DDPG + HER in FetchPush-v1 Gym Environment with Python3 and Tensorflow 2.0

Simone De Angelis 1760464
Veronica Romano 1580844

January 31, 2021

Chapter 1

Introduction and Work Purposes

Reinforcement Learning is learning what to do and how to map situations to actions. The end result is to maximize the numerical reward. The learner is not told which action to take, but instead must discover which action will yield the maximum reward. As illustrated in Figure 1.1, an agent, which is who have to reach a goal, stars from a state s_t and through actions a_t, manipulates the environment. From this it takes a reward r_{t+1}, if goes more close to the goal and it's intended to encourage good agent behavior, and discovers the next state s_{t+1}. These two are the reward and the state that permits the agent to re-implement the cycle.

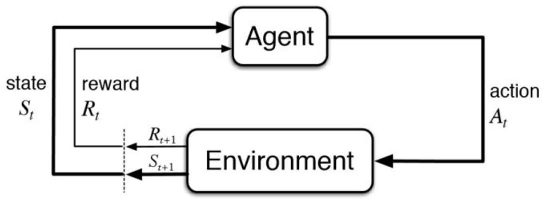

Figure 1.1: Formulation of a basic Reinforcement Learning project.

This proceeds until the goal is reached.

In few words, an *agent* chooses an action from the *action space* which it can perform in a given *environment* for which it gets *rewarded* if that action meets some criteria. The agent learn to take actions for maximizing this reward.

1.1 Environment and Tools

OpenAI Gym is a open source tool that allows us to work with algorithm for reinforcement. Previously there aren't many standard environment that could be used to the developed of these algorithms. In fact with the rise of Gym, reinforcement learning becomes more practical and implementable with respect to the traditional machine learning methods. Gym is available on the corresponding GitHub repository. On the other hand there is also MuJoCo (Multi-Joint dynamics with Contact). This is a physical engine for detailed, efficient rigid body simulations with contacts. It has a dynamic library with C/C++ API. mujoco-py allows using MuJoCo from Python3. It includes an XML parser, model compiler, simulator, and interactive OpenGL visualizer. To use it a specific license is needed; we use this for visualizing and using our environment, and to provide a graphical visualization of the results reached. MuJoCo is also customizable, so the environment can be modified by changing its XML code. This permits also to create new environments for implementing new tasks. However the best part is that MuJoCo provides the physical interactivity (like calculation of contact forces) which helps an engineer or researcher to test their model rigorously before moving to production.

This work consists in implementing the FetchPush-v1 environment provided by Gym OpenAI (using MuJoCo as physical simulator), and by using a specific algorithm, which in our specific case is DDPG+HER, experimenting on this environment. Also a specific framework will be assigned for implementing the code. All the result reached will be illustrated in Chapter 3. While in the following sections we will exploit in details the algorithm used for our task, the environment, and the implementation framework.

1.1.1 FetchPush Gym Environment

The release provided by OpenAI Gym, contains four environments using the Fetch research platform. These environments use the MuJoCo physics simulator. Our environment is, as previously mentioned, FetchPush-v1 (Figure 1.2). Fetch goal is to move a box by pushing it until it reaches a desired goal position. This tasks has the concept of a "goal". By default it uses a sparse reward of -1 if the desired goal was not yet achieved and 0 if it was achieved (within some tolerance). This because the sparse rewards are more realistic in robotics applications and the developers encourage their use.

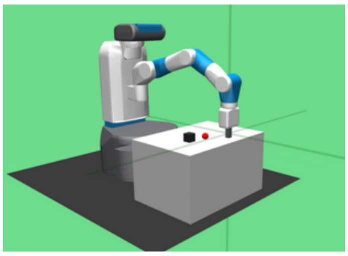

Figure 1.2: FetchPush-v1 Gym environment.

The Fetch environment is based on the 7-DoF Fetch robotics arm which has a two-fingered parallel gripper that are locked to prevent grasping. The learned behavior is usually a mixture of pushing and rolling. The goal is 3-dimensional and describes the desired position of the object (or the end-effector for reaching). Rewards are sparse and binary in fact the agent obtains a reward of 0 if the object is at the target location and 1 otherwise. Actions are 4-dimensional: 3 dimensions specify the desired gripper movement in Cartesian coordinates and the last dimension controls opening and closing of the gripper that in FetchPush-v1 is always closed. However in other environments, like FetchPickandPlace-v1, this last dimension is very relevant for the task. Observations include the Cartesian position of the gripper, its linear velocity as well as the position and linear velocity of the robot's gripper. If an object is present, we will have also the object's Cartesian position and rotation using Euler angles, its linear and angular velocities, but also the position and linear velocities relative to gripper. The observation space is of type gym.spaces.Dict space, with at least three keys that are:

- observation: It's an array of 25 elements. The first 3 elements represent the position of the gripper, the following 3 is the achieved goal so the position of the box in the actual state.
- desired_goal: The goal that the agent has to achieve. In FetchPush-v1 would be the 3-dimensional target position on which we would like the box will be
- achieved_goal: The goal that the agent has currently achieved instead. Ideally, this would be the same as desired_goal as quickly as possible.

Chapter 2

Implementation Assignment

2.1 DDPG + HER Algorithm

Deep Deterministic Policy Gradient (DDPG) (illustrated in [3]) is a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces.

2.1.1 DDPG (Deep Deterministic Policy Gradient)

It concurrently learns a Q-function and a policy. As DPG, it uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. DDPG, being an actor-critic technique, consists of two models: Actor and Critic. The Actor is a policy network that takes the state as input and outputs the exact action (continuous), instead of a probability distribution over actions. So the actor function $\mu(s|\theta^{\mu})$ specifies the current policy by deterministically mapping states to a specific action. The Critic $Q(s,a)$ is a Q-value network that takes state and action as input and outputs the Q-value. It is learned using Bellman equation as in Q-learning. DDPG is used in the continuous action setting and the word "Deterministic" in DDPG refers to the fact that the actor computes the action directly instead of a probability distribution over actions. This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving:

$$a^*(s) = argmax_a Q^*(s,a) \tag{2.1}$$

DDPG put together the learning of an approximator $Q^*(s,a)$ and the learning of an approximator $a^*(s)$, and it does this in a way which is specifically for continuous action spaces. This refers to the way in which the max over actions is computed in $\max_a Q^*(s,a)$. When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. However, when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

As in DQN, DDPG uses a replay buffer which is a finite sized cache R. Transitions were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. When the replay buffer was full, the oldest samples were discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. The replay buffer can be large due to the fact that DDPG is an off-policy algorithm. It allows the algorithm to benefit from learning across a set of uncorrelated transitions. A copy of the actor and critic networks is created; respectively $Q'(s,a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$. These copies are used for calculating the target values. Also the weights of these target networks are updated slowly to make them following the learned networks. This means that the target values are constrained to change slowly, and this improves the stability of learning. The authors of [3] found that having both target μ' and Q' is required to have stable targets in order to consistently train the critic without divergence. Moreover the main advantage of off-policy algorithms such as DDPG is that the exploration can be done independently from the learning algorithm. The exploration policy μ' is obtained by adding noise to our policy $\mu(s_t|\theta_t^{\mu})$.

The Figure 2.1 shows the pseudocode of the algorithm.

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 2.1: Pseudocode of DDPG algorithm.

2.1.2 HER (Hindsight Experience Replay)

Hindsight Experience Replay is a reinforcement learning algorithm which can learn from failure. It's been proven that HER can learn successful policies on most of the new robotics problems from only sparse rewards. In many tasks the first iteration is not a success or if it's so, it's probably only a lucky episode. A classical reinforcement algorithm will not learn anything from this experience since it just obtains a constant reward equal to -1, that does not contain any learning signal. HER allows to learn also from this type of experiences. In fact also if we have not succeeded at a specific goal, we have at least achieved a different one. We can imagine that we want to reach this new goal at the beginning instead of that we want to achieve originally. By doing this substitution, the reinforcement learning algorithm can obtain a learning signal since it has achieved some goal; even if it wasn't the one that we meant to achieve originally. If we repeat this process, we will eventually learn how to achieve arbitrary goals, including the goals that we really want to achieve. This technique is called Hindsight Experience Replay because it replays experience with goals which are chosen in hindsight, after the episode has finished. More deeply, the idea behind HER is that after experiencing some episode we store in the replay buffer every transition from a state to the following, not only with the original goal used for this episode but also with a subset of other goals. The goal will influence only the agent's action and therefore we can replay each trajectory with an arbitrary goal assuming that we use an off-policy algorithm like DDPG. One important aspects is the choice of the additional goals used for replay. In the simplest version the goal is substituted with the goal achieved in the final state of the episode. Another version is to choose the goal in a random way. The complete algorithm of HER is illustrated in Figure 2.2

In [2] to show the performances of HER, are used different Fetch environments including FetchPush. In this paper, the DDPG algorithm is used for performing training with Adam as optimizer, as in our case. Also in this paper is shown that DDPG without HER is unable to solve any of the tasks. It confirms that HER is a crucial element which makes learning from sparse, binary rewards possible. Moreover it also states that DDPG+HER performs better even if the goal state is identical in all episodes. In [2] are also shown several strategies for choosing goals to use with HER. As mentioned one strategy is to choose the goal corresponding to the final state of the environment (*final* strategy). Other strategies are:

- *future* that replays with k random states which come from the same episode as the transition being replayed and were observed after it
- *episode* that replays with k random states coming from the same episode as the transition being replayed
- *random* that replays with k random states encountered so far in the whole training procedure

All of these strategies have a hyperparameter k which controls the ratio of HER data to data coming from normal experience replay in the replay buffer. In the paper is shown that all these strategies excluded the *random* one are able to solve pushing task regardless the value of k.

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 $G := \mathbb{S}(\textbf{current episode})$
    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 2.2: Pseudocode of HER algorithm.

2.1.3 DDPG and HER work together

HER works very well in goal-based environments with sparse rewards. Vanilla DDPG and DDPG+HER have been compared. This comparison includes the sparse and the dense reward versions of the environment. Has been shows that DDPG+HER with sparse rewards outperforms significantly Vanilla DDPG with sparse or dense rewards and DDPG+HER with dense rewards. There results are exposed in [1]. This paper in fact evaluates the performance of DDPG with and without HER on all the Fetch and HandManipulateBlock environments.

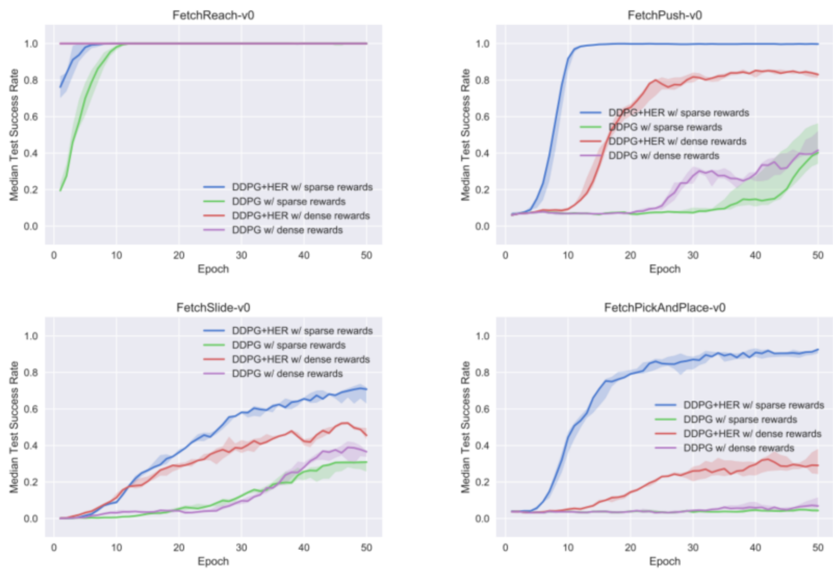

Figure 2.3: This Figure illustrates the comparison between Vanilla DDPG and DDPG+HER with sparse and dense rewards in all four Fetch environments provided by OpenAI Gym.

The results reached for Fetch environments, in which we are mainly interested, are illustrated in Figure 2.3. It depicts the median test success rate for all four Fetch environments. FetchReach-v0 is clearly a very simple environment and can easily be solved by all four configurations. On the remaining environments, DDPG+HER clearly outperforms all other configurations. Interestingly, DDPG+HER performs best if the reward structure is sparse but is also able to successfully learn from dense rewards. For vanilla DDPG, it is typically easier to learn from dense rewards with sparse rewards being more challenging. Moreover, as we can see from the Figure 2.3, for FetchPush-v0 environment the combined approach DDPG+HER with sparse reward is that reached better performance and better success rate. the authors believe that a possible reason why DDPG+HER typically performs better with sparse rewards is mainly due to the following two reasons. The first reason is that learning the critic is much simpler for sparse rewards. In the dense case, the critic has to approximate a highly non-linear function that includes the euclidean distance between positions and the difference between two quaternions for rotations. On the other hand, learning the sparse return is much simpler because the critic has only to differentiate between success and failed states. The second reason is that a dense reward biases the policy to a specific strategy. The dense reward however encourages the policy to chose a strategy that achieves the desired goal directly

Chapter 3

Training Procedure and Plots

As previously mentioned, our work consisted of exploring in FetchPush-v1 environment, implementing the algorithm DDPG+HER. To implement our work we write a code in Python3 and TensorFlow 2.0. The code is structured in this way:

- a *buffer.py* file in which is implemented the replay buffer with high dimension in which the transitions are stored through the use of agent function *remember()*. These transitions include state, action, reward, new state reached after implementing an action, a flag *done* which indicates if the goal is reached and the goal. These stored transitions will be then used to define the new goals for HER implementation. In this file is also defined a function called *sample_buffer()* which is used in DDPG implementation for sampling a random minibatch of N transitions from the replay buffer.

- a *ddpg_tf2.py* file in which all the parameters for the agent are defined and the DDPG algorithm is implemented. Here are also initialized the target actor and target critic networks with the parameters reported in Table 3.1, size of the replay buffer, the value of the discount factor γ, the value for updating the target networks τ and also the noise is set. It is used for adding noise to the policy for the choice of the action to execute. In this file are also created the actor and critic target networks by coping the original actor and critic networks. To each of the for networks is applied an Adam optimizer with different learning rate for the actor and critic networks. Here are defined a function for updating the network parameters that takes tau as input and implement the last stage of the pseudocode of DDPG algorithm, a remember function for storing the transitions, a function for saving and one for loading the models of the networks. In the file is also defined a function for choosing an action which takes as input the state and the goal, concatenate these as illustrated in the pseudocode of HER and gives it to the actor which computes the policy for choosing the action. This function returns the action. The last function defined is the learning function, in which the critic network is updated by computing the target and applying the Mean Squared Error loss. While the actor network is updated by applying the gradient ascendant. this permits to update the networks parameters.

- a *networks.py* file in which the actor and critic networks are created by following the conventional parameters used in this kind of tasks found on several papers regarding this argument (also reported in Table 3.1). The networks are created by using *tensorflow.keras* and are formed by only dense layers with *relu* as activation function, and a final critic dense output layer without activation function, and a final actor dense output layer with *tanh* as activation function. Moreover the weights of the networks for training are saved in 2 files with extension *.h5*, one for the critic network and one for the actor network. the critic network will return the Q function, while the actor network will return the policy μ.

- a *main_ddpg.py* file in which the environment is imported. To the agent, so the DDPG algorithm, are provided the environment, the dimension of the *observation* space as input dimension, the dimension of the *desired_goal* and the *action_space* dimension. this file is created by following the stages of the pseudocodes as deeply illustrated in Chapter 2.

All the parameters we have used are taken from literature works and from the papers cited in the Bibliography. In the following section the performed experiments are deeply explained. The following Table 3.1 reports all the parameters used for all the experiments while the changed parameters will be illustrated in the specific section.

Parameters Table	
FC layers Critic	2
FC Critic dimensions	512
Learning Rate Critic	0.002
FC layers Actor	2
FC Actor dimensions	512
Learning Rate Critic	0.001
Discount factor γ	0.99
Update factor τ	0.005
Noise	0.1
Replay Buffer dimension	1000000

Table 3.1: Table of the parameters used for all the experiments

3.1 Experiments and Plots

We performed some experiments changing the way in which HER is implemented. First of all we implemented first only the DDPG algorithm but as we expected this provided no good results. Then we added also HER. In this experiment we implement a sort of *episode* HER. It takes k random states from the same episode and replays them as the transition being replayed. For the first groups of experiment we try to change the parameter k that establish the number of states taken for replaying. Following the procedure implemented in [2] we performed these experiments to show that changing the value of k, the system is always able to learn, as explained in [2]. By following more strictly their training procedure (in detail what explained in Appendix A) we performed a training with 200 epoches and each epoch is formed by 50 cycle and each cycle by 16 episodes. In this way each epoches contains $16 * 50 = 800$ iterations. The learning is performed only if the robotic arm effectively reach the goal. In fact at the end of each cycle are performed 40 optimization steps in which the learning takes place. In this case a real episodic HER is implemented. We started by taking k random box stored positions and then if, for each of them, the condition that the random box position is closer to the goal with respect the initial position (it means that the box has been moved toward the goal), we stored this current transition. However as previously mentioned the learning is done only at the end of each cycle. This approach works, in fact as mentioned in [2] for each value of k in the episodic HER we can obtain good results. However, we found a problem with this approach to which we have tried to give an explanation. The problem is that we reached different results as we can see from the plots in Figure 3.1, 3.2, 3.3, 3.4; moreover in different experiments with the same parameter k, sometimes the robot doesn't learn and sometimes it reach a normalized success rate equal to 92%, 98% or equal to 78% (reported in Figure). We think that the problem could be in the fact that the procedure takes random transitions from the replay buffer and so random position for the box. Moreover during the training we noted that if the initial random positions are good positions, in such a way that the condition of HER is verified, the system generally learns and the results are quite better. Another possible reason is that the learning should be slower and so increasing the number of epochs, also in that case it should be able to reach a good value of success rate. As we can see from the plot, for different value of the parameter k we are able to learn. In Figure 3.3 we can see that after reaching a normalized success rate equal to 92%, this rate decreases. However in the code we imposed a condition for which the model are saved only if the *win_percent* value is higher than the one in the previous step, so it means that the models have not been updated when the value of *win_percent* decreases. As test we verify the models for which we have reached the best normalized success rate, so the models collected with k=24. We set 100 as the number of iterations for the test, and only 1 time on 100 it didn't reach the goal. The Table 3.2 is a report of the parameters used for these experiments.

For the second two experiments we tried to modify the learning procedure. The concept is the same of the previous experiments. We perform this in two ways, by taking the first 24 states of the same episode and the last 24 states of the same episode. They are not randomly taken. We performed a learning with 50000 episodes, and each episode is formed by 50 iterations or steps. During these iterations we evoked the learning function from the agent class each time also if the robotic arm didn't still reached the goal, so when *done=False* and after adding the current transition to the replay buffer. If the robot reached the goal and so *done=True*, we increased the success rate. While for each episode, after iterating all the 50 steps, if the robot has not still reached the goal and so *done=False*, we take the final box position from the state (these are the third, forth and fifth components of the state). At this point if the HER condition is verified, such that the distance between the final box position and the initial box position is less or equal than 0.05, for

Parameters Table for the First Set of Experiments	
Batch Size	128
Number of Episodes	16
Number of Cycles	50
Number of Epochs	200
k	4-8-16-24
Number of Epochs for test	100

Table 3.2: Table of the specific parameters used for the second experiment with k equal to 8 random transitions.

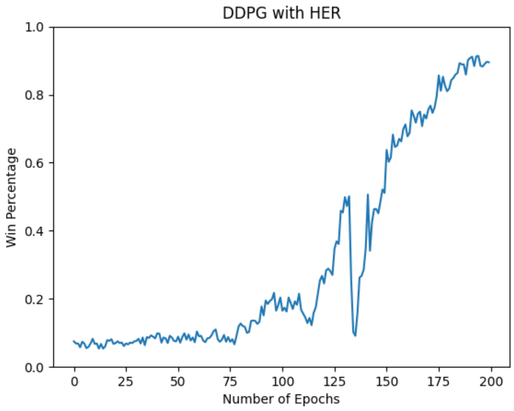

Figure 3.1: Plots obtained by setting $k = 4$, success rate reached 93%.

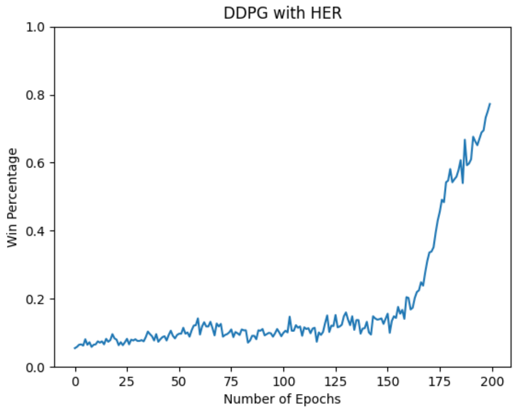

Figure 3.2: Plots obtained by setting $k = 8$, success rate reached 78%.

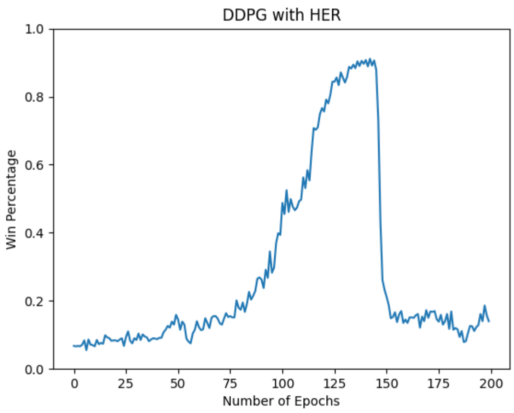

Figure 3.3: Plots obtained by setting $k = 16$, success rate reached 92%.

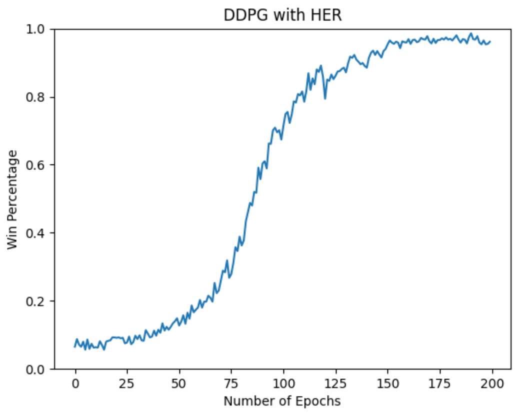

Figure 3.4: Plots obtained by setting $k = 24$, success rate reached 99%.

each of the 24 states taken from the beginning or from the end, we take a random stored position for the box in the current episode. Now if the final box position is more close to the goal than to the starting position, we stored this transition with the random position of the box, and we perform the learning. However the learning is performed also if the last condition is not verified and so for each of the 24 transition considered. This because we consider the fact not only that the box is near to the goal but also the fact that the end-effector of the robotic arm has moved the box. As we can see from Figure 3.5 the learning produces good results reaching a success around 95% both by considering the first and the last 24 stored transition and also the plots result to be very similar. However we have to mentioned that the training by considering the first 24 transitions is faster that the one that considers the last 24 transitions. In the Table 3.3 are reported the specific parameters used for the first two experiments. While the results are illustrated in Figure 3.5.

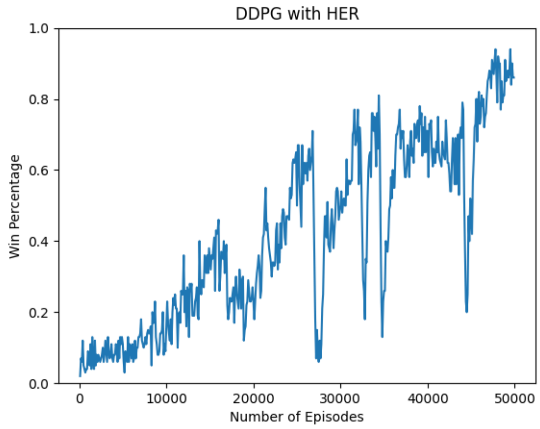

Figure 3.5: Plots of the normalized success rate for the last 24 states taken, with normalized success rate equal to 95%.

Parameters Table for the Second Set of Experiments	
Batch Size	64
Number of episodes	50000
k	24 (first and last)

Table 3.3: Table of the specific parameters used for the second two experiments.

Bibliography

[1] Plappert, Matthias, et al. ”Multi-goal reinforcement learning: Challenging robotics environments and request for research.” arXiv preprint arXiv:1802.09464 (2018).

[2] Andrychowicz, Marcin, et al. ”Hindsight experience replay.” Advances in neural information processing systems. 2017.

[3] Lillicrap, Timothy P., et al. ”Continuous control with deep reinforcement learning.” arXiv preprint arXiv:1509.02971 (2015).