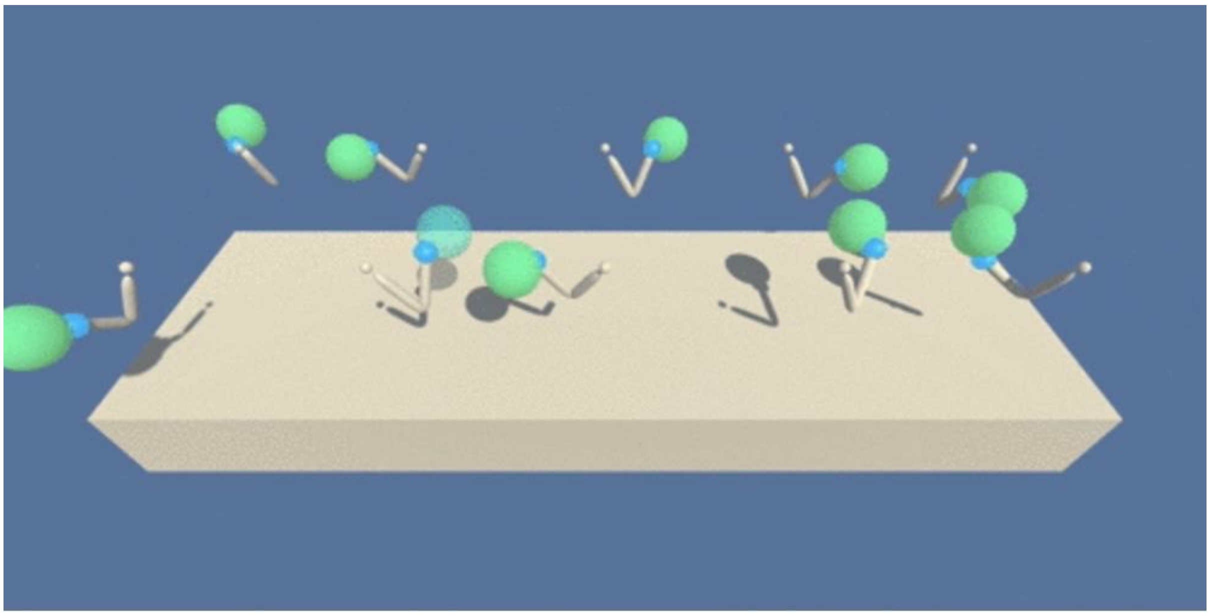
DDPG – Continuous Control

1. Introduction

In this project, we will work with the Unity ML-Agents Reacher environment, which in this version contains 20 identical agents, each with its own copy of the environment.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of our agent is to maintain its position at the target location for as many time steps as possible.

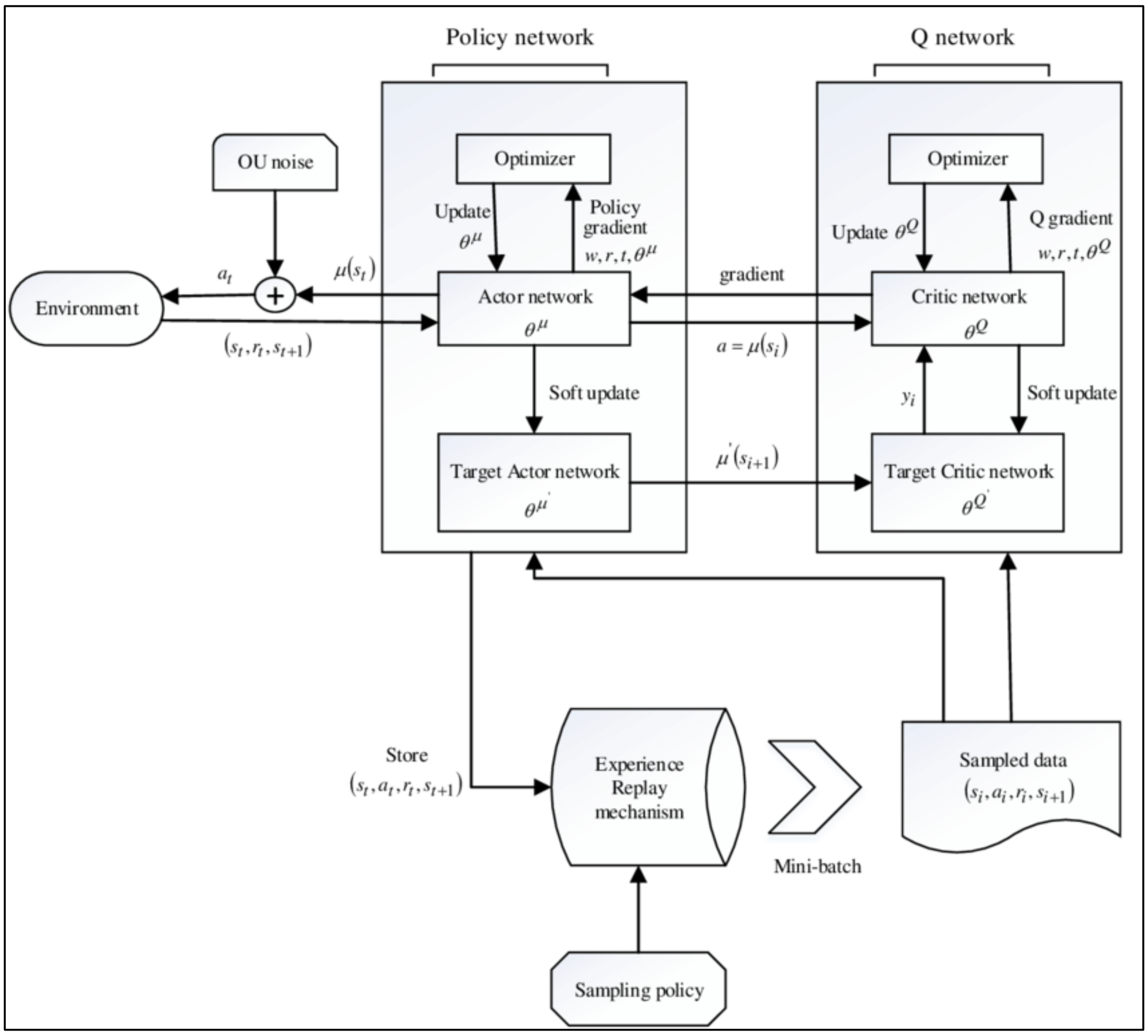

Example of Unity ML-Agent Reacher environment with ten agents

The observation space, for each agent, consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. 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.

2. Implementation

2.1. Learning Algorithm

The learning algorithm in this project is Deep Deterministic Policy Gradients (DDPG) and it was implemented using PyTorch. The image below illustrates the main components of DDPG.



Policy network
Q network
OU noise
Optimizer
Update θ^{μ}
Policy gradient w, r, t, θ^{μ}
Optimizer
Update θ^{Q}
Q gradient w, r, t, θ^{Q}
a_t
$\mu(s_t)$
Environment
Actor network θ^{μ}
gradient
Critic network θ^{Q}
(s_t, r_t, s_{t+1})
$a = \mu(s_i)$
Soft update
y_i
Soft update
Target Actor network $\theta^{\mu'}$
$\mu'(s_{i+1})$
Target Critic network $\theta^{Q'}$
Store (s_t, a_t, r_t, s_{t+1})
Experience Replay mechanism
Mini-batch
Sampled data (s_i, a_i, r_i, s_{i+1})
Sampling policy

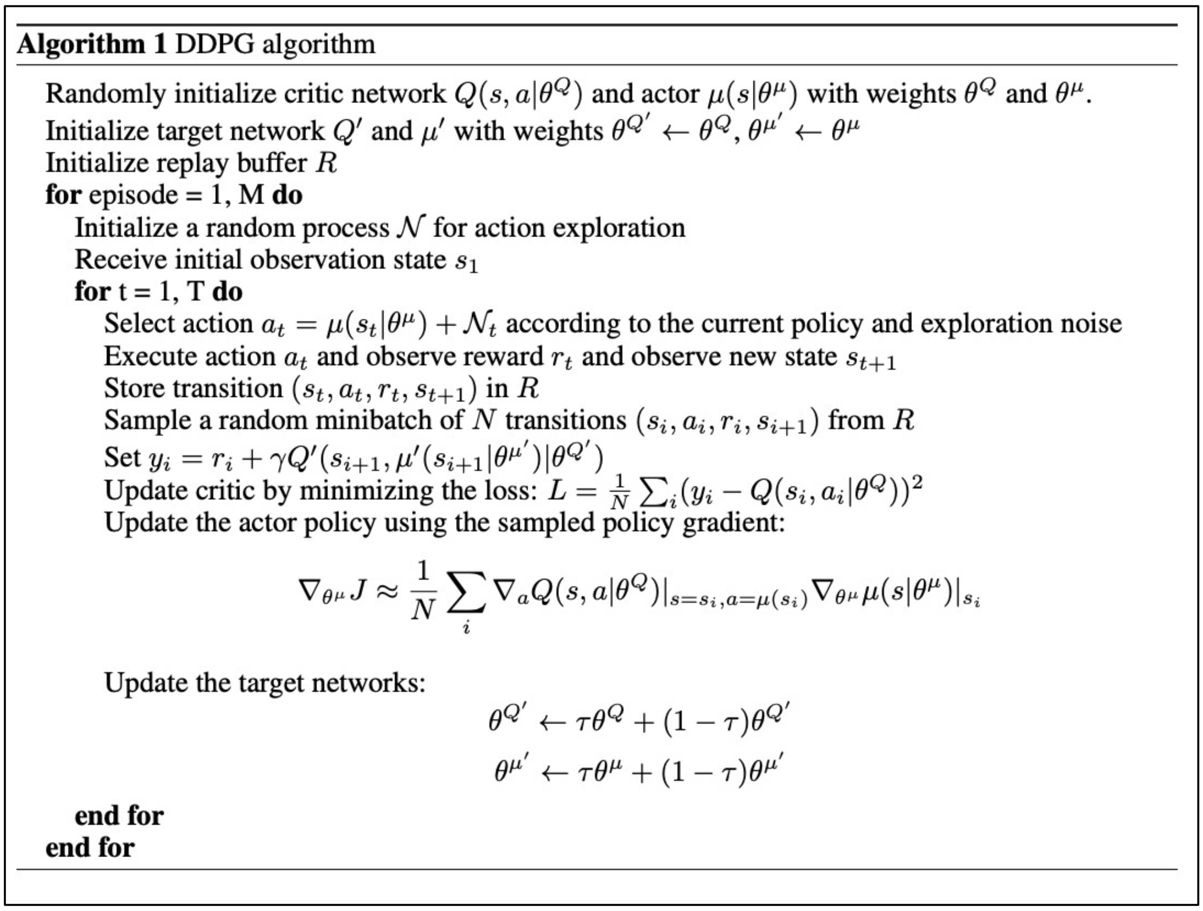
image source: The DDPG algorithm structure (Kang *et al.*, 2021, p. 3)

DDPG algorithm is a model-free, online, off-policy reinforcement learning method which concurrently learns a Q-function and a policy. A DDPG agent is based in the actor-critic model that searches for an optimal policy that maximises the expected cumulative long-term reward.

DDPG uses two target networks to add stability to training by learning from estimated targets and updating target networks slowly.

2.2. DDPG Code Break Down

The pseudocode below describes the DDPG algorithm's process.



```
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
```

Pseudocode: DDPG Algorithm (Lillicrap *et al*., 2015, p.5)

Five helper classes are used to implement the DDPG algorithm:

3. Class “Actor” implements the Neural Network which is composed by the following layers:
   - 33 input neurons
   - 400 Fist hidden layer neurons
   - 300 Second hidden layer neurons
   - 4 Output neurons

The “forward” method builds a network that maps state to action values using RELU and TAHN as the activation functions.

Class “Critic” implements the Neural Network which is composed by the following layers:
- 33 input neurons
- 400 Fist hidden layer neurons
- 300 Second hidden layer neurons
- 4 Output neurons

The “forward” method builds a network that maps state to action values using RELU as the activation function.

Class “ReplayBuffer” implements the experiences store and contains two main methods, “add” and “sample”, which add new experiences to the memory and randomly sample a batch of experiences respectively, and a helper "_len_" method that returns the current size of the internal memory to be used in the "Agent" class when determining if there are enough experiences collected compared to the set BATCH_SIZE.

Class "OUNoise" implements the Ornstein-Uhlenbeck noise process to construct an exploration policy as seen in Lillicrap *et al.* (2015) implementation. Methods of this class are "reset" that resets the internal state (= noise) to mean (mu), and "sample" where it updates the internal state and returns it as a noise sample.

Class "Agent" interacts and learns from the environment. The class initialises the Actor and Critic local and target networks and sets Adam as optimiser. Methods of this class are "Step" which saves experiences in the replay buffer and uses random sample from buffer to learn, "Act" which returns actions for given states as per current policy, "Reset" which resets and updates the internal state, and returns a noise sample, "Learn" which updates policy and value parameters using given batch of experience tuples, and "soft_update" which updates the model parameters.

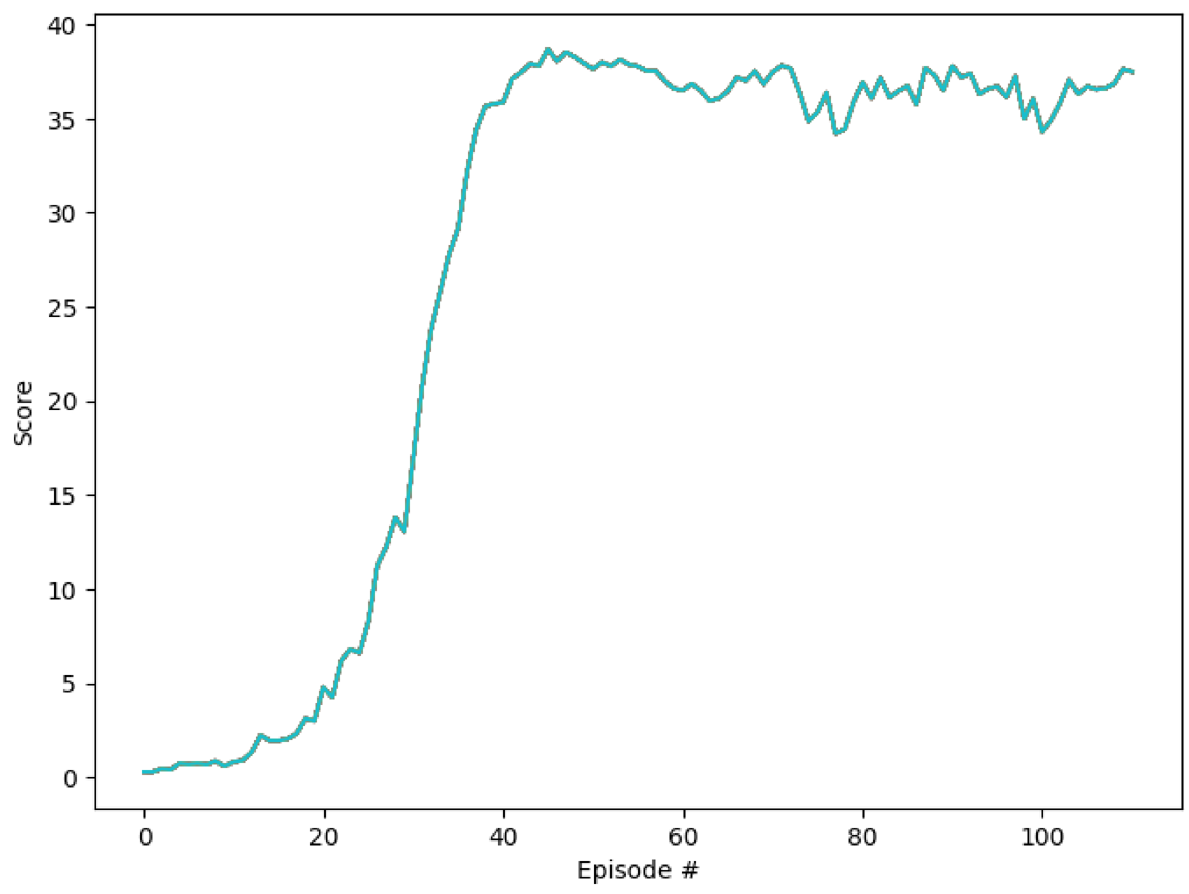
3.1. Hyperparameters

The table below shows the chosen hyperparameters, decided by a combination of trial and error and of continuous control baselines in bibliography.

Name	**Value**	**Description**
BUFFER_SIZE	1e5	Replay buffer size
BATCH_SIZE	128	Minibatch size
GAMMA	0.99	Discount factor
TAU	1e-3	Soft update of target parameters
LR	1e-4	Learning rate for actor and critic
WEIGHT_DECAY	0.	L2 weight decay

2.3 Results

The plot below illustrates the agent's rewards per episode. The agent was able to meet that target of +30 average reward across all 20 agents over 100 episodes after just above 100 episodes.



3 Future Work

Using a hyperparameters tuning tool might increase the agent's performance. For example, Tune, a Ray library, can run experiments in parallel and once it has converged in the best performing hyperparameters it can continue the training loop using them.

Also, it would be beneficial to experiment and compare the performance of different algorithms in this environment such as Proximal Policy Optimization (PPO), Asynchronous Advantage Actor Critic (A3C) and Distributed Distributional DDPG (D4PG) that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

4 References

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. & Wierstra, D. (2015). "*Continuous Control with Deep Reinforcement Learning*". Available at: https://doi.org/10.48550/arXiv.1509.02971.

Kang, C., Rong, C., Ren, W., Huo, F. & Liu, P. (2021). "*Deep Deterministic Policy Gradient Based on Double Network Prioritized Experience Replay*". IEEE Access. PP. 1-1. Available at: https://doi.org/10.1109/ACCESS.2021.3074535.