The Reacher Unity Environment with Deep Deterministic Policy Gradient (DDPG)

Paolo Recchia

1 The DDPG algorithm

Deep–Q–network (DQN) successfully trained an agent in order to master in different Atari games [1]. In this context the off–policy ϵ–greedy action selection is simpler than a more general action selection technique. In fact, with a discrete action space, the $\max_a$ operation in the loss function

$$\mathcal{L}(\theta_Q) = \left[R + \gamma \max_{a'} Q(s', a', \theta_Q) - Q(s, a, \theta_Q) \right]^2. \tag{1}$$

is more straightforward. If the action space is continuous the same operation is much harder instead, and in some special cases it can be computationally intractable.

Thanks to another network (often called "actor" network) the Deep Deterministic Policy Gradient algorithm (DDPG) [2] makes the $\max_a$ operation easier. In order to maximize the action–value function $Q(s, \mu(s))$, the actor network train a deterministic policy (which gives a deterministic action) $a = \mu(s|\theta_\mu)$ with gradient ascent,

$$\theta_\mu \leftarrow \theta_\mu + \alpha \nabla_{\theta_\mu} Q(s, \mu(s|\theta_\mu)|\theta_Q). \tag{2}$$

One of the advantage of DDPG is that the gradient in (2) can be calculated exactly [3]. The output of the actor network is then used to train the deep–Q–network $Q(s, \mu(s)|\theta_Q)$ (in this context called the "critic" network) as in DQN. The same keys optimization of DQN are implemented in DDPG too, namely the use of a replay buffer and a separate fixed target network. However it worth to underline some important differences

Target Networks As in DQN it's better to introduce different target networks for both the actor and the critic. In fact, as we can see in (1) and (2), to make a reliable learning process, the target parameters of the actor $\mu(s|\theta_\mu)$ and the critic $Q(s, \mu(s)|\theta_Q)$ has to be decoupled. We indicate these target networks with $\mu'(s|\theta'_\mu)$ and $Q'(s, \mu(s)|\theta'_Q)$ respectively.

"Soft" Target Update In DQN algorithm, at every C step, the weights of actor and critic networks get directly copied in their respective target ones. In DDPG we softly update them with

$$\theta' \leftarrow \tau\theta + (1 - \theta)\theta' \tag{3}$$

where τ is a small positive number $\tau \ll 1$. In addition, to make the learning process more robust, we softly update the networks M times each C steps, at each time we sample randomly a batch of size D from a memory buffer B of fixed size N

Exploration with noise DDPG is an off–policy algorithm which means that the policy, interacting with the environment, is different from the policy learned. In this case we can use the off–policy to promote exploration of the action space by adding some noise

$$\mu'(s) = \mu(s|\theta_\mu) + \mathcal{N}, \tag{4}$$

in this context we chose a noise $\mathcal{N}$ generated by the Ornstein–Uhlenbeck process [4].

With the aforementioned keys optimization the loss function (1) reads as

$$\mathcal{L}(\theta_Q) = \mathbb{E}_{(s,a,r,s')\sim U(B)}\Big[r + \gamma Q'(s', \mu'(s'|\theta'_\mu)|\theta'_Q) - Q(s,a|\theta_Q)\Big]^2 \tag{5}$$

where the symbol $\mathbb{E}_{(s,a,r,s')\sim U(B)}$ indicates the expected value calculated on a batch of experiences (s,a,r,s') uniformly sampled from the buffer memory B.

As in [2] the algorithm to train the agent in Reacher Unity Environment reads as

Algorithm 1: DDPG [2]

Initialize buffer memory B to capacity N
Initialize the critic $Q(s, a|\theta_Q)$ and actor $\mu(s|\theta_\mu)$ networks with random weights θ_Q and θ_μ
Initialize target networks Q' and μ' with weights $\theta'_Q = \theta_Q$ and $\theta'_\mu = \theta_\mu$
for episode $i = 1$ **to** Env Solved **do**
  Reset Reacher Unity Environment and set the input state of the first step s_1
  Initialize the random noise $\mathcal{N}$
  for time–step $t = 1$ **to** T **do**
    Select action by actor network $a_t = \mu(s_t|\theta_\mu) + \mathcal{N}_t$
    The agent executes action a_t and record r_t and next state s_{t+1}
    Store the experience (s_t, a_t, r_t, s_{t+1}) in the buffer memory B
    if Every C steps **then**
      for # of update j **to** M **do**
        Sample random mini–batch of size D experiences from B
        Update the critic network parameters θ_Q by minimizing the loss function $\mathcal{L}(\theta_Q)$ in (5)
        Update the actor network parameters θ_μ by maximizing $Q(s, (\mu(s|\theta_\mu)|\theta_Q)$ with gradient ascent
        Perform the soft update

$$\theta'_Q \leftarrow \tau\theta_Q + (1-\theta)\theta'_Q$$
$$\theta'_\mu \leftarrow \tau\theta_\mu + (1-\theta)\theta'_\mu$$

      end
    end
    Set $s_t = s_{t+1}$
    If s_{t+1} is a terminal state go to the next episode
  end
end

2 The Reacher Unity Environments

The aim of the Reacher Unity game is to collect as many objects as possible with a double jointed arm.

The state space has 33 dimensions and contains the position, rotation, velocity, and angular velocities of the arms. 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.

Two versions of the Unity Reacher Environment are availble

- the first one contains just one single agent
- the second one contains 20 different agents

In this project we use the second version.

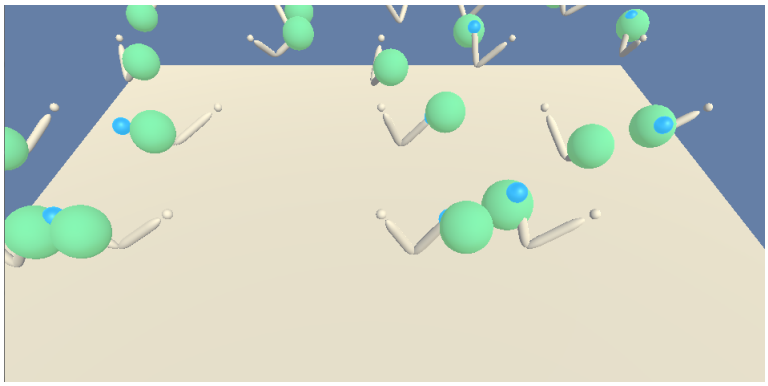

Figure 1: Reacher Unity game

In this way we can train the 20 agents in parallel and sharing the experiences collected in the memory buffer B. 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. We consider the environment solved if the agents get an average score of $+30$ (over 100 consecutive episodes, and over all agents). In particular 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).

3 The hyperparameters of the algorithm and the outcome

3.1 The hyper parameters for both networks

Discount factor $\gamma = 1$

Buffer memory size $N = 1 \times 10^5$

Batch size for random sampling $D = 128$

Step for updating the target networks $C = 20$

How many updates M **each** C **step** $= 10$

Parameter of soft update $\tau = 1 \times 10^{-3}$

Kind of optimizer: Adam (adaptive moment estimation)

Learning rate of Adam optimizer $= 1 \times 10^{-4}$

Input layer $= 33$ (the state space size)

3.2 The actor network

The actor network is a fully forward connected neural network with

first hidden layer = 400 nodes with ReLU activation function

second hidden layer = 300 nodes with ReLU activation function

output layer = 4 (the number of action) with tanh activation function (since each actions has to be a value in a range $(-1, +1)$

3.3 The critic network

The critic network has a slightly different architecture

first hidden fully connected layer = 400 nodes with ReLU activation function

first hidden layer (not connected to the input layer) = 4 nodes (they take as input the action given by the actor network)

second hidden fully connected layer = 300 nodes with ReLU activation function

output layer = 1 the action value function $Q(s, a)$

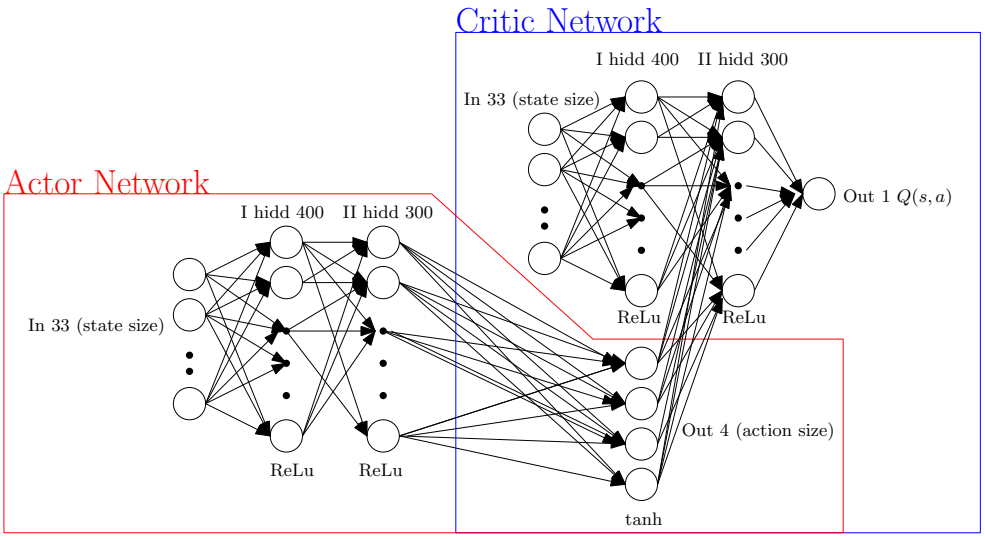

Figure 2: The Actor–Critic network

3.4 The outcome

As we can see in figure 3, by the aforementioned hyper–parameters, the agent solved the game at episode 9, which means that the average score from episode 10 to 109 (included) of the average score of the 20 agents was greater than +30.

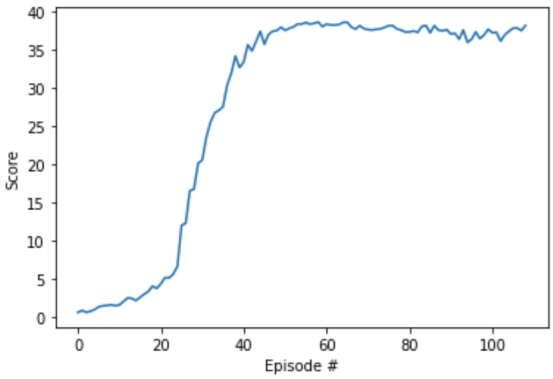

Figure 3: In this test the agent solved the environment at episode 9 with an average score of 30.006.

4 Insights for future work

4.1 Actor–Critic methods

As in DDPG a basic Actor–Critic method uses two networks: one for the actor and one for the critic. However, differently from DDPG the actor network is used to learn the probabilistic policy $\pi((a|s)|\theta_\pi)$ instead of a deterministic action $\mu(s|\theta_\mu)$, and the critic network is used to learn the value function $V(s|\theta_v)$ instead of the action–value function $Q(s, a|\theta_Q)$. The algorithm works like

1. The actor interact with the environment and collect the current state s, the action a, the reward r and the next state s' and eventually collect (s, a, r, s') in a buffer memory

2. With the tuple (s, a, r, s') we use the TD–estimate

$$r + \gamma V(s'|\theta_v) \tag{6}$$

3. We calculate the advantage function with critic value function

$$A(s, a) = r + \gamma V(s'|\theta_v) - V(s|\theta_v) \tag{7}$$

4. Finally we update the actor network to improve the policy approximation with the advantage function as a baseline

Different kind of Actor–Critic methods exist. The most popular are the following

Asynchronous Advantage Actor–Critic (A3C) First of all instead of using a TD–estimate as in (6) we can use an n–step bootstrapping. Basically it uses n–step return to learn the value function

$$r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} V_{t+n-1}(s_{t+n}|\theta_v). \tag{8}$$

In addition, A3C doesn't use the replay buffer memory, it uses a parallel training by creating multiple instances of the environment and agents. These agents, running in parallel, provide experiences already decorrelated, and so we don't need a replay buffer anymore.

Advantage Actor–Critic (A2C): It exists a synchronous version of the A3C. In A3C all the agents update the parameters of the policy network independently to each others. In this case the weights an agent is using can be different at a given time. In A2C, before updating the network, it waits all the agents to finish, then synchronizes the weights and update the policy network at once.

4.2 True policy gradient methods

Differently from Actor–Critic methods, in true policy gradient method there is only one neural network made to learn the best policy $\pi_\theta(a|s)$ directly. As any RL algorithm, the aim is to maximize the expected discounted return, which for both episodic and continuous task can be written as

$$U(\theta) = \sum_\tau P(\tau|\theta)R(\tau) \tag{9}$$

where $\tau = (s_0, a_0, \dots, s_N, a_N, s_{N+1})$ indicates a trajectory and $R(\tau) = \sum_k r_k$ is the related return. In order to update the policy parameters, we need to maximize (9) with gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla U(\theta). \tag{10}$$

The gradient can be calculated exactly

$$\nabla U(\theta) = \sum_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) \tag{11}$$

but usually it is approximated by adding up a batch m of possible trajectories

$$\nabla U(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{N} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) R(\tau^{(i)}). \tag{12}$$

The general algorithm is called REINFORCE and it works like this

1. We use the policy $\pi_\theta(s|a)$ to collect a batch of m possible trajectories $\tau^{(i)} = (s_0^{(i)}, a_0^{(i)}, \dots, s_N^{(i)}, a_N^{(i)}, s_{N+1}^{(i)})$
2. We use this batch to calculate the gradient (12)
3. We update the policy network parameters as (10)
4. Loop over 1–3 until convergence

An important improvement to this general REINFORCE algorithm is called Proximal Policy Gradient (PPO) [5]. In PPO two network are used (as in DQN), the local π_θ and the target $\pi_{\theta'}$. In addition the target network is updated by maximizing a different function. In fact, thanks to some key approximations and further adjustment, we can update the policy by maximizing a function called clipped surrogate function

$$\mathcal{L}_{surr}^{clip}(\theta, \theta') = \sum_t \min \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t^{future}, clip_\epsilon \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right] \tag{13}$$

where $R_t^{future} = \sum_{k=t+1} r_k$ and $clip_\epsilon$ clip its argument in a given range $[1-\epsilon, 1+\epsilon]$. These modifications proved a better performance and outcome in different task, rather than REINFORCE.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.

[2] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[3] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In International conference on machine learning, pages 387–395. PMLR, 2014.

[4] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. Physical review, 36(5):823, 1930.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.