12 July 2019 10:59

Continuous Control

Learning Algorithm

In this project, a double-jointed arm agent is trained to 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 should be a number between -1 and 1.

I used 20 identical agents, using multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

I chose to use the Deep Deterministic Policy Gradient algorithm. The DQN is not ideal to use in a continuous action space, and while adapting it for a continuous space would imply discretizing the action space, where the number of actions increases exponentially with the number of degrees of freedom. Such large action spaces are difficult to explore efficiently, and thus successfully training DQN-like networks in this context is likely intractable. Additionally, naive discretization of action spaces needlessly throws away information about the structure of the action domain, which may be essential for solving many problems. Instead I used a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces, called DDPG (based on paper)

In DDPG, there are two networks, one for the actor and one for the critic. The actor is used to approximate the optimal policy deterministically, generating the best belief (argmax Q(s,a)). The critic learns to evaluate optimal action value function by using the actor's best belief. In fact the actor is an approximate maximiser which calculates a new target value for training the action value function.

The original DQN has two copies of the network weights (Regular/Target). The weights of the regular network are copied into the target network every 10000 steps. Within DDPG, there are two copies of the network weights for both the actor and the critic. The target networks are updated using a soft update strategy, by slowly blending the regular network weights with the target network weights. Basically every step there is a mix in 0.01% of regular network weights with the target network weights, which facilitates faster convergence.

Hypermeters:

```
BUFFER_SIZE = int(1e6)   # replay buffer size
BATCH_SIZE = 64          # minibatch size
GAMMA = 0.99             # discount factor
TAU = 1e-3               # for soft update of target parameters
LR_ACTOR = 1e-3          # learning rate of the actor
LR_CRITIC = 1e-4         # learning rate of the critic
WEIGHT_DECAY = 0.0       # L2 weight decay
```

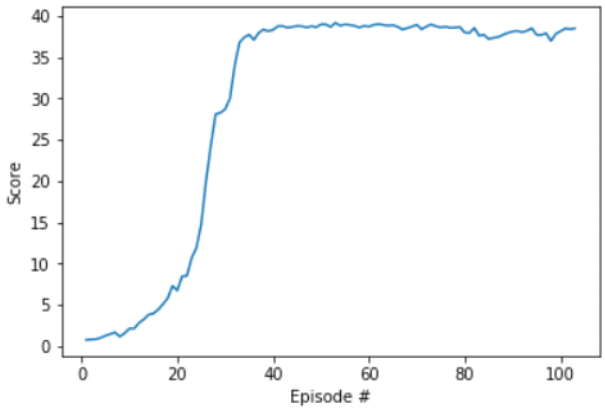
Within the model architecture, the number of units for the fully connected layers made a significant difference, and while I first used fc1_units=400, fc2_units=300, after several trials fc1_units=200, fc2 _units=400 nourished faster convergence.
I also used Leaky ReLU, which has a small slope for negative values, instead of altogether zero, with two main benefits:

* It fixes the "dying ReLU" problem, as it doesn't have zero-slope parts and
* It speeds up training. There is evidence that having the "mean activation" be close to 0 makes training faster. Unlike ReLU, leaky ReLU is more "balanced," and may therefore learn faster.

Plot of Rewards

[version 2] the agent is able to receive an average reward (over 100 episodes, and over all 20 agents) of at least +30.

```
Episode 100     Average Score: 29.14    Score: 38.156
Episode 103     Average Score: 30.27    Score: 38.507
Environment solved in 103 episodes!     Average Score: 30.27
```



Ideas for Future Work

Using PPO (Proximal Policy Optimization) is now the default reinforcement learning algorithm.

PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

There is a new variant which uses a novel objective function by implementing a way to do a Trust Region update which is compatible with Stochastic Gradient Descent, and simplifies the algorithm by removing the KL penalty and the need to make adaptive updates.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\,]$$

* ϑ is the policy parameter
* Et denotes the empirical expectation over timesteps
* rt is the ratio of the probability under the new and old policies, respectively
* At is the estimated advantage at time t
* ε is a hyperparameter, usually 0.1 or 0.2