Project2: Continous control - DDPG for Reacher-v2 environment

UDACITY

Contents

1 Environment description 2

2 DDPG algorithm description 3

3 Implementation details 5

4 Training and Result 6

5 Summary and ideas for future work 6

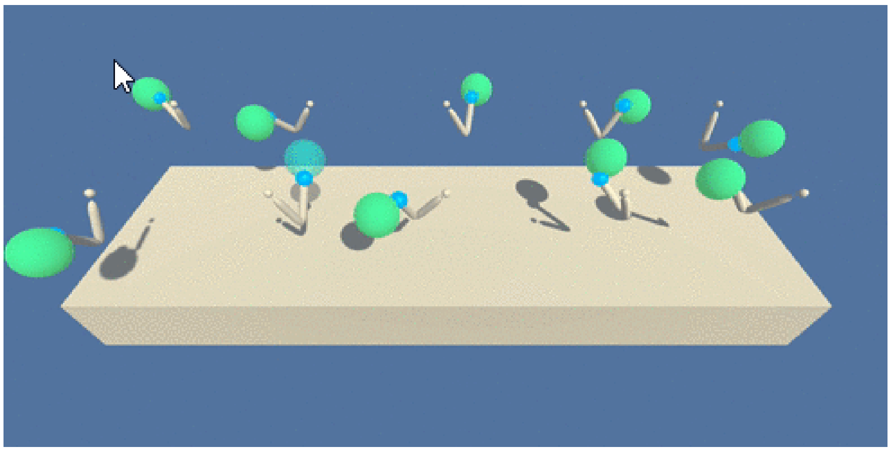


Figure 1: Unity ML-Agents Reacher Environment

1 Environment description

The environment is a Unity environment called Reacher-v2. The goal is to train an agent that can move a double-jointed robot arm to target locations. A screenshot of the environment is shown in figure 1.
To increase training speed an enviroment with multiple identical arms, that are actuated in paralell, is used. This way learning can be speed up.

Statespace and dynamics

The observation space of one arm has 33 dimensions, $\mathcal{S} \subset \mathbb{R}^{33}$ and contains the the angle between the joints, the velocity and angular velocity of the rigid bodies as well as the distance of the finger tip from the target location.

Actionspace

The action space of the agent is continous and consists of 4 inputs, which are two torque values per joint:

$$\mathcal{A} = \{\tau_{x,1}, \tau_{y,1}, \tau_{x,2}, \tau_{y,2}\}, \tag{1}$$

whereby each torque is normalized to be within $\tau_i \in [-1, 1]$.

Reward and solution criteria

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. 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. The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

2 DDPG algorithm description

The problem with DQN on continous actions-spaces is the maximum operation $\max_a Q(s,a)$, which is needed for calculation of the Q-targets and ϵ-greedy policy $\pi(a|s)$. To circumvent this problem deep deterministic policy gradient algorithm concurrently learns a action-value-function and a policy [Lil+19]. The goal of the policy network μ_ϕ is just to maximize the action-value network $Q_\theta(s,a)$. If the deterministic policy is written as

$$\mu(s) = \mu_\phi(s), \tag{2}$$

the loss function for the network training would be

$$L(\phi) = -E_{s\sim\mathcal{D}}[Q_\theta(s, \mu_\phi(s))], \tag{3}$$

whereby $\mathcal{D}$ is the replay buffer containing tuples of the form $(s_t, a_t, s_t^{`}, r)$. One can see, that to perform gradient descent on this loss function, also the gradient of the action-value function is needed. This gradient can be obtained by backpropagating through the Q-value network. The loss function for the action-value network is analogous to the DQN algorithm, but instead of the max operation the policy network is used:

$$L(\theta, \mathcal{D}) = \sum_i \left(y_i - Q_\theta(s_i, \mu_\phi(s))\right)^2, \tag{4}$$

$$y_i = r_i + Q_\theta(s_i', \mu_\phi(s)) \tag{5}$$

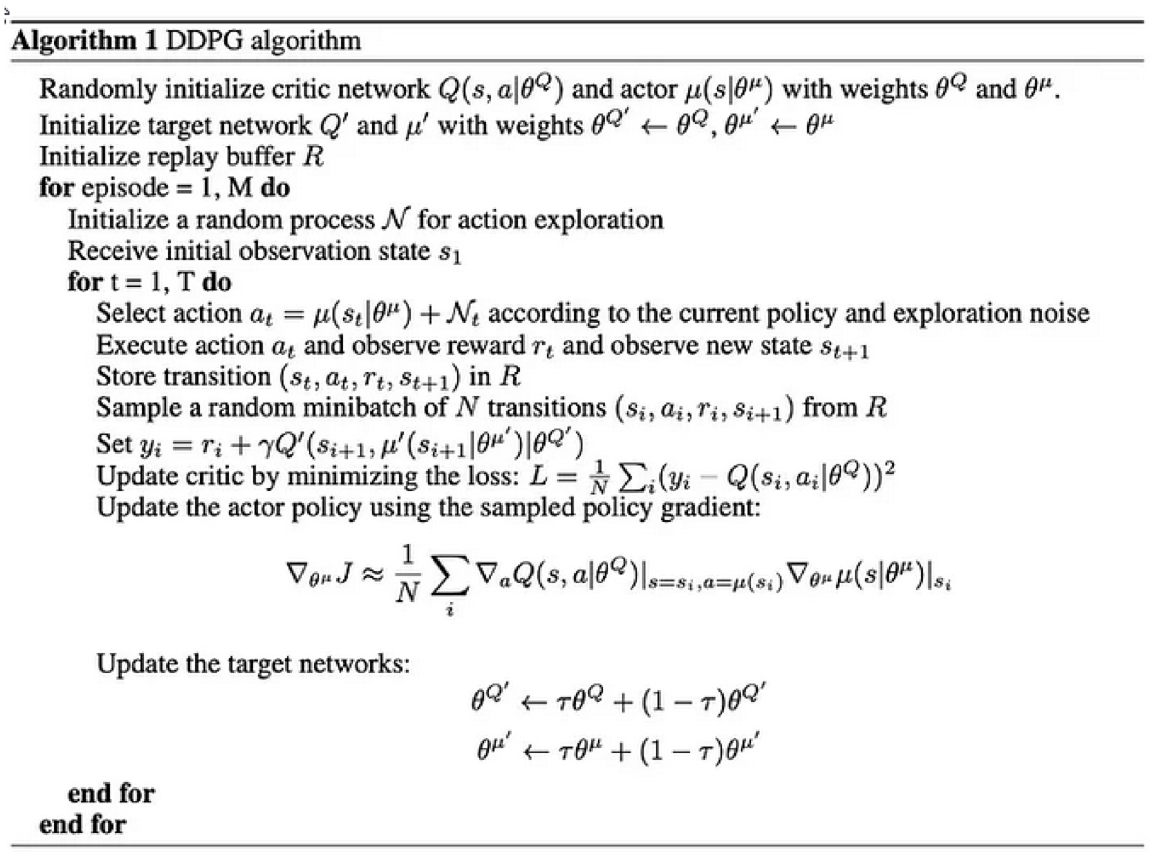
To provide exploration a noise term is added to the behaviour policy

$$a_t = \mu_\phi(s_t) + \mathcal{N}_t. \tag{6}$$

To generate the noise $\mathcal{N}_t$, an Ornstein–Uhlenbeck (OU) noise process was used. It has been suspected that this time correlated nosie provides better performane during training. Recent results however suggest that, normal gaussian noise is also sufficient [Opea]. The following tricks were introduced to make the training of the algorithm more stable.

- Target Q-network (see project1 for details)
- Replay buffer (see project1 for details)
- Gradient clippling was suggested in the project instructions. Therefore it was implemented for the critic network.

The full aglorithm is shown in figure 4.



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
```

Figure 2: Summary of DQN algorithm

3 Implementation details

As a baseline implementation the code from the Udacity ”Pendulum” coding exercise was used. Some adjustments were made to cope with the banana environment. The action-value function approximation or critic network architecture is shown in figure 4. The policy network architecutre is shown in figure 3.

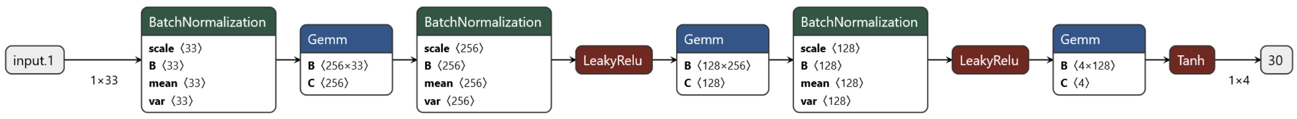


Figure 3: Architecture of the actor network $\mu_\theta(s)$. Visualized with: netron.app.

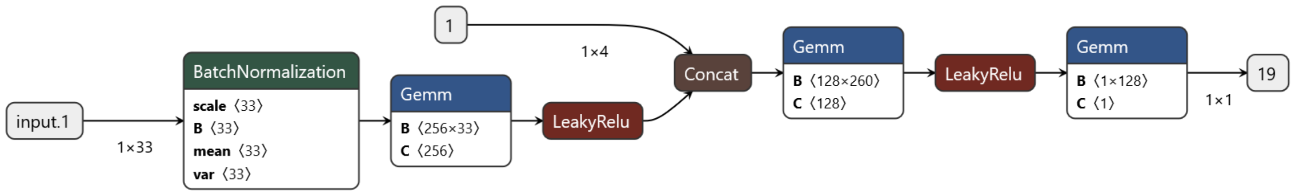


Figure 4: Architecture of the critic network $Q(s, a)$. Visualized with: netron.app.

The following hyper-parameters were used for training:

```
BUFFER_SIZE = int(4e5)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 2e-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        # L2 weight decay
UPDATE_STEPS = 8
NUM_BUFFER_SAMPLES = 16
MU_NOISE      = 0       # mean of OU-noise
THETA_NOISE = 0.15 # parameter of OU-noise
SIGMA_NOISE = 0.2  # standard deviation of OU-noise
```

4 Training and Result

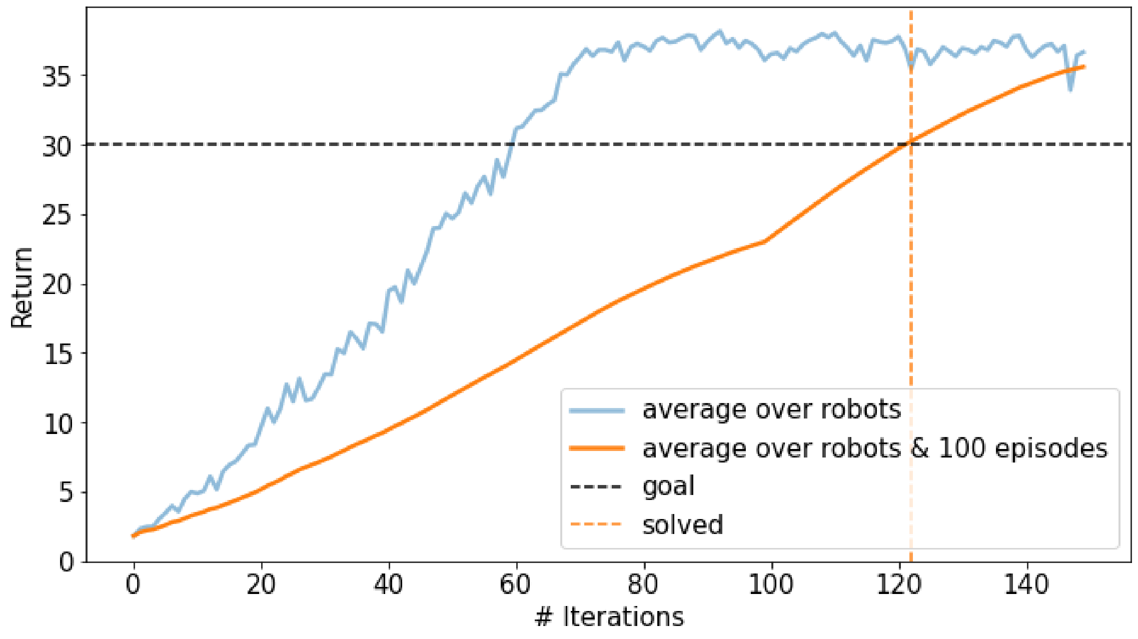


Figure 5: Progress of training

5 Summary and ideas for future work

The training of an agent was pretty straight forward using the DDPG baseline algorithm provided in the coding exercise. Just some slight adjustment to the network and the hyperparameters were needed. Care had to be taken with the correct shape of the noise. Initially one dimensional noise was generated and the learning failed.
To improve performance, it would be interesting to further tune the hyperparameters or compare the performance to other algorithms like Proximal Policy Optimization (PPO), A3C or the improved version of DDPG which is called Twin Delayed DDPG (TD3) [FvHM18]. TD3, incorporates the following tricks into DDPG [Opeb]:

- TD3 learns two Q-functions instead of one and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- TD3 updates the policy (and target networks) less frequently than the Q-function.
- TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

References

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. Oct. 22, 2018. DOI: 10.48550/arXiv.1802.09477. arXiv: 1802.09477 [cs, stat]. URL: http://arxiv.org/abs/1802.09477 (visited on 12/08/2023). preprint.

[Lil+19] Timothy P. Lillicrap et al. *Continuous Control with Deep Reinforcement Learning*. Version 6. July 5, 2019. DOI: 10.48550/arXiv.1509.02971. arXiv: 1509.02971 [cs, stat]. URL: http://arxiv.org/abs/1509.02971 (visited on 12/08/2023). preprint.

[Opea] OpenAI. *Deep Deterministic Policy Gradient — Spinning Up Documentation*. URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html (visited on 12/08/2023).

[Opeb] OpenAI. *Twin Delayed DDPG — Spinning Up Documentation*. URL: https://spinningup.openai.com/en/latest/algorithms/td3.html (visited on 12/08/2023).