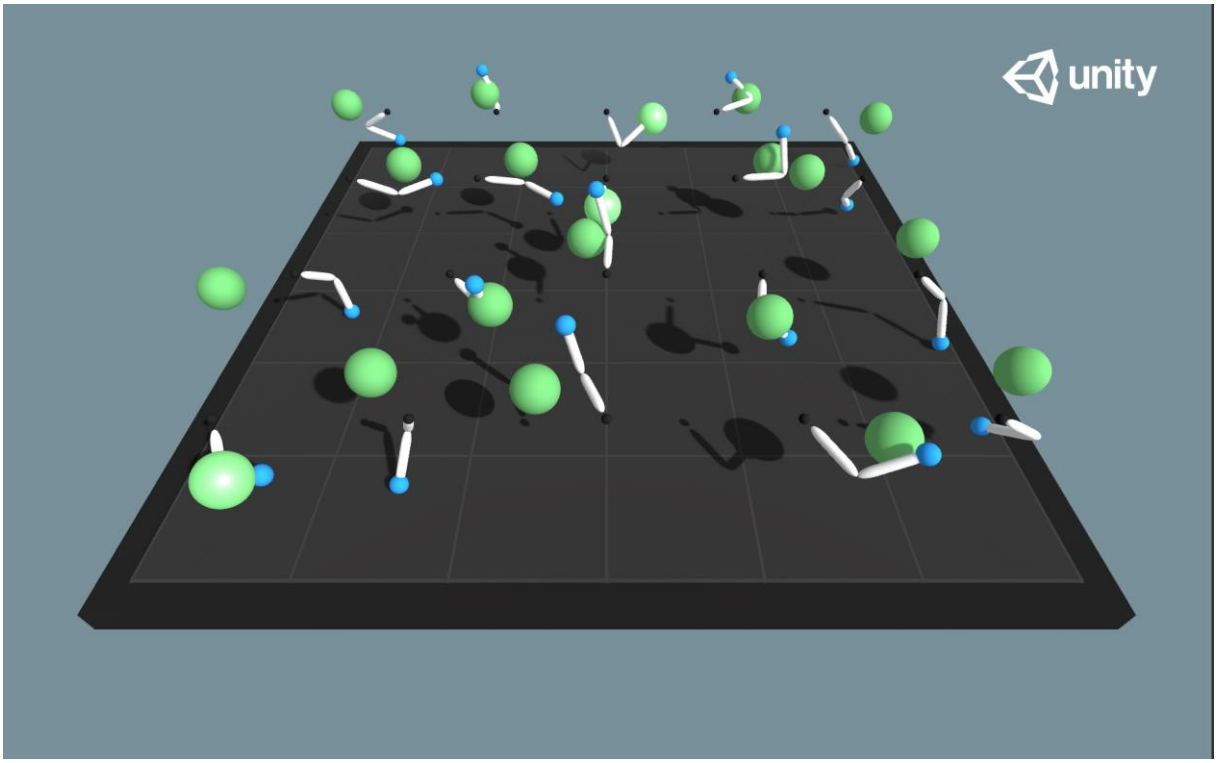
Project 2: Continuous Control

Author : Engin Bozkurt

The general overview and goals of the project

The project demonstrates how policy-based methods can be used to learn the optimal policy in a model-free Reinforcement Learning setting using a Unity environment, in which 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 the agent is to maintain its position at the target location for as many time steps as possible.*

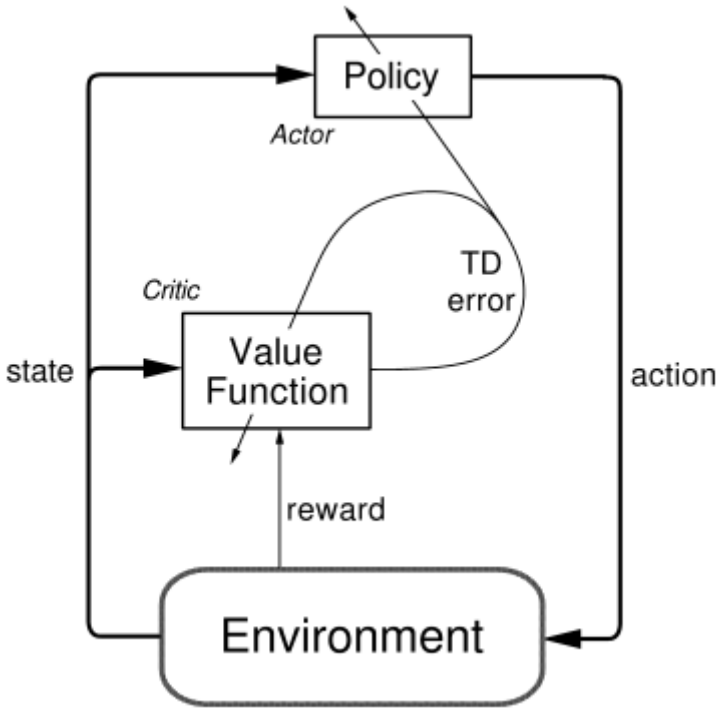
The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector is a number between -1 and 1.



Implementation

Actor-Critic Algorithms

The Actor-Critic learning algorithm is used to represent the policy function independently of the value function. The policy function structure is known as the *actor*, and the value function structure is referred to as the *critic*. The actor produces an action given the current state of the environment, and the critic produces a TD (Temporal-Difference) error signal given the state and resultant reward. If the critic is estimating the action-value function Q(s,a), it will also need the output of the actor. The output of the critic drives learning in both the actor and the critic. In Deep Reinforcement Learning, neural networks can be used to represent the actor and critic structures.



DEEP DETERMINISTIC POLICY GRADIENT (DDPG) Algorithm

At its core, DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a **deterministic** target policy, which is much easier to learn.

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

$$\text{Parameters:}$$

$$\theta^{Q} : \text{Q network}$$

$$\theta^{\mu} : \text{Deterministic policy function}$$

$$\theta^{Q'} : \text{target Q network}$$

$$\theta^{\mu'} : \text{target policy network}$$

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning. Here's why: In methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to divergence. For example:

This depends Q function itself (at the moment it is being optimized)

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma max Q(s',a') - Q(s,a)]$$

So, we have the standard Actor & Critic architecture for the deterministic policy network and the Q network:

Here's the pseudo-code of the algorithm that we want to implement:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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

Algorithm Description

Choosing an action

Pass the current states through the actor network, and get an action mean vector μ. While in training phase, use a continuous exploration policy, such as the Ornstein-Uhlenbeck process, to add exploration noise to the action. When testing, use the mean vector μ as-is.

Training the network

Start by sampling a batch of transitions from the experience replay.

- To train the **critic network**, use the following targets:

  $y_t = r(s_t, a_t) + \gamma \cdot Q(s_{t+1}, \mu(s_{t+1}))$

- First run the actor target network, using the next states as the inputs, and get $\mu(s_{t+1})$. Next, run the critic target network using the next states and $\mu(s_{t+1})$, and use the output to calculate y_t according to the equation above. To train the network, use the current states and actions as the inputs, and y_t as the targets.

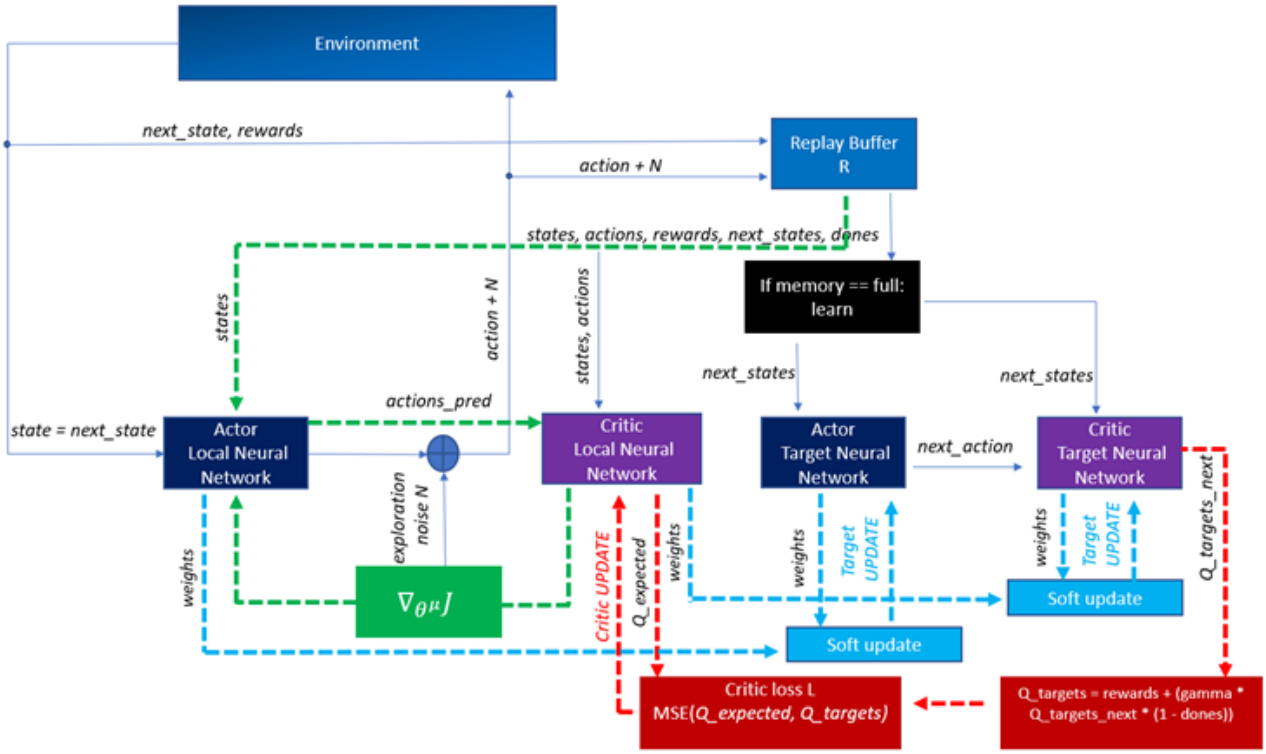
- To train the **actor network**, use the following equation:

  $\nabla_{\theta_\mu} J \approx E_{s_t \sim \rho_\beta}[\nabla_a Q(s,a)|_{s=s_t, a=\mu(s_t)} \cdot \nabla_{\theta_\mu} \mu(s)|_{s=s_t}]$

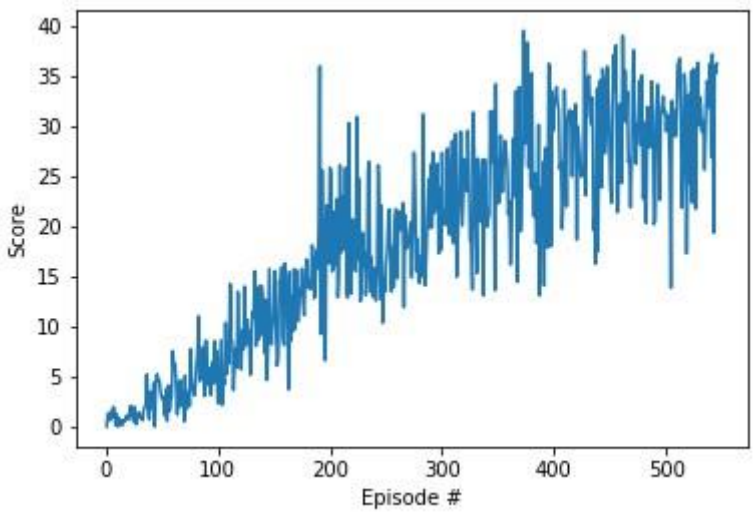
- Use the actor's online network to get the action mean values using the current states as the inputs. Then, use the critic online network in order to get the gradients of the critic output with respect to the action mean values $\nabla_a Q(s,a)|_{s=s_t, a=\mu(s_t)}$. Using the chain rule, calculate the gradients of the actor's output, with respect to the actor weights, given $\nabla_a Q(s,a)$. Finally, apply those gradients to the actor network.

After every training step, do a soft update of the critic and actor target networks' weights from the online networks.

General overview of DDPG algorithm flow:



Results



A smaller learning rate for critic leads to actor overfitting. If the policy loss falls weirdly fast, I changed the LR, because the actor is overfitting.

In terms of hyperparameters, I used, the learning rate for both the actor and critic network was 1e-4.

***Soft Updates**: In DQN, the target networks are updated by copying all the weights from the local networks after a certain number of epochs. However, in DDPG, the target networks are updated using soft updates where during each update step, 0.01% of the local network weights are mixed with the target networks weights, i.e. 99.99% of the target network weights are retained and 0.01% of the local networks weights are added.*

***Experience Replay**: In ER, we maintain a Replay Buffer of fixed size (say N). We run a few episodes and store each of the experiences in the buffer. After a fixed number of iterations, we sample a few experiences from this replay buffer and use that to calculate the loss and eventually update the parameters. Sampling randomly this way breaks the sequential nature of experiences and stabilizes learning. It also helps us use an experience more than once.*

Environment solved in 445 episodes! Average Score: 30.01

Hyperparameters

Replay buffer size	1e5
Batch size	128
Discount factor (Gamma)	0.99
Tau	1e-3
Actor Learning rate	1e-4
Critic Learning rate	1e-4
The start rate of epsilon	1.0
The end rate of epsilon	0.05
The decay rate of epsilon	3e-5
Number of episodes	1000

Looking Forward – Further Improvement

Some ways to potentially get better performance :

1.Using a priority algorithm for sampling from the replay buffer instead of uniformly sampling. Prioritized Experience Replay.

2.Experimenting with different stochastic policies to improve exploration.

3.Using recurrent networks to capture temporal nuances within the environment.

4. Using the Proximal Policy Optimization algorithm