Deep Deterministic Policy Gradients (DDPG)

Collaboration & Competition Project

Algorithm Pseudocode - Source: https://arxiv.org/pdf/1509.02971.pdf

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

What DDPG is about:

DDPG can be thought as the extension of the Q-Learning algorithm for continuous tasks where discrete state/actions may not suffice.

The actor will approximate a deterministic policy, in contrast to other actor networks we may come across in actor-critic methods, thus the actor will always map the “best action” whenever we query the network.

The critic is used to approximate the maximizer over the Q values of the next state (SarsaMax – Q-Learning) and not as a learned baseline, as with other actor-critic algorithms.

DDPG implements a replay buffer, much like in DQNs where the SARS experiences are saved.

About the Project

This project uses the **Deep Deterministic Policy Gradients (DDPG)** algorithm to utilize an actor-critic approach to solve a UnityML Agents environment that utilizes two competitive agents playing tennis.

Project Files Involved

There are 3 total files involved for this project

- **model.py** contains the models for the actor and the critic neural networks
- **ddpg_agent.py** contains the code defining the Actor class, accompanied by the ReplayBuffer and the OUNoise classes.
- **Tennis.ipynb** contains the notebook with the solution for the 1 Agent Environment approach

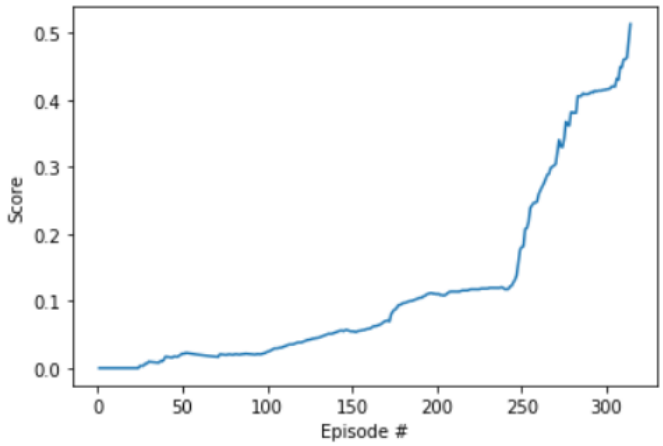
This project uses as a codebase the exact same approach as the Continuous Control project used to solve another UnityML Agents environment which can be found also at one of my GitHub Repositories at

https://github.com/ioannisa/DDQN-Continuous-Control

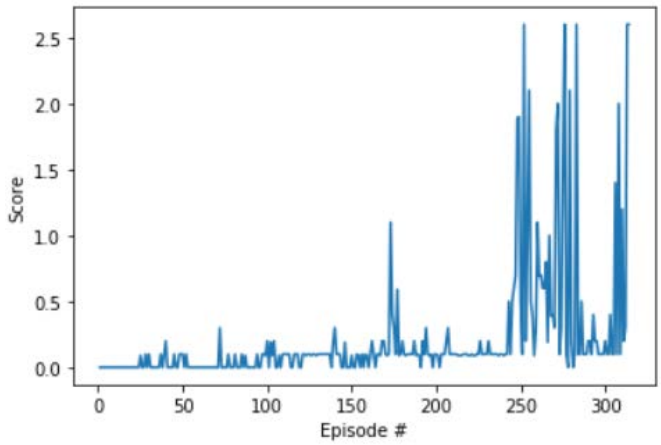
Approaching the project

The problem was solved using only 214 episodes (314 in order to achieve 0.5 in consecutive 100 episodes).

Agents average score over 100 episodes



Agents average score per episode



The Architecture

There have been two neural networks for the utilization of the DDPG algorithm.

I defined the **actor network** using three internal fully connected layers of dept 512, 256 depths respectively, with batch normalization to their inputs.

The **critic network** utilized two internal layers of 512 and 512 layers, with batch normalization to the inputs as well.

Another very important factor was the **noise** at the “OUNoise” function. Starting with a noise of 0.1, the system was too unstable. Started to gradually reduce the noise, where at value 0.01 it seemed that the network behaved very efficiently.

Gradient Clipping appeared to be much beneficial to the critic network, as shown in the lines bellow

```
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

Hyper Params

```
BUFFER_SIZE = int(2e6)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 2e-3              # for soft update of target parameters
LR_ACTOR = 2e-4         # learning rate of the actor
LR_CRITIC = 2e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
```

What's next

It seemed that even slight changes to the network were changing the behavior greatly.

The entire structure is so fragile, that even the random seed seemed to be deterministic to the entire network performance. The best value that worked for me was changing random seed to 19, but playing with more values could possibly help produce even better results.

Further finetuning the hyperparams may provide some even more refined results. Changing the sigma coefficient to 0.01 helped the neural network at later episodes, and maybe an even lower value could help it even further. In possible future implementation I would decay the noise, starting from a value of around 0.05 all the way down to 0.01.

More algorithms for the same environment, like PPO, A3C, D4PG could possibly converge faster.