Continuous Control Project Report

1. Project Overview and Environment Introduction

In this 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 your 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.

For this project, we will provide you with two separate versions of the Unity environment:

The first version contains a single agent. The second version contains 20 identical agents, each with its own copy of the environment. The second version is useful for algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

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

Option 2: Solve the Second Version The barrier for solving the second version of the environment is slightly different, to take into account the presence of many agents. In particular, your 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.

2. Learning Algorithm

Recently significant progress has been made by combining advances in deep learning with reinforcement learning, resulting in the " Deep Q Network" (DQN algorithm). With Deep Q-Learning, a deep neural network is used to approximate the Q-function or action-value function. Given a network F, finding an optimal policy is a matter of finding the best weights w such that F(s,a,w) ≈ Q(s,a). the Deep Q-Learning algorithm represents the optimal action-value function q_* as a neural network (instead of a table).

However, while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action space. For physical control tasks with continuous (real-valued) and high-dimensional action spaces, DQN cannot be directly applied. The number of actions increases exponentially with the degree of freedom. Additionally, simple discretization of action spaces throws away information about the structure of the action domain, which is essential to solving many problems.

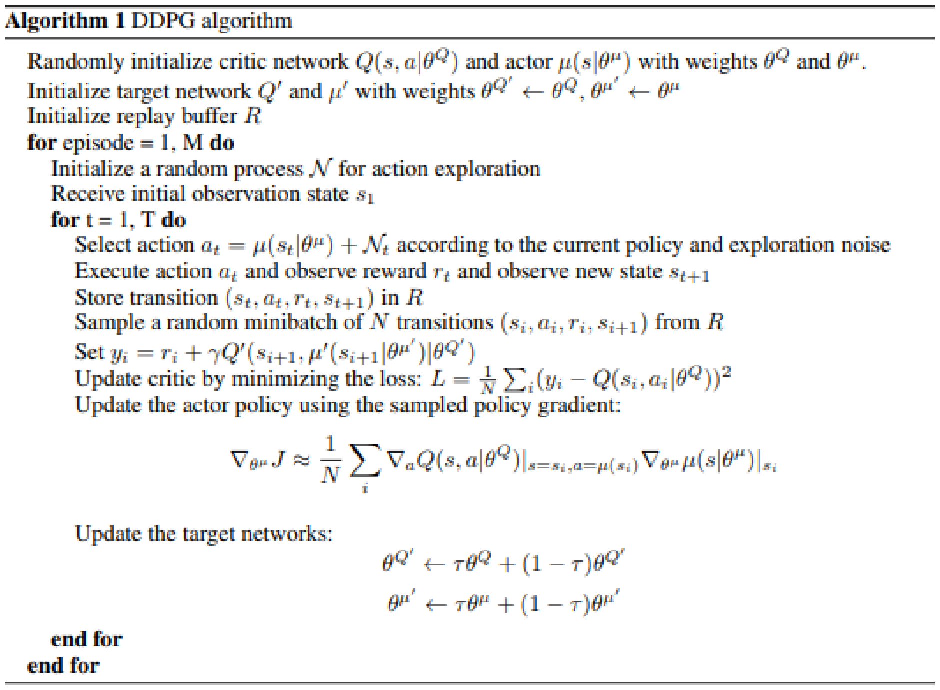
To solve the continuous control problem, a model-free, off-policy actor-critic algorithm using deep function approximations that can learn policies in high-dimensional, continuous action spaces call Deep DPQ (DDPG) is used in this project.

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

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution across a discrete action space 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:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma max\overbrace{Q(s',a')}^{\text{This depends Q function itself (at the moment it is being optimized)}} - Q(s,a)]$$

The oseudo-code of the algorithm looks like this:



Algorithm 1 DDPG algorithm

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

It consists of four major parts:

1.Experience replay: a replay buffer samples experience to update neural network parameters. During each trajectory roll-out, we save all the experience tuples (state, action, reward, next_state) and store them in a finite-sized cache — a “replay buffer.” Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks. In optimization asks, we want the data to be independently distributed. This fails to be the case when we optimize a sequential decision process in an on-policy way, because the data then would not be independent of each other. When we store them in a replay buffer and take random batches for training, we overcome this issue.

2.Actor(Policy) & Critic(Value) network updates: The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation. However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value. For the policy function, our objective is to maximize the expected return. To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter.

3.Target network updates: We make a copy of the target network parameters and have them slowly track those of the learned networks via “soft updates".

4.Exploration and exploitation: In Reinforcement learning for discrete action spaces, exploration is done via probabilistically selecting a random action (such as epsilon-greedy or Boltzmann exploration). For continuous action spaces, exploration is done via adding noise to the action itself (there is also the parameter space noise but we will skip that for now). In the DDPG paper, the authors use Ornstein-Uhlenbeck Process to add noise to the action output (Uhlenbeck & Ornstein, 1930). The Ornstein-Uhlenbeck Process generates noise that is correlated with the previous noise, as to prevent the noise from canceling out or “freezing” the overall dynamics.

3. Training Process of the project

The neural network architecture used for this project can be found here in the model.py file of the source code.

The network architecture is defined using pytorch.

It is observed that an average score of +30 over 244 consecutive episodes.

The initial attempt was use the DDPG code from the Udacity repository. And the training process was slow and the score was below 1 even after a few hundreds of episodes.

```
Episode 100 Average Score: 0.51
Episode 200 Average Score: 0.43
Episode 300 Average Score: 0.48
Episode 400 Average Score: 0.46
Episode 500 Average Score: 0.48
```

Then the neural network architecture is studied to see how to improve the training process. The initial network consists of two layers with ReLu as the activation function. But changing the neuron numbers didn't help much.

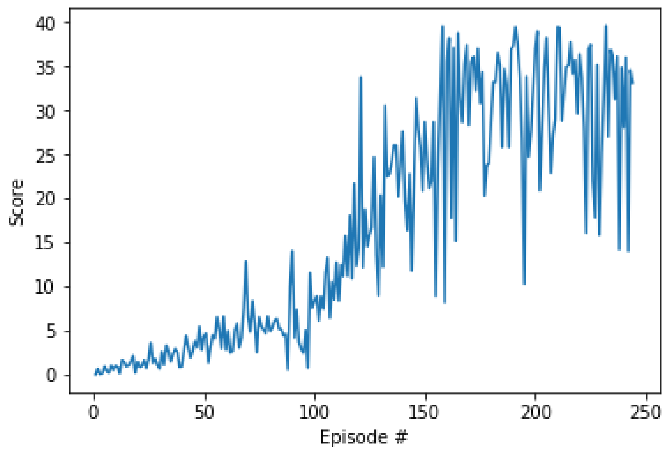
It turned out Batch normalization actually can help. Running computations on large inputs can inhibut learning. Normalization scales the features to be within the same range, typically between 0 and 1. It could be placed in between first layer and second layer. Or it can be done after the second layer or after each layer.

Gradient Clipping can help learning by clipping outsized gradients. By using torch.nn.utils.clip_grad_norm_ function and setting the function to "clip" the norm of the gradients at 1, it places an upper limit on the size of the parameter updates, and prevents them from growing exponentially.

results after adding batch normalization after layers and gradient clipping.

Episode 100 Average Score: 3.55 Episode 200 Average Score: 23.62 Episode 244 Average Score: 30.01 Environment solved in 244 episodes! Average Score: 30.01

Total Training time = 60.2 min



4. Ideas for Future Improvement

1. **further tuning of the model**

It seems some parameters in the nueral network can affect the convergence and performance of the training.

2. **Other algorithms**

We can try algorithm such as Trust Region Policy Optimization (TRPO), or Distributed Distributional Deterministic Policy Gradients (D4PG) to see if they are more robust.

3. **Prioritized Experience Replay**

DDPG samples experience transitions uniformly from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.