SAPIENZA
UNIVERSITÀ DI ROMA

Reinforcement Learning implementation

DDPG + HER

Salvatore Cognetta 1874383

February 20, 2021

Contents

Chapter 1

Algorithms

1.1 DDPG: Deep Deterministic Policy Gradient

In real world scenarios, like pysical control tasks, there are continuous and high dimensional action spaces. Because it's not possible to apply Q-learning to continuous action spaces, caused by really slow optimization of the action a_t at every timestep, DDPG was introduced.
DDPG is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.
This algorithm is based on DPG, which uses an actor-critic approach. In this approach there are two different functions:

- an actor function $\mu(s|\theta^\mu)$, which specifies the current policy mapping states to a specific action;
- a critic function $Q(s|a)$, which is learned using the Bellman equation.

Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, to update the actor it's simply performed a gradient ascend:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

1.1.1 Target networks

In DDPG are used neural network function approximators to learn state and action spaces online. Moreover, because Q learning with neural networks is very unstable in very environments, the paper propose a copy for actor-critic networks, called target networks $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$, using soft target weights updates, instead of directly hard copying them. The weights of these target networks are updated accordingto the following rule:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

Therefore, the target values changes slowly, improving the stability of learning.

1.1.2 Replay Buffer

Using neural networks for reinforcement learning, althought introduces a new issue, in fact most optimization algorithms assume that the samples are independently and identically distributed. When the samples are generated from exploring sequentially, like in RL case, this assumption no longer holds. Moreover, to benefit from hardware optimizations, it's better to learn in mini-batches rather then online. To address this problems, the paper propose a replay buffer R, a finite sized cache which stores the transitions, tuples containing (s_t, a_t, r_t, s_{t+1}). At each timestep, actor and critic are updated by sampling a minibatch from the buffer.

1.1.3 Exploration and noise

Because DDPG is an off-policy algorithm to make DDPG policies explore better, noise is added to their actions at training time. The authors of the original DDPG paper recommended time-correlated Ornestein-Uhlenbeck noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. In this work there are implemented both, and one can choose by setting the variable $isOUNoise$ of the agent.
The resulting exploration policy is:

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

1.1.4 Pseudocode

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

Figure 1.1: DDPG pseudocode

1.2 HER: Hindsight Experience Replay

The idea behind Hindsight Experience Replay (HER) is very simple: after experiencing some episode $s_0, s_1, ..., s_T$ we store in the replay buffer every transition $s_t \rightarrow s_{t+1}$ not only with the original goal used for this episode but also with a subset of other goals.
There are different strategies for the additional goals:

- **finite**: the only additional goals used for replay were the ones corresponding to the final state of the environment;
- future: replay with k random states which come from the same episode as the transition being replayed and were observed after it;

- random: replay with k random states encountered so far in the whole training procedure;
- episode: replay with k random states coming from the same episode as the transition being replayed.

In this implementation the chosen goal is the final one of each episode.

1.2.1 Pseudocode

Algorithm 1 Hindsight Experience Replay (HER)

Given:
- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$. ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode = 1, M **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

Figure 1.2: HER pseudocode

1.3 Implementation

Different hyperparameters and the structure of the neural networks are taken from both papers.
For DDPG:

We used Adam (Kingma & Ba, 2014) for learning the neural network parameters with a learningrate of 10^{-4} and 10^{-3} for the actor and critic respectively. For Q we included $L2$ weight decay of 10^{-2} and used a discount factor of $\gamma = 0.99$. For the soft target updates we used $\tau = 0.001$. The neural networks used the rectified non-linearity (Glorot et al., 2011) for all hidden layers. The final output layer of the actor was a tanh layer, to bound the actions. The low-dimensional networks had 2 hidden layers with 400 and 300 units respectively ($\approx$ 130,000 parameters). Actions were notincluded until the 2nd hidden layer of Q. When learning from pixels we used 3 convolutional layers(no pooling) with 32 filters at each layer. This was followed by two fully connected layers with 200 units ($\approx$ 430,000 parameters). The final layer weights and biases of both the actor and criticwere initialized from a uniform distribution $[-3 \times 10^{-3}, 3 \times 10^{-3}]$ and $[3 \times 10^{-4}, 3 \times 10^{-4}]$ for the low dimensional and pixel cases respectively. This was to ensure the initial outputs for the policyand value estimates were near zero. The other layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The actions were not included until the fully-connected layers. We trained with minibatch sizes of 64 for the low dimensional problems and 16 on pixels. We used a replay buffer size of 10^6. For the exploration noise process we used temporally correlated noise in order to explore well inphysical environments that have momentum. We used an Ornstein-Uhlenbeck process (Uhlenbeck& Ornstein, 1930) with θ= 0.15 and σ= 0.2.

For HER:

We train for 200 epochs. Each epoch consists of 50 cycles where each cycle consists of running the policy for 16 episodes and then performing 40 optimization steps on mini batches of size 128 sampled uniformly from a replay buffer consisting of 106 transitions. We update the target networks after every cycle using the decay coefficient of 0.95. Apart from using the target network for computing Q-targets for the critic we also use it in testing episodes as it is more stable than the main network. The whole training procedure is distributed over 8 threads. For the Adam optimization algorithm we use the learning rate of 0.001.

Chapter 2

Environment

In this work the agent must be trained to perform the Fetch Pick And Place task, according to the DDPG + HER algorithm. This task consists of a robotic arm that has a gripper on the end-effector, this robo-arm must move in direction of a cube placed on a table, take it with the gripper hand and displace it to a specific goal.
This environment is taken from OpenAI Gym, using mujoco for the physic part.

2.1 OpenAI Gym

The OpenAI Gym framework makes available different kind of environments. After creating one of it, we can move according to a specified action (taken from our policy) with the step funcion. Each step returns an observation, a reward, a done boolean and finally an info array.
The *FetchPickAndPlace*, like all goal-based environments, uses a dictionary observation space. It includes a desired goal, which the agent should attempt to achieve (desired_goal), the goal that it has currently achieved instead (achieved_goal), and the actual observation (observation), e.g. the state of the robot.

Chapter 3

Considerations

Unfortunately the FetchPickAndPlace enviroment is not working as properly, probably because of some problem in her implementation. I've tried different experiments, but none of them seems improving the learning of this task. However the vanilla DDPG is working correctly, in fact it was tested on the Pendulum enviroment which brings to a very good result after only 200 epochs, as we can see in the image below.

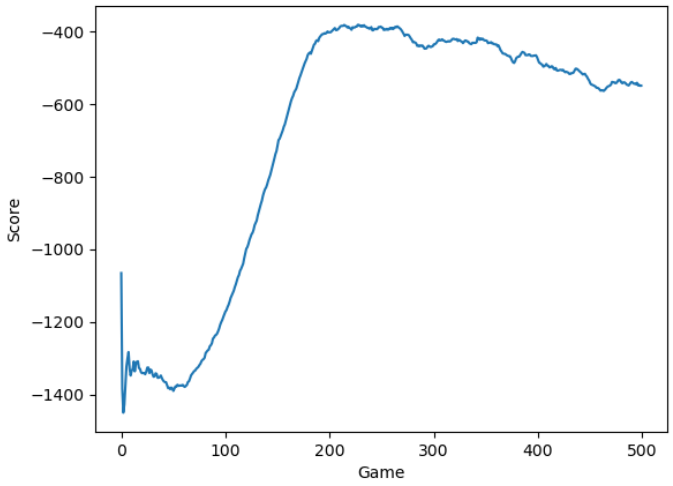

Figure 3.1: Pendulum results