SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'informazione, Informatica e Statistica

Probabilistic Reasoning and Learning

Deep Deterministic Policy Gradient

Professor
Roberto Capobianco

Student
Giuseppe L'Erario

Academic Year 2018/2019

Contents

1 Introduction

Reinforcement learning is a group of algorithms that take as inspiration evolution and learning in the natural world.

The goal of a typical RL algorithm is to build an agent that learns autonomously how to act in a specific environment on the base of the reward it receives during the process.

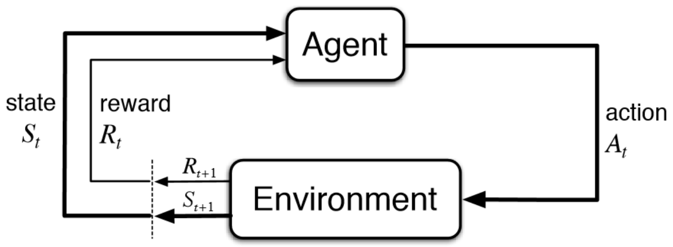


Figure 1: A typical Reinforcement Learning setup.

Each timestep t the environment "shows" the actual state to the agent, that takes an action based on its knowledge. This action affects the environment, that returns a new observation and a reward. The reward is then used by the agent as a feedback to update its knowledge.

A great step forward in the field has been made by *DeepMind* in the work *"Continuous control with deep reinforcement learning"* [1]. The core of this paper is one of the most famous RL algorithms: the **Deep Deterministic Policy Gradient** (DDPG), which will be the subject of this report.

2 The state of the art up to DDPG algorithm

As previously mentioned, the goal of the Reinforcement Learning is to develop an agent able to improve its performance over time.

The behavior of the agent is described by a function $\pi : S \rightarrow P(\mathcal{A})$ that maps the states to a probability distribution over the states.

An important concept is the *discounted future reward* $R_t = \sum_{t=i}^{T} \gamma \cdot r(s_i, a_i) = r_t + R_{t+1}$. This quantity represents the reward the agent can gain in the actual state at timestep t plus the the future rewards that can be obtained acting accordingly to the policy π. $\gamma \in [0, 1]$ is the *discount factor* that weights the importance of the future rewards.

The Reinforcement Learning allows agents to automatically learn a policy π that maximizes the discounted future reward R.

Other famous algorithm like *Q-Learning* or *SARSA*[1] leverage on the concept of the *Q-Value*. It is similar to the discounted future reward R but takes as input also the action a, describing the future reward $Q(s, a)$ of the current state s taking the action a.

These algorithms work on finite environment. Indeed, Q-learning cannot estimate value for unseen states, restricting its usefulness only to small grid worlds. What if we had possible infinite spaces?

The solution to this problem was found by *DeepMind* ([2] and [3]): the **Deep Q Network** algorithm is capable of overcoming even the humans on many Atari games, a truly breakthrough!

DQN leverage on a neural network architecture to estimate the Q-value function. In particular, the neural network takes as input an image representing the actual state and returns as output the Q-value for each action the agent can take.

Prior the DQN, learning using neural networks was difficult and unstable. DQN introduced two innovations that renders stable and robust the learning:

- the experience replay;
- the target network.

DQN is a valid approach for high-dimensional problems. However, it can face only problems in which only a finite set of discrete actions are available.

For continuous action spaces one cannot simply discretize the domain: this will lead to the course of dimensionality problem, since thousands of possible actions could originate from a single state.

Again, *DeepMind* solved this problem by using a neural network architecture to support the **Deterministic Policy Gradient** algorithm and leveraging on the innovations of **DQN**.

3 Deep Deterministic Policy Gradient

Before describing the DDPG algorithm is useful to recall the concepts of **Policy Gradient** and **Q-function**.

[1]Respectively, Q-Learning is an Off-policy algorithm and SARSA an ON-policy one. This means that the first one learns the Q-function based on an action that comes from another policy (actually, a greedy policy), while the second learns the function based on the actual policy of the agent (leading to longer and safer paths w.r.t. Q-learning).

While Action-Values methods learn the value of an action, Policy-Gradient methods learn the policy $\pi(a|s,\theta^\pi)$, that maps the state to the action, stochastic or deterministic[2].

More specifically, they optimize the *policy function parameters* θ^π in order to maximize a score function $J(\theta^\pi)$, clearly related to the reward. This is done in two steps:

- measure the quality the policy π using the score function $J(\theta^\pi)$
- maximize the performance climbing the score function, and hence updating the parameters through gradient ascend:

$$\theta^\pi \leftarrow \theta^\pi + \alpha \nabla_{\theta^\pi} J(\theta^\pi) \tag{1}$$

The Q-function is computed using the recursive Bellman equation:

$$Q(s_t,a_t) = \mathbb{E}(r(s_t,a_t) + \gamma Q(s_{t+1},a_{t+1})) \tag{2}$$

Deep Deterministic Policy Gradient relies on an *actor-critic* architecture, meaning that learns approximations of both policy and value functions. More specifically the approximating functions are two neural networks.

The actor learns and uses a *deterministic* policy $\mu(s,\theta^\mu)$, producing an action based on the current state.

The critic $Q(s,a,\theta^Q)$ learns how to judge a decision taken by the actor $\mu(s,\theta^\mu)$ in a given state, relying on the Bellman equation:

$$y_t = r(s_t,a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta^Q)) \tag{3}$$

and minimizing the loss:

$$L(\theta^Q) = (Q(s_t,a_t|\theta^Q) - y_t)^2 \tag{4}$$

and then updating the parameters θ^Q in order to approximate the Q-value function.

To update the actor one need to know the expression of the gradient of the score function and then perform the update step (1).

Silver et al. [4] provided the expression of the *policy gradient*:

$$\nabla_{\theta^\mu} J = \mathbb{E}[\nabla_{\mu(s_t)} Q(s_t,\mu(s_t)|\theta^Q) \nabla_{\theta^\mu} \mu(s_t|\theta^\mu)] \tag{5}$$

[2]A stochastic policy tells us that a decision is *likely* to be made. This is useful for exploration of new solutions. A deterministic policy maps *exactly* a state to a precise action.

that also shows why the policy is deterministic: $\nabla_{\theta^\mu}\mu(s_t|\theta^\mu)$ cannot be a probability distribution. Anyway, to enhance exploration, Ornstein-Uhlenbeck noise is added to the process during the training.

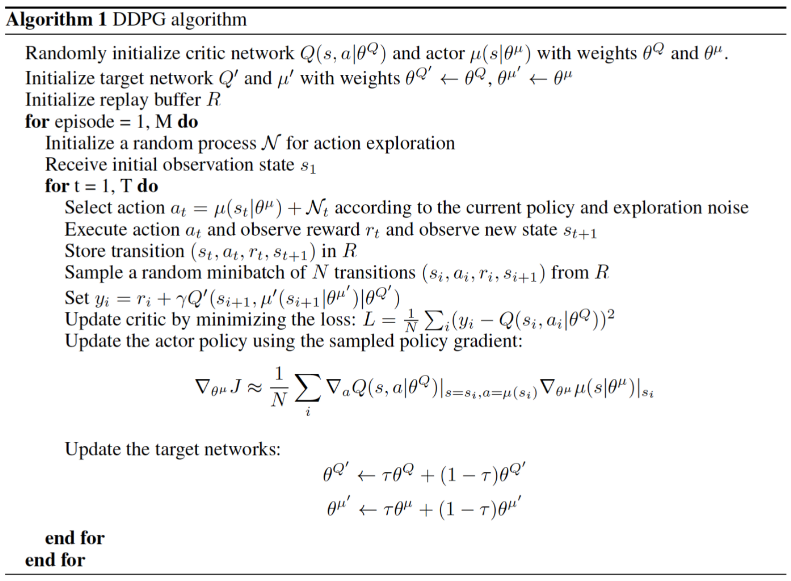
In practice this is not enough. Neural networks learn correlation between sequential states, despite the problem is structured as a Markov process. This leads to instability. The solution is the same found by the authors of DQN: the **experience replay**. The algorithm collects experiences in form of tuples $< s_t, a_t, r_t, s_{t+1} >$ and stores them in a buffer. Then it feeds the networks with a batch of randomly sampled experiences from the replay buffer.

Furthermore, directly update actor and critic parameters using the gradient leads to divergence. The solution in this case is to create two additional networks, called *target* actor and critic networks, $\mu'(s|\theta^{\mu'})$ and $Q'(s,a|\theta^{Q'})$, used to produce the TD error.

The weights of these target networks are updated slowly using the learned networks weights:

$$\theta' = \tau\theta(1-\tau)\theta' \tag{6}$$

with $\tau \ll 1$. The complete algorithm is shown in fig 2.



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: The complete DDPG algorithm.

4 Results

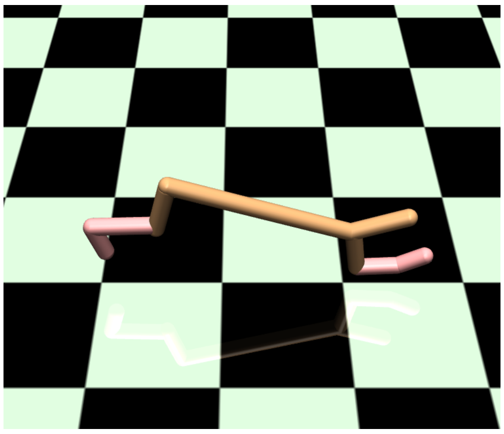


Figure 3: The cute half Cheetah.

The implementation is tested on `HalfCheetah-v2` by MuJoCo using Gym, a toolkit for fast algorithm prototyping developed by OpenAi.

Gym provides the agent with observations and collected reward, relieving the user of the design of the environment.

The `HalfCheetah-v2` environment has an observation space dimension equal to 17 and an actions space dimension equal to 6. The goal is to let the Cheetah run as much as possible.

The chosen parameters for the two networks are:

- Actor Network: three layers with 400 and 300 neurons in the middle. The input dimension is equal to the dimension of the observation space, the output equal to the action space dimension;
- Critic Network: three layers with 400 and 300 neurons in the middle. The first input dimension is equal to the dimension of the observation space. The second input, connected to the second layer, has the dimension of the action space. The output equal to 1;
- learning rate: 0.001 for the critic, 0.0001 for the actor;
- target network update rate: 0.001.

The results are shown in figs 4 and 5. The training was done in two slots, due to the long training time. After $\sim$ 18000 episodes and 20 hrs the reward converges to 2700: the Cheetah is able to run!

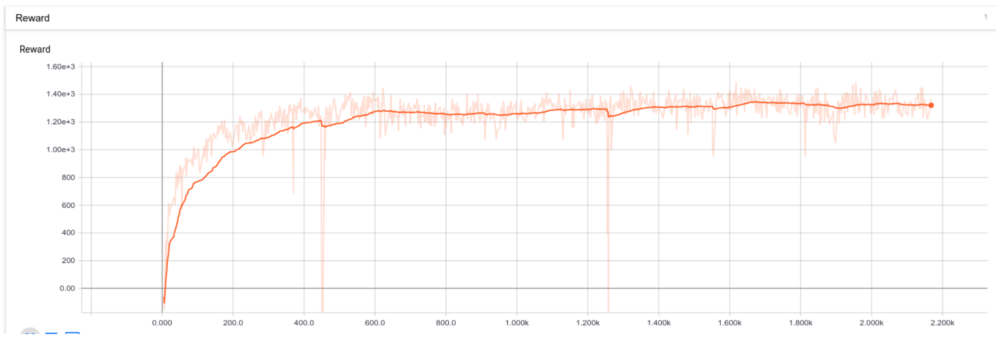


Figure 4: First training slot.

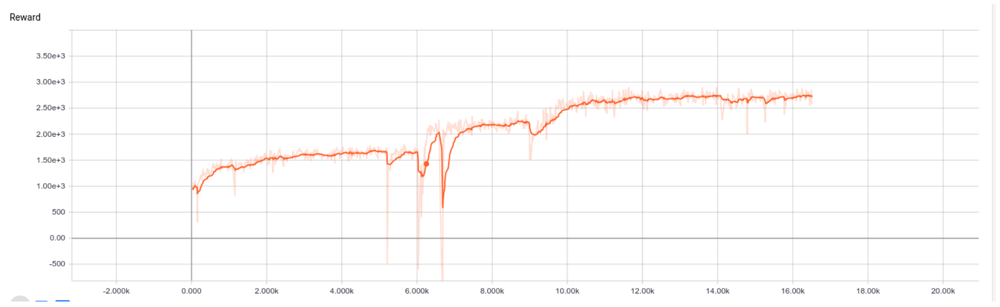


Figure 5: Second training slot.

5 Conclusions

As the results show, DDPG is very effective and stable algorithm, despite the use of the neural network.

DDPG is not the state of the art anymore. Nevertheless, it gave birth to a sequence of astonishing continuous control learning algorithm.

For example **A3C** from [5]. A3C uses multiple agents that interact with their own instance of the environment. They train their own network and share their results. In this way we do not need the replay buffer and the training is faster. An extension of this algorithm is **GA3C**, by Nvidia, that uses the GPU.

Still, in policy gradient methods a problem could be how large is the gradient. Indeed a big step can cause the agent ”fall off the hill”! The authors of [6] have found a proper way to restrict the policy change without decreasing the learning rate. **Trust Region Policy Optimization (TRPO)** speeds the training since the agent walks on a ”very safe path” that leads to the top of the hill.

More recently OpenAI created **Proximal Policy Optimization (PPO)**: it is based on the same consideration of TRPO but simplifies its formulation.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1928–1937, PMLR, 20–22 Jun 2016.

[6] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 1889–1897, PMLR, 07–09 Jul 2015.

[7] P. Emami, "Deep deterministic policy gradients in tensorflow."

[8] T. Simonini, "An introduction to policy gradients with cartpole and doom."

[9] B. Lau, "Using keras and deep deterministic policy gradient to play torcs."