Continuous Control with Deep Deterministic Policy Gradient (DDPG) and Soft-Actor-Critic (SAC)

OVERVIEW

The agent implementation follows very closely the approach, network architectures, and hyperparameters found in the original DDPG paper mentioned below. Additionally, I have provided an older (late 2018) version of the Soft-Actor-Critic algorithm. We can see that the Soft Actor Critic is much more efficient in training time, however requires additional work to balance exploration and exploitation. In its pure form, SAC is not well applicable to this problem due to dealing with a high amount of randomness and too much exploration. The continuous control version is working on the 20 "agents" environment, acting in 20 distinct instances of the same environment. The DDPG agent has 1 set of networks (local and target for both actor and critic) that are updated and one common replay buffer of experiences being "shared" among the agents. A distinct action for each environment's state (20 at each timestep) is taken and recorded across the 20 agents in the replay buffer. The SAC algorithm works slightly differently, with its older implementation, using in addition to the actor and critic network also a policy network.

LEARNING ALGORITHM

The DDPG and SAC architecture here is based on the papers titled, *Continuous Control With Deep Reinforcement Learning* and *Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. Architecture and hyperparameters follow the papers closely.

Below are the outlines of both the DDPG and the SAC algorithm.

DDPG

Algorithm 1 DDPG algorithm

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

SAC

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

By plainly comparing these two pseudo-code algorithms we can already see the much less compute intense implementation of the soft actor critic algorithm. In experiments run with the two algorithms it became clear that SAC iterates much faster over time steps and episodes.

NETWORK ACHITECTURE FOR THE DDPG WORKING ALGORITHM

The architectures in this implementation is as follows:

Actor
Input: Linear(num_units = state size) > batch norm
Hidden 1: Linear(num_units = 64) > relu > batch norm
Hidden 2: Linear(num_units = 64) > relu > batch norm
Output: Linear(num_units = action size) > tanh

Critic
Input: Linear(num_units = state size) > batch norm
Hidden 1: Linear(num_units = 64) > relu > batch norm
Concat: Concat(Critic-hidden1-output, Actor-output)
Hidden 2: Linear(num_units = 64 + action_size) > relu > batch norm
Output: Linear(num_units = 1)

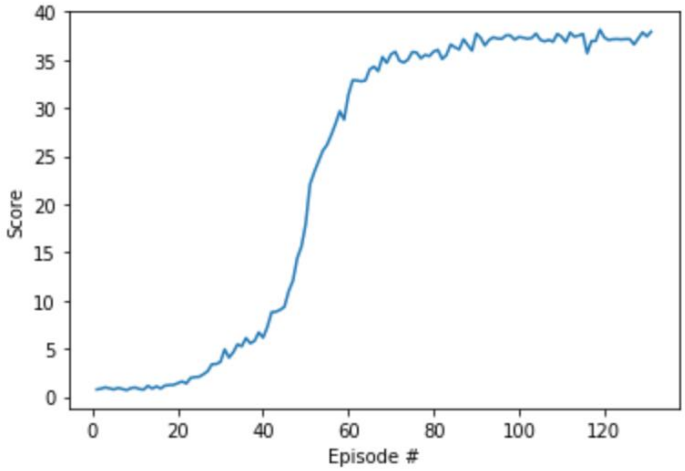
HYPERPARAMETERS

Hyperparameter	Value	Description
minibatch size	64	Number of training examples to sample from memory
replay buffer	1000000	Number of experiences to store in memory
gamma	0.99	Discount factor of gamma used in Q-learning update
update frequency	20	how many timesteps between agent updates
n updates	10	how many network updates to perform per agent update
actor learning rate	1e-4	The learning rate used by Adam
critic learning rate	3e-4	The learning rate used by Adam
tau	1e-3	The parameters used by soft update of target network weights
L2 weight decay	0	weight decay used by Adam

Hyperparameter	Value	Description
max timesteps	1000	max number of timesteps for each episode

PLOT OF REWARDS

While the SAC algorithm requires further tuning regarding the trade-off of exploration vs exploitation, the DDPG algorithm was able to solve the environment over 130 episodes.



IDEAS FOR FUTURE WORK

Additional work should go towards implementing the newest version of the Soft-Actor-Critic algorithm to properly compare its performance with DDPG. Some particular parts to be included are:

1. **Prioritized Experience Replay**: One of the possible improvements already acknowledged in the original research lays in the way experience is used. When treating all samples the same, one is not using the fact that one can learn more from some transitions than from others. Prioritized Experience Replay (PER) is one strategy that tries to leverage this fact by changing the sampling distribution.
2. **Noise Addition**: While the DDPG implementation follows this approach, the current late 2018 implementation of the SAC algorithm does not include the OUNoise process.
3. **Batch Normalization**: In order to make training of the neural networks more efficient, batch normalization should also be added to the SAC neural networks.