P2- ContinuousControl project

The presented solution employs the *Deep Deterministic Policy Gradient* (*DDPG*) method.

This method is a model-free off-policy learning algorithm suitable for continuous actions. It belongs to the groups of actor-critic methods, which extends policy gradient and q-values methods. Two independent networks are used for the actor and the critic. The former provides the best action given the state, and the latter provides the q-value for a pair (state, action).
These two networks are used in combination to learn the best policy by proposing a policy (actor) and criticising the results using the q-values (critic).

Because the critic uses a DQN, the algorithm can be seen as an extension of DQN to continuous action space. Similarly to DQN, DDPG approximate the Q-table, and mitigation techniques must be considered to avoid instabilities. DDPG uses a replay-buffer and a target network. The former stores experience coming from past episodes in the shape of tuples (state, action, reward, next_state, done), the latter mitigates the problem of learning by using a moving target, and represents a copy of the network updated with a certain delay.

Both actor and critic use two networks: a target and a local.

A common method to balance exploitation and exploration is using epsilon-greedy policies. This method uses the current best policy with probability (1 - epsilon) and explores a uniformly random generated action with probability (epsilon).
However, DDPG commonly substitute this method with a soft-update (polyak method) of the target policy, and a noise to the predicted actions.

DDPG does not learn at avery step. To avoid overfitting, the method learns only a certain number of steps, so that new experience is available in the replay buffer. Moreover, the learning phase is repeated several times once the learning is activated.

Important insights throughout the implementation

Throughout the implementation a few important elements have emerged:

- The **noise** added to the actions makes a huge difference in the learning of the algorithm. Initially, we implemented the Ornstein–Uhlenbeck process, which is inspired by the velocity of Brownian particles. However, the learning was very unstable, and we could not obtain reliable results. Therefore, we moved to the Gaussian noise at zero mean (as suggested byOpenAI). Thanks to this type of noise, the algorithm increased in robustness and stability.
- The **lean every N step** and the **learning per step** numbers make a big impact on the learning phase. Initially, we set (20, 10) respectively, but the networks were very unstable, especially in the first 100 iterations, with a very slow learning rate. Therefore, we increased the number to (50, 40), and the algorithm started to learn faster and more robustly.

RL Hyperparameters
Max number of episodes: 2000

Max time per episode: 1000

Gamma: 0.99

learn every N step: 50

learning per step: 40

Replay Buffer: 1e6

Batch size: 128

Soft update value: 0.9999

NN Hyperparameters

Actor

Network structure: One Regularisation layer, two linear layers with ReLu activation, and a hyperbolic tangent as output layer

First Hidden Layer = 128 nodes

Second Hidden Layer = 128 nodes

Critic

Network structure: One Regularisation layer, two linear layers with ReLu activation, and a linear layer.
This network takes in the state and the action, but the action are fed inside the network only *after* the first linear layer

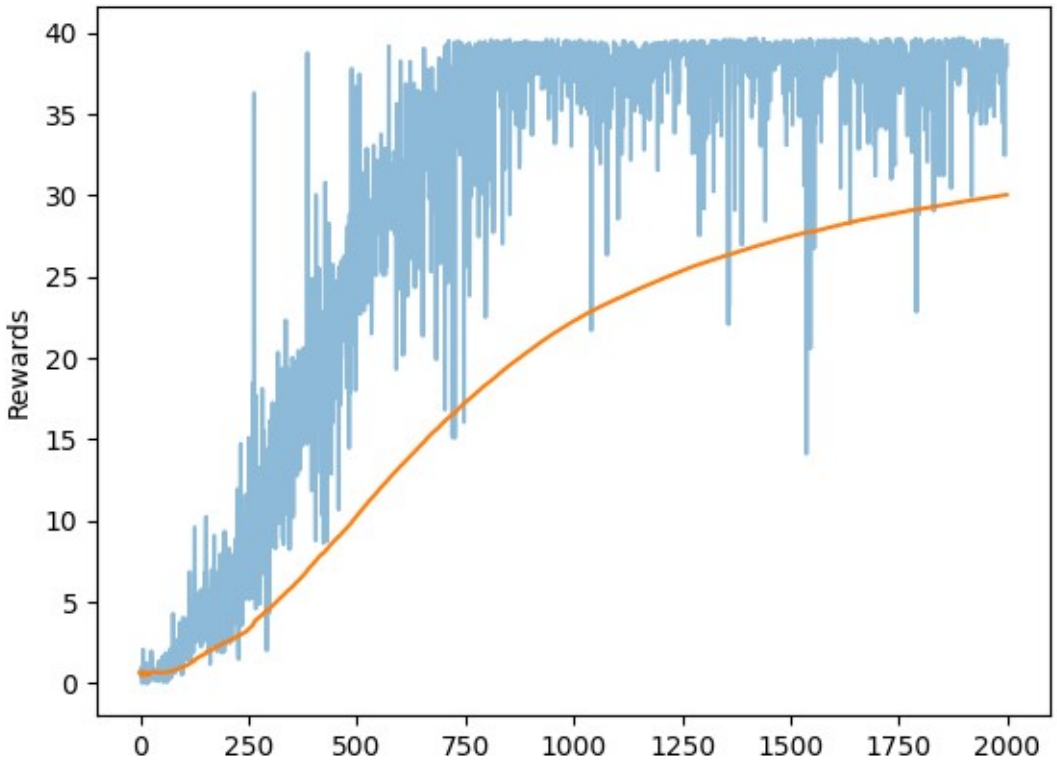
First Hidden Layer = 128 nodes

Second Hidden Layer = 128 nodes

The result

We consider the algorithm finished when an average value of +30.0 is reached.

The learning graph below shows that we reach the target after roughly 500 iterations, the network touches the ceiling of +40 after roughly 750 iterations.



Future work

We can solve the problem with a method like PPO and A3C. Also, we can move from the single to the multiple reachers to leverage on the experience of a bigger number of agents.