Continuous Control with Deep Reinforcement Learning

Ismail Geles

April 2020

Abstract

The second project of the Udacity Deep Reinforcement Learning Nanodegree is about an Unity environment (named Reacher) with 20 robotic arms (multiple identical agents). To solve the environment a DDPG agent was trained as specified in this report.

1 The Environment

The identical agents to train should master positioning 20 double-jointed robotic arms on a target location each.

1.1 States and Actions

The observation space (state space) consists of 33 dimensions and contains an agent's position, rotation, velocity and angular velocities of the arm.
Given this information, the agent has to learn how to best select actions.
The action an agent can take is described by a vector with four real numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.
There are infinitely many actions available, therefore the action-space is considered continuous.
The task is episodic and is solved when the average of all agents reaches a score of +30 over 100 consecutive episodes.

2 Implementation of the DDPG Agent

The codes basic structure was taken from previous tasks of the Nanodegree. A DDPG agent, as described in the initial paper "Continuous control with deep reinforcement learning" (`arXiv:1509.02971`), was used.

The 20 robotic arms are basically controlled by one agent, consisting of an actor network and a critic network, each having a regular and a target network. The critic is used to approximate the maximizer of Q-Values of the following states. That is why DDPG is a renowned Deep Q-Learning approach for continuous actions spaces. The experiences are stored in a replay buffer and updates to the networks are soft.
All 20 arms are using the same actor and critic networks, which learn from a shared replay buffer, storing experiences of all arms. The expected reward was discounted and with the Ornstein-Uhlenbeck process noise was added to the actions in order to bring in some randomisation.

2.1 Network Architecture

The model, used to solve the environment, has two identical network architectures for the actor and critic. After trying out some architectures the best results were achieved with two hidden layers consisting of 128 units each.

So to summarize the layer structure of both networks (in `model.py`):

33 units (input) - 128 units (1st hidden) - 128 units (2nd hidden) - 4 units (output)

The activation function for the hidden layers used, is a Rectified Linear Unit (ReLU) function for both networks. The actor networks output is mapped to a hyperbolic tangent function in order to assure values between $[-1, 1]$.

2.2 Network Hyper-parameters

Following hyper-parameters were used for training:

Parameter	Value
buffer size	1e5
batch size	128
gamma (discount factor)	0.99
tau (soft update interpolation)	1e-3
learning rate actor	5e-4
learning rate critic	5e-4
update rate	1
max episodes	2000
weight decay	0.0

Note: after trying out different update rates and weight decay it was better to update the networks at every timestep and use no weight decay at all. An episode is considered finished if any of the robotic arm agents is done.

3 Training Results

The rewards over the number of episodes can be found below. Notice that the environment was solved at episode 23, following an average over +30 for the next 100 episodes.

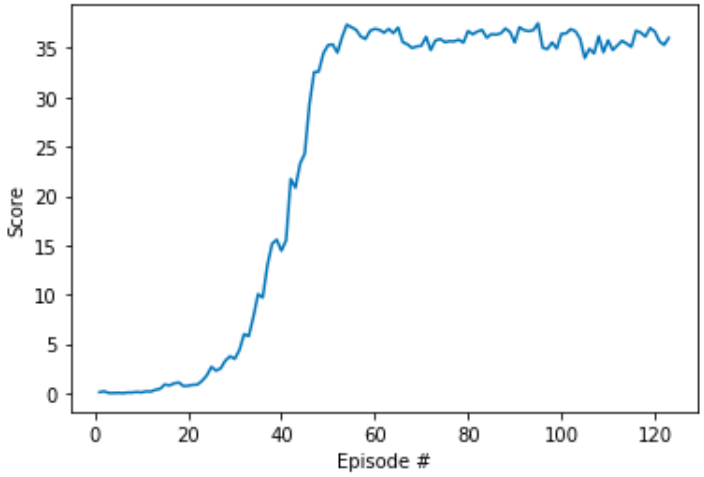

Figure 1: Average scores of all 20 agents over episodes during training.

4 Ideas for Future Work

There are still many possibilities of network architectures and hyper-parameters to try out for better performance. Another approach to tackle this problem would be solving the environment with other actor-critic methods, like A2C, A3C, D4PG, PPO,..., especially since this environment offers a simple method to control all the 20 agents in parallel.