Continuous Control Project

Introduction:

The project is about constructing a reinforcement-learning algorithm in order to maintain a double-joined arm position at some target location (see Figure 1). The arm agent can move in the environment by taking continuous actions to target locations. The agent gets a reward of +0.1 for each step the arm is in the goal location. The objective of the algorithm is train the arm agent to stay in the target location as many time steps as possible.

The low-dimensional state space is composed of 33 variables corresponding to position, rotation velocity and angular velocities of the arm. The action is a vector of four variables corresponding to the torque applicable to two joints. The action's vector components are real values between [-1, +1].

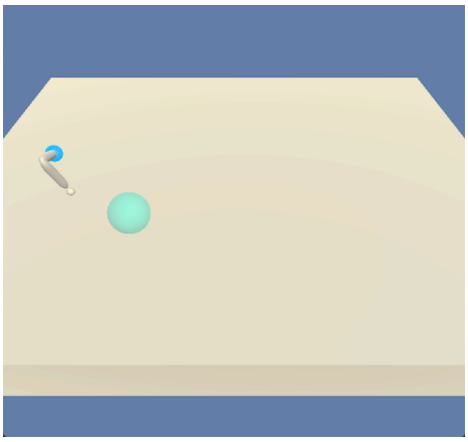


Figure 1. Double-joint arm environment.

In this project, we will consider the task episodic, and will train one single agent. The environment is considered solved when the agent achieves an average score of at least +30 over 100 consecutive episodes.

Learning algorithm:

I attempted solving the environment using the DDPG algorithm. I adapted the code from the lesson and implement it with pytorch.

Implementations:

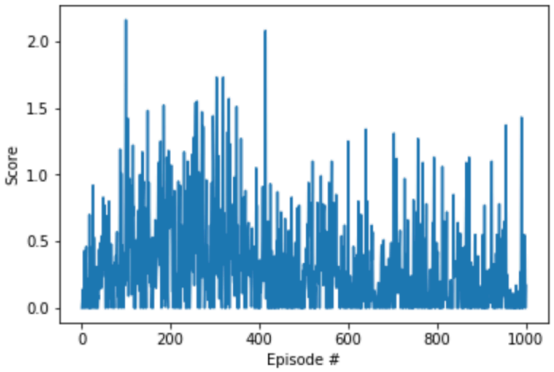
The main functions used to train the agent are: ddpg-agent.py and model.py

The function model.py contains the neural network architectures, which are defined in two separate class objects: Actor and Critic. The number of hidden layers and the number of units in each hidden layer are the same in both architectures. The activation function of all hidden layers is the rectified non-linear function (ReLU). The activation of the last layer of the Actor is the tanh function, while the last layer of the Critic is linear.

In all these experiments, the following hyper-parameters were kept fixed: number of episodes = 1000, internal iterations = 1000, seed = 2, memory size = 10^5, batch size = 64, discount factor (γ) is 0.99, the soft update target factor is 0.001, the learning rates were 1^{-4} and 1^{-3} for Actor and Critic, respectively, the L2 weight decay was left to zero. I added an update every hyper-parameter to control when to update the target network parameters. The number of hidden layers and units, the update_every and the noise parameters σ and θ were changed.

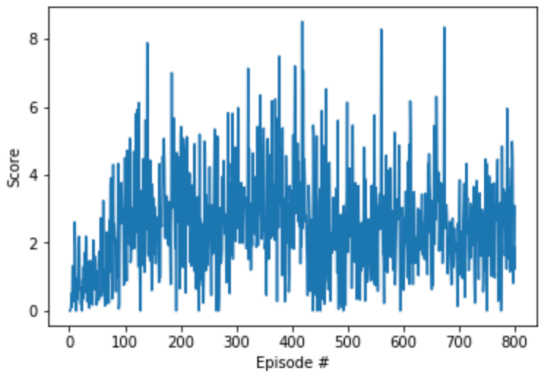
Results:

I conducted different experiences and within every experience I changed the hyper-parameters of the actor-critic neural networks. I also changed the values of the hyper-parameters defined in the baseline DDPG example. Figure 2. Shows the average scores over 100 consecutive episodes. Under each subfigure, I provide a table of the hyper-parameters that I used to train the agent.



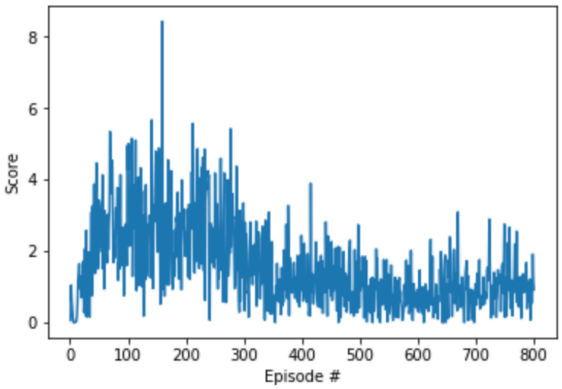
- Actor: 2 hidden layers (64, 64)
- Critic: 2 hidden layers (64, 64)
- Update_every = 1
- OUNoise: $\sigma = 2.0, \theta = 1.0$

Difficulty with the agent to learn.



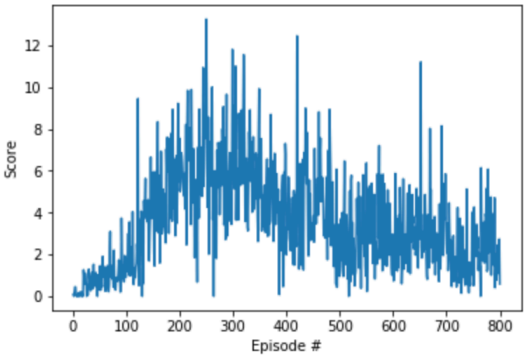
- Actor: 3 hidden layers (128, 128, 128)
- Critic: 3 hidden layers (128, 128, 128)
- Update_every = 5
- Use layer normalization after each hidden layer
- OUNoise: $\sigma = 2.0, \theta = 1.0$

The agent started learning better but far from getting the ultimate score (+30)



- Actor: 3 hidden layers (128, 128, 256)
- Critic: 3 hidden layers (128, 128, 256)
- Update_every = 1
- Use layer normalization after each hidden layer
- OUNoise: $\sigma = 0.2, \theta = 0.15$

Update the target networks every step was not a good idea.



- Actor: 3 hidden layers (128, 128, 256)
- Critic: 3 hidden layers (128, 128, 256)
- Update_every = 5
- Use layer normalization after each hidden layer
- OUNoise: $\sigma = 0.2, \theta = 0.15$

This the best result obtained, though the environment is not resolved yet. The agent is still far from 30 scores

Figure 2. DDPG scores at different hyper-parameters

Conclusion:

We noticed that training with one single agent was a difficult task. I attempted to adapt the DDPG algorithm to train a multi-agent environment, but this was not possible due to the limited computation resources available. The implementations were done using CPU with a macOS laptop. The conducted experiences using one single agent led me to conclude that the actor-critic architectures are the most important part of the DDPG algorithm that we need to care about for improving the agent's performance. Also, including batch/layer normalization after each hidden layer of the actor and critic neural networks helped boosting the results and providing better stability of the scores.