Deep Reinforcement Learning Nanodegree
Project 2 – Continuous Control Report

Chris Cadonic
chriscadonic@gmail.com

March 21, 2021

1 Introduction

As part of this the deep reinforcement learning nanodegree program, this report discusses my work in training an agent to continuously control an arms that aims to reach towards a certain location that changes in 3D space over time.

1.1 Environment Overview

The environment, called *Reacher*, is a continuous control learning condition wherein an arm is tasked to reach up to have the proximal section of the arm to remain in a specified location that may move around in 3D-space. The task of the agent is to reach up and finely adjust its positioning to continue to meet the requirements of being in this location.

For this task, the environment can be summarized as follows:

MDP property	**characteristics**
observation space	vector of length 33, containing continuous values for position, rotation, velocity, and angular velocities
action space	vector of length 4, containing continuous values for torque to be applied to two arm joints
rewards	+0.01 for each step the agent's hand is in the target location

Table 1: Hyperparameters experimented with to train an agent using DQN.

With these characteristics, the agent can learn how to maximize the amount of reward by adjusting joint torques such that it learns how to keep it's hand in the target position as often as possible. An example image to illustrate the environment utilized for this project is shown in Figure 1, where 20 separate agents are trained in parallel to learn how to properly move an arm (white) such that its hand (blue blobs) remain in the green target locations.

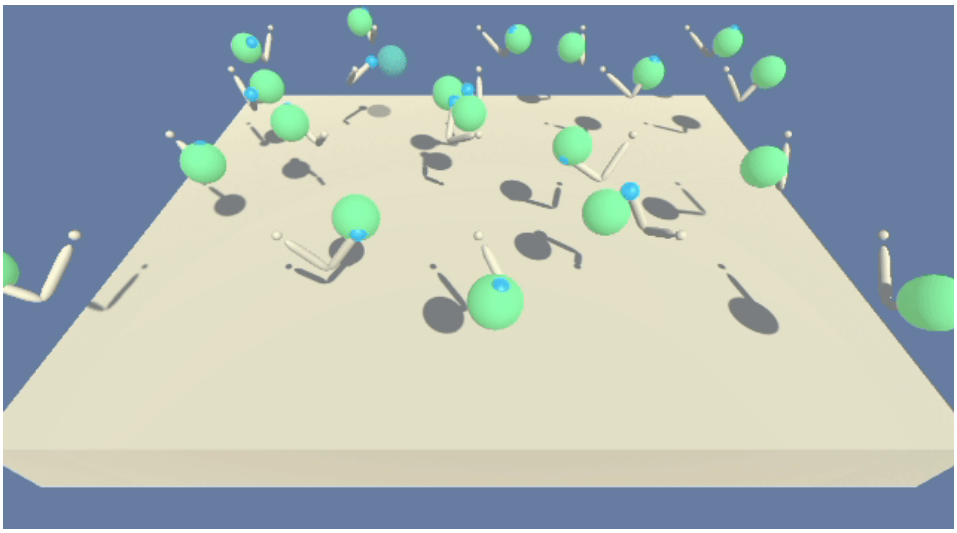

Figure 1: A snapshot of the continuous control Unity-ML Reacher environment. White arms are agent-controlled (20 independent agents shown here), aiming to contain the blue hand in the green target locations.

2 Approach

The overall approach used herein was to utilize the *Deep Deterministic Policy Gradient* (DDPG) algorithm. This algorithm is explained in more detail in the following section.

2.1 DDPG

This algorithm is a policy gradient algorithm that adapts Q-learning for a continuous action space, wherein both a Q-function approximation is learned alongside a serially adapted policy. Similar to the DQN algorithm, target networks are utilized in learning both an *actor* network and a *critic* network drawing from experiences in an expanding experience replay buffer.

Here, the *actor* network is responsible for leveraging the approximation of the Q-function to determine an optimal action per time step. In this specific instance, the Q-function is represented not as a tabular matrix, due to the continuous nature and under the assumption that the Q-function is differentiable, instead the actions are selected as a continuous maxima on the Q and utilize gradient ascent.

Additionally, the *critic* network is only concerned with improving the Q-function approximation, whereby the network evaluates the value of taken actions against the value of predicted actions suggested by the *actor* network.

Finally, a unique addition to the DDPG algorithm in comparison to other deep network algorithms for handling exploration vs. exploitation regards how noise is introduced to action values. The standard DDPG algorithm utilized *Ornstein-Uhlenbeck* noise processes [**?**], which is a mean-regressing noise process that results in a dampening normal noise over a time period. Specifically, over each training episode the noise injected to action values are slowly dampened over an episode to revert to noise generated around a tight Gaussian with mean 0.

2.2 Implementation

The specific implementation of DDPG completed in this work took two major forms, one to implement the standard DDPG algorithm as detaile din [**?**], and another that adapts this to learn from multiple agents in parallel using a shared experience replay buffer.

The former involves the following characteristics:

- both *actor* and *critic* networks utilized a baseline and target network pairing during training,

where target networks are updated using a per-step soft-update schedule with ratio τ,

- the *actor* network seemed to benefit from a simpler architecture in comparison to the *critic* network,
- using a repeatable update schedule per step (i.e., the networks can be updated m times per step) along with learning rate α,

Using these implementation details on top of the DDPG algorithm, I settled on the following hyperparameters:

hyperparameter	**utility**	**value**
α	learning rate	0.001
τ	target network soft-update ratio	0.001
γ	discount factor on future returns	0.99
t update	number of steps to take before updating networks	1
num updates	number of updates to complete each time networks are updated during training	2
μ	regression mean for Ornstein-Uhlenbeck noise	0.0
θ	factor for weighing the delta of value from mean	0.15
σ	factor for weighing the Weiner (Gaussian) noise process	0.2

Table 2: Hyperparameters experimented with to train an agent using DQN.

Values were either determined experimentally, such as by varying learning rate α, τ, γ, and the integers for the update schedule, or utilized from the original DDPG paper [**?**] as in the parameters for the noise process μ, θ, and σ.

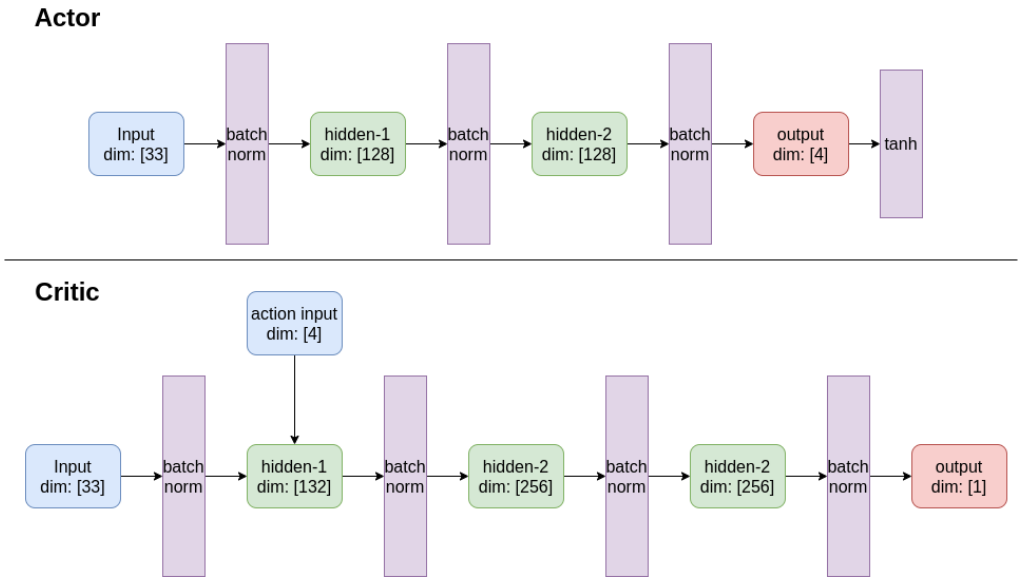

Figure 2: The architectures used in the final *actor* and *critic* networks in the DDPG algorithm. Though not specified, all activation functions are ReLU unless otherwise noted.

Lastly, different variations on architecture were tested, with major architectural changes primarily affecting the bound of steady-state long-term performance of the agents. For best results, it appeared

that a two hidden layer *actor* network and a two to three layer *critic* network produced fairly stable results. Architectures are outlined above in Figure 2.

3 Results and Discussion

3.1 Learning Performance

Figure 3

- $\epsilon_{decay} = 0.996$,
- $\alpha = 0.001$,
- $\tau = 0.01$,
- $\gamma = 0.99$.

3.2 Next Steps

4 Conclusion

References