [2]:

Table 1: **Hyper parameters**

Replay Memory Buffer	100,000
Mini-batch Size	128
Discount Factor	0.99
Tau (for soft update of parameters)	1e-3
Learning Rate Actor	2e-4
Learning Rate Critic	2e-4
Max Number of Episodes	2000
Max Number Time-steps per Episode	1000
L2 Weight Decay	0.0

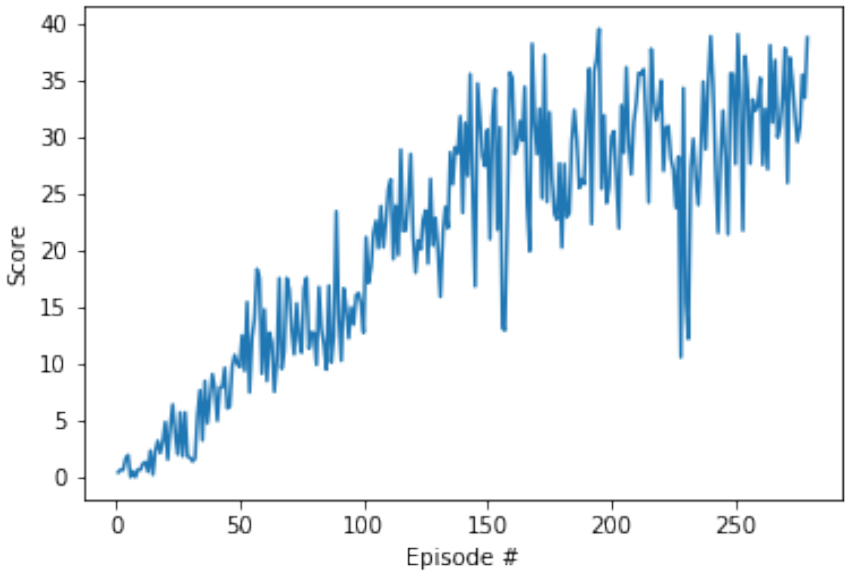
The hyper-parameters were based off of the original implementations, more specifically that of the pendulum example. The learning rate for the critic was decreased by almost an order of magnitude, which

makes the agent change (learn) its hyper-parameters slower. This lower value may have prevented the critic from diverging during training - though choosing correct learning rates can be notoriously finicky. Another noticeable change was increasing the maximum number of time-steps for the episode to allow the agent to learn more from each episode and have the ability to use the current policy to acquire more reward. The Ornstein-Uhlenbeck process was used to introduce noise into the states during training, and I reduced the sigma parameter, to improve training (possibly making sure that not too much noise was introduced). I adjusted the learning rates and the OU sigma parameter with help from folks from the Knowledge center!

The architectures for the actor and critic networks (and therefore their target networks) were fully-connected multi-layer perceptrons. The actor network (to estimate the policy) had one hidden layer with 256 nodes. The critic had 3 layers (256, 256, and 128 nodes respectively). Relu activations were used for introducing nonlinearity and tanh was used at the last layer, forcing the output into the interval $(-1, 1)$ for the actor network. The weights for both were initialized from uniform distributions according to [1].

3 Results

Figure 1: Training Results



The algorithm was able to make the agent reach an average score of 30 over the past 100 episodes in 279 episodes.

4 Future Ideas

One way to improve this agent's future performance would be to do a soft update of the target network(s) after a number of time-steps, instead of at each learning step. This may make training more stable. Another idea is to use batch normalization [3](ensuring unit mean and variance for all samples in the learning batch), which can accelerate training, and was used by the DDPG authors to improve learning across different tasks [1]. One other interesting thing that I did not implement was initializing the non-final layers of the neural networks with uniform distributions on $[-\frac{1}{\sqrt{f_{in}}}, \frac{1}{\sqrt{f_{in}}}]$ where f_{in} is the number of inputs to the layer [1].

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.02971

[2] A. C. et. al, "Deep deterministic policy gradient, pendulum and bipedal," 2018. [Online]. Available: https://github.com/udacity/deep-reinforcement-learning

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015. [Online]. Available: https://arxiv.org/abs/1502.03167