Continuous Control environment using Deep Deterministic Policy Gradient (DDPG) agent

Learning Algorithm

The agent/agents are trained using the DDPG algorithm. You can learn more at this link. Also a very good code example can be found in the DDPG-Pendulum Udacity's tutorial.

The DDPG pseudo-code can be seen in the below diagram:

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

- DDPG was introduced as a slight different **actor-critic algorithm**. Here, the actor implements a current policy to deterministically map states to a specific "best" action (the Actor directly maps states to actions instead of outputting the probability distribution across a discrete action space). The critic implements the Q function, and is trained using the same paradigm as in Q-learning, with the next action in the Bellman equation given from the actor's output. The actor is trained by the gradient from maximizing the estimated Q-value from the critic, when the actor's best predicted action is used as input to the critic.
- DDPG also implements a **Replay Buffer**. As used in Deep Q learning (and many other RL algorithms), DDPG also uses a replay buffer to sample 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. 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 because 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.

- **Actor & Critic Network Updates**: The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

- 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:

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

- In order to encourage exploration during training, **Ornstein-Uhlenbeck noise** is added to the actors selected actions. In the DDPG paper, the authors use *Ornstein-Uhlenbeck Process* to add noise to the action output (Uhlenbeck & Ornstein, 1930):

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

- The *Ornstein-Uhlenbeck Process* generates noise that is correlated with the previous noise, as to prevent the noise from cancelling out or "freezing" the overall dynamics. Wikipedia provides a thorough explanation of the *Ornstein-Uhlenbeck Process*.

- Another detail is the use of **soft updates** (parameterized by tau below) to the target networks instead of hard updates as in the original DQN paper.

More info about DDPG can be found to this article

Hyperparameters

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.9             # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-3         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # weight decay
```

- BUFFER_SIZE = a buffer should be big for maximum experience for training.
- BATCH_SIZE = big batch size for more generic update during training.
- GAMMA = discount factor for Q values.
- TAU = controlling how much two neural networks should have similar weights.
- LR_ACTOR = learning rate of 1e-3 is one of the optimal learning rates for Adam optimizer.
- LR_CRITIC = learning rate of 1e-3 is one of the optimal learning rates for Adam optimizer.
- WEIGHT_DECAY = No weight decay is performing better in the DDPG implementation.

Clearly algorithm was giving better results with big buffer size and batch size more than 128 (here used 256). The parameter of the noise showed that smaller was better and changed from 0.5 of Pendulum example to 0.02 here.

Model architecture

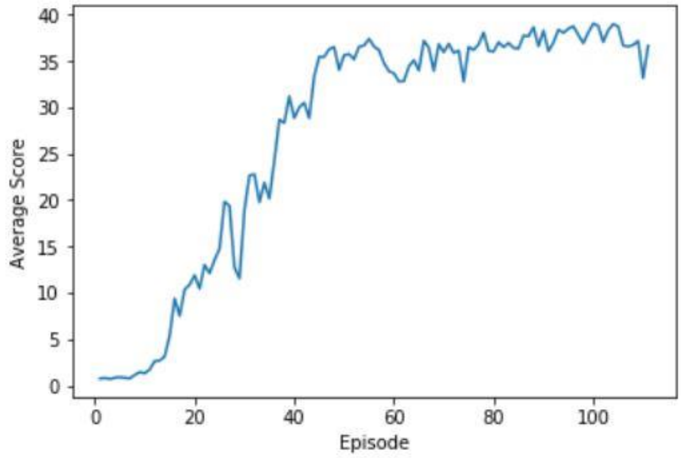
Actor Neural Network: Here 3 fully connected layers were used with every one smaller than another. Starts with 512 then 256 and finally 128 nodes. At every layer Batch normalization is used. More info about batch_norm layers at this link. Relu activation functions were used after each layer and finally tahn function so the values of actions will be between -1 and 1.

Critic Neural Network: Here 2 fully connected layers were used with 512 and 256 nodes. Batch normalization and relu activation functions were used as in actor NN. Also we used a torch.cat method (details can be found here)

During training we used gradient clipping which made a huge difference in the stability of the reward increase.

Plot of Rewards

```
In [13]: plot_list(all_episode_scores)
```



The number of episodes needed to solve the environment were 111. GPU was used to decrease time

```
Episode 109     Average Score: 29.56     Time per episode: 25.88
Episode 110     Average Score: 29.88     Time per episode: 25.78
Episode 111     Average Score: 30.23     Time per episode: 26.10

 Environment solved in 111 episodes!     Average Score: 30.23     Total time: 2576.29
```

Ideas for future

- This DDPG implementation was very dependent on hyperparameter, neural network number of nodes, noise settings and random seed. Solving the environment using [PPO](), [TRPO]() or [D4PG]() might allow a better solution to this task.
- Add ***prioritized*** experience replay: Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.