Project 2: Continuous Control

1. Introduction

The Continuous Control project has a double-jointed arm (the agent's arm) that can move to target locations. The agent is given a reward of +0.1 for each step that the agent's hand is in the target or goal location. Thus the goal is for the agent to maintain its position at the target location for as many time steps as possible. Note this environment is continuous and here we solve the multi-agent environment.

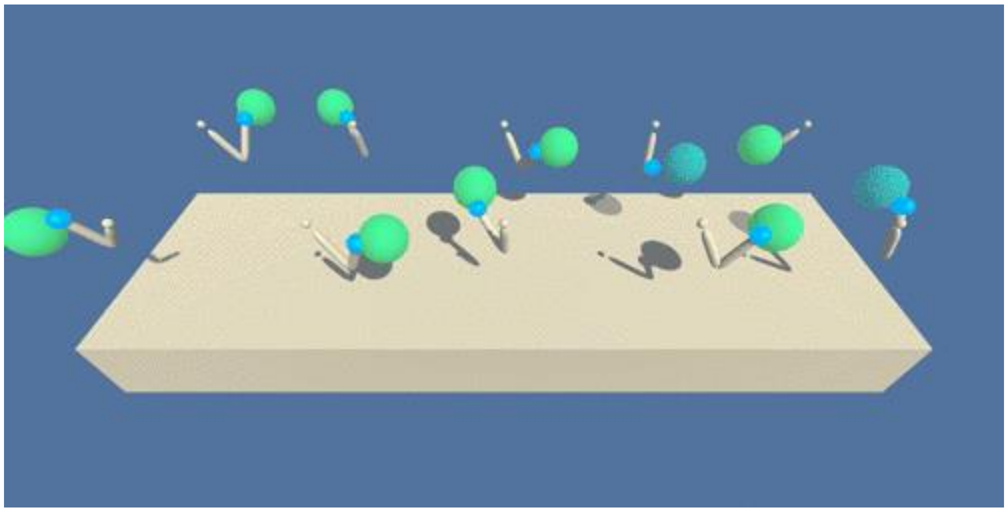


Figure 1 The Reacher Environment

We will investigate, implement and assess the Deep Deterministic Policy Gradient (DDPG) deep reinforcement learning algorithm that maximize rewards over training.

We are going to train an agent to keep its arm in the target location for as many times as possible using the DDPG algorithm.

We have a reinforcement learning setup with states, actions, rewards between the agent and the environment.

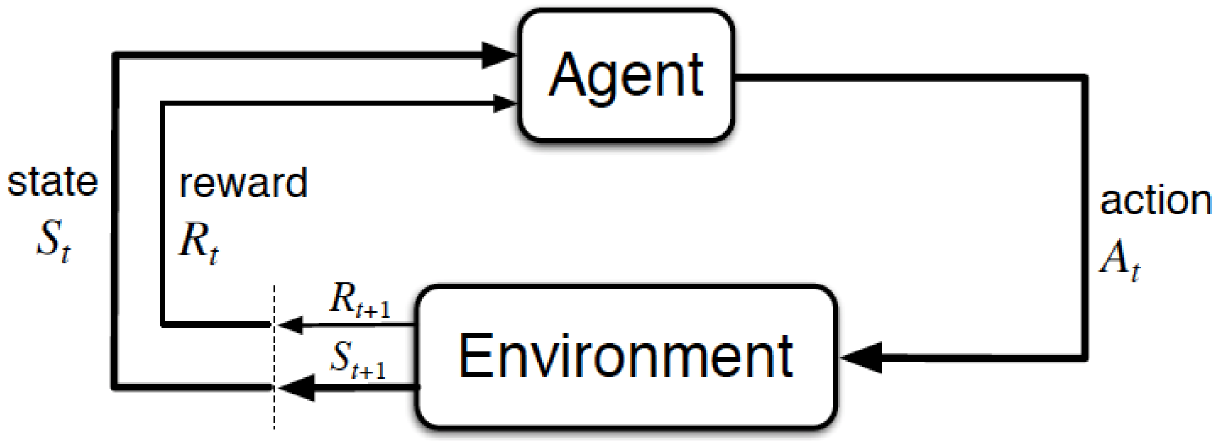


Figure 2 The agent-environment interaction in reinforcement learning. (Source: Sutton and Barton, 2017)

Rewards

A reward of +0.1 is awarded for keeping the arm in the target location for as many time steps as possible. Thus the goal of our agent is to maintain its position in the target location for as many times as possible and in the process get an average score of +30 over 100 consecutive episodes.

States

The state space has 33 variables corresponding to position, rotation, velocity and angular velocities of the arm.

Actions

Each action is a vector with four numbers corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

We finish this project by looking at potential extensions to the above algorithms.

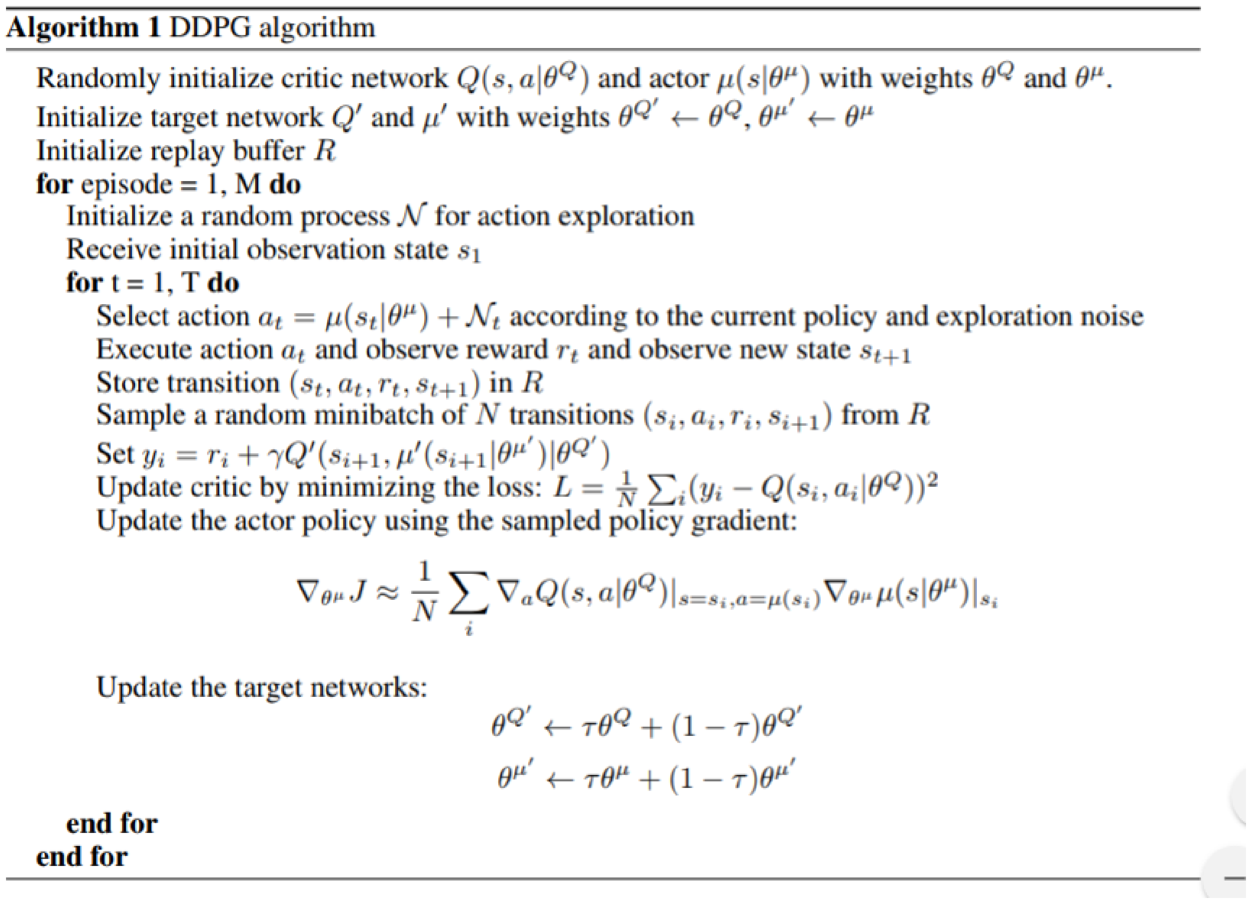
2. Learning Algorithm

The deep reinforcement learning algorithm implemented is Deep Deterministic Policy Gradient with a Replay Buffer and Noise.

Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient algorithm is a model-free approach to reinforcement that can learn action policies directly from low-dimensional observations using an Actor-Critic network as an underlying deep learning architecture.

The DDPG algorithm is an extension of the Deterministic Policy Gradient (DPG) algorithm (Silver et al., 2014). The algorithm itself is as follows:



Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Figure 3 DDPG Algorithm

The algorithm has a number of features and or attributes:

An Actor-Critic Network

Function approximation is done through a parameterized Actor function which specifies the current policy by deterministically mapping states to a specific action and the Critic function is learnt using the Bellman equation as in Q-Learning. The Actor continuously updates the policy while the Critic evaluates this off-policy. Here I have implemented the Actor-Critic networks as three layer neural nets:

```
Actor(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=400, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=4, bias=True)
)
Critic(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=404, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=1, bias=True)
)
```

I found that using the exact same network for both Actor and Critic networks improved agent training.

Replay Buffer

Just like in Deep Q-Network (DQN) we use a replay buffer to enable the agent to learn from experience. The replay buffer stores transitions sampled from the environment according to our exploration policy, discussed below. The additional power of this replay buffer is that multiple agents can share experiences and benefit from learning across a set of uncorrelated transitions.

Noise

In their paper, the Google Deepmind team, introduced a Noise component to address exploration. Exploration is a major challenge of learning in continuous action spaces. Since DDPG is off-policy the problem of exploration can be addressed independently from the learning problem.

DDPG introduces an exploration μ' by adding noise sampled from a noise process Ņ to the actor policy:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

Ņ can be chosen to suit the environment. Here we implemented an Ornstein-Uhlenbeck process to generate temporally correlated exploration for exploration efficiency. The agent samples this noise during training to enable exploration.

3. Ideas for future work

- Implement and potentially combine other algorithms like Distributed Distributional Deterministic Policy Gradient (D4PG), Proximal Policy Optimization (PPO) and Asynchronous Advantage Actor-Critic (A3C). My hypothesis is that these algorithms could be combined just like DQN, Duelling DQN and others to create another algorithm like Rainbow.

- Hyper-parameter tuning

  I started tuning the hyper-parameters for this algorithm, adding more layers, drop out and batch normalization didn't improve the speed of training or convergence. I can tune these a bit more using grid search.