Udacity Reacher Project Report – June 2020

1. Introduction

In the Unity Reacher 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.

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 should be a number between -1 and 1.

Option 1: Solve the First Version with 1 agent
The task is episodic, and to solve the environment, your agent must get an average score of +30 over 100 consecutive episodes.

Option 2: Solve the Second Version with 20 parallel agents
The barrier for solving the second version of the environment is slightly different because it involves 20 agents. Agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically, after each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores. This yields an average score for each episode (where the average is over all 20 agents). The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

I pursued Option 2 in my project.

2. Model

The Deep Deterministic Policy Gradient (DDPG) algorithm is an off-policy, policy gradient-based algorithm that uses two separate deep neural networks (one actor, one critic) to explore the stochastic environment and learn the best policy to achieve maximum reward.

The DDPG code in PyTorch from the Udacity lesson was used as a starting point and adapted for the 20 agent environment, with two deep neural networks (actor-critic), each with two hidden layers of 256-128 nodes (based on recommendations from the DDPG paper), ReLU activation functions on the hidden layers and tanh on the output layer.

3. Hyperparameters

This phase required research to determine how to tweak the hyperparameters optimally for my model. I started with batch_size of 64, but found that increasing it to 128 increased the speed of training. There was no change to the default Ornstein-Uhlenbeck noise parameters (0.15 theta and 0.2 sigma.)

```
BATCH_SIZE = 128       # minibatch size
BUFFER_SIZE = int(1e5)  # replay buffer size
GAMMA = 0.99           # discount factor
LR_ACTOR = 1e-4        # learning rate of the actor
LR_CRITIC = 1e-4       # learning rate of the critic
TAU = 1e-3             # for soft update of target parameters
```

WEIGHT_DECAY = 0 # L2 weight decay

4. Conclusion

I experimented with the hyperparameters and hidden layer sizes, and found that decreasing the size of the hidden layers to 256, 128 (instead of 400, 300 that was recommended in the DDPG paper) led to faster training. I also found that DRL training can be finicky and inconsistent. I did most of my preliminary development and experimentation on the Udacity workspace and ended up with a successful training run that achieved the goal (+30 avg) in 101 episodes. When I transferred that same exact model to my local Windows 10 machine and did another training run, performance was not quite as strong – the goal was achieved in 118 episodes. So it appears that even with a good model, there is an element of random chance involved in each training run.

Further exploration could be done with different agent models. The two that I would consider as top priorities would be the following;

- D4PG (distributed distributional DDPG algorithm) obtains state of the art performance in a variety of control-related task including hard manipulations and locomotion tasks, which could make it very well-suited for this Reacher application.
- TD3 (Twin Delayed Deep Deterministic policy gradient) is another algorithm that performs extremely well in continuous control applications – it increases stability and performance with consideration of function approximation error, by maintaining a pair of critics along with a single actor.

Additionally, Prioritized Experience Replay could be introduced to make learning more efficient and speed it up by a factor of 2. Uniform sampling from a replay buffer as we did in our model is a good default strategy, especially for initial project design. But prioritized sampling, as the name implies, will weigh the samples and ensure that “important” ones are drawn more frequently for training.