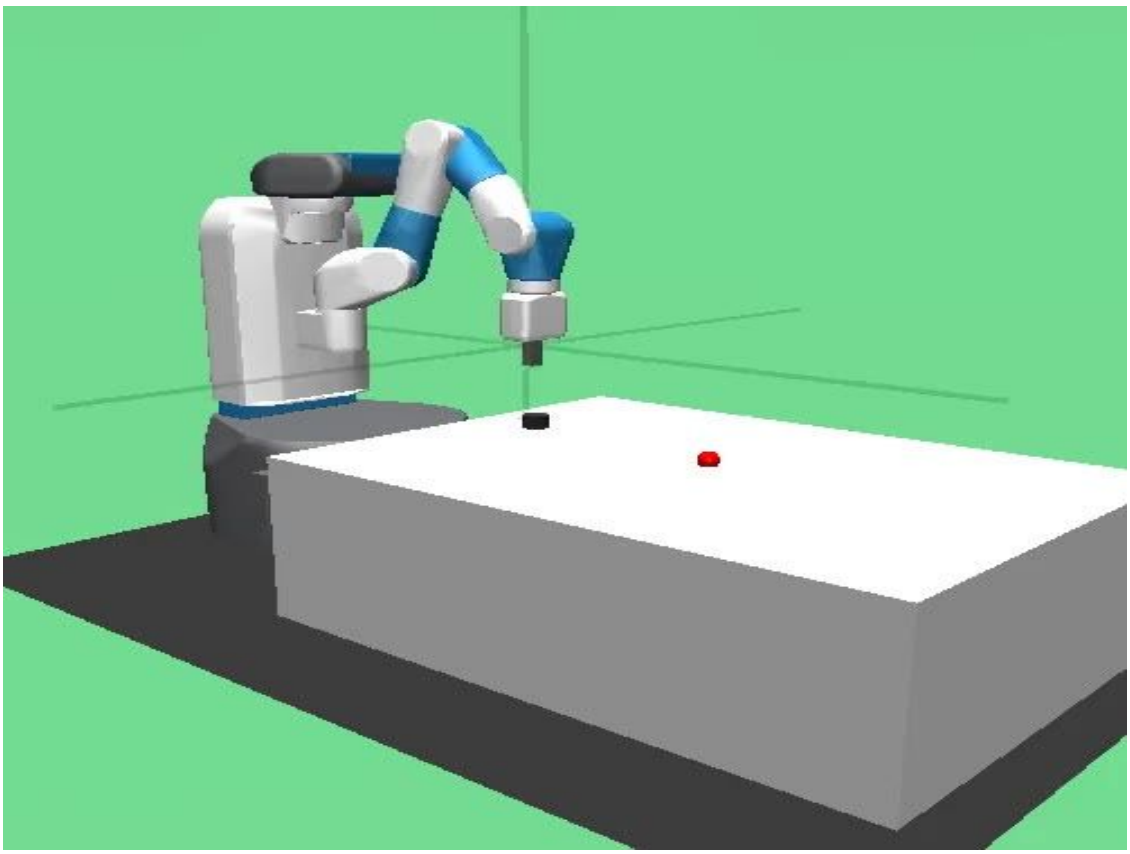
REINFORCEMENT LEARNING PROJECT REPORT

Algorithm: Deep Deterministic Policy Gradient (DDPG) + Hindsight Experience Replay (HER)

Environment: FetchSlide-v1



Caterina Borzillo (id: 1808187)

Federica Cocci (id: 1802435)

february 2021

CONTENTS

1. Introduction
2. Environment
    a. FetchSlide-v1
    b. Environment's representations
3. Deep Deterministic Policy Gradient (DDPG)
    a. Models: Actor and Critic
    b. Algorithm implementation
4. Hindsight Experience Replay (HER)
    a. Replay strategies
    b. Algorithm implementation
5. Implementation
6. Conclusions

1 INTRODUCTION

The purpose of this Reinforcement Learning project is to teach an agent to accomplish a specific task which is sliding a puck across a table and hitting a target. In particular, a goal position is chosen on the table and out of reach for the robot arm and the end-effector of the robot has to slide the given puck to this goal.

Our learning experience is based on a novel and interesting technique called Hindsight Experience Replay (HER) which lets the agent learn from sparse and binary rewards. This technique cooperates with the Deep Deterministic Policy Gradient algorithm (DDPG) which is an off-policy RL algorithm that is used for continuous action spaces.

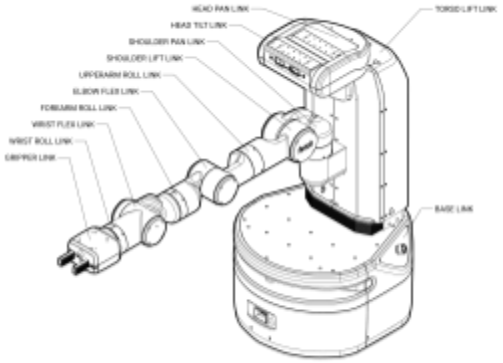
In our experiment, we make use of MPI (Message Passing Interface) which is a communication protocol specifically used to allow applications to run in parallel.

2 ENVIRONMENT

We had to work with one of the eight simulated robotics environments released by OpenAI: all of these environments are used to train models which work on physical robots. Our environment is called "FetchSlide-v1" and we have used MuJoCo as a physics simulation.

2.a FetchSlide-v1

In this environment, there is a 7-DOF Fetch Robotics arm. The arm has a two-fingered parallel gripper. In general the robot can perform several tasks: Pushing, Sliding, Pick-and-Place. We have worked with the task of sliding (hence the environment's name): there are a puck, placed on a table, and a target position that is outside the robot's reach (so it has to hit the puck with a force that the object slides and then stops in the right point of the table due to friction).



Fetch Robotics arm

2.b Environment's representations

Here we present how states, goals, rewards,... are represented in the MuJoCo engine.

- States: the state of the system consists of angles and velocities of all robot joints and positions, rotations and velocities (linear and angular) of all objects.
- Goals: the goal describes the desired position of the puck with some tolerance (noise) ε.
- Rewards: we use binary and sparse rewards $r(s, a, g) = -[f_g(s') = 0]$ where s' is the state after the execution of the action a in the state s.
- Actions: the action space is 4-dimensional (three dimensions specify the desired relative gripper position at the next timestep; "relative" means relative to the current gripper position). The last dimension is about the desired distance between the 2 fingers.
- Strategy for sampling goals for replay: HER uses to replay with the goal corresponding to the final state reached in each episode but there are different strategies for choosing which goals to replay. In our implementation, we have chosen "future".

3 DEEP DETERMINISTIC POLICY GRADIENT (DDPG)

The algorithm we're going to use to let the agent learn its task is DDPG, which is a reinforcement learning technique that combines Q-learning and Policy gradients methods to find the solution. That's because DDPG is an actor-critic technique that is composed by two models (or, in particular, two neural networks): the Actor and the Critic networks. The actor is a policy network that has the task of choosing the next exact continuous action, given a state as input. The critic, instead, is a Q-value network that, given a state and an action as input and returns as output the Q-value that is a value that tells us how useful a given action is in gaining some future rewards.

DDPG has in its name the word "deterministic", which refers to the fact that the actor network returns as output an action, instead of a probability distribution over actions. Also, this algorithm is an off-policy method because the agent interacts with the environment to collect samples that will be stored in a data buffer (called replay buffer) and this data buffer information will be used then to train an updated new policy.

While in DQN (Deep Q-Networks) the optimal action is chosen by taking the argmax over the Q-values of all actions, in DDPG the actor is a policy which does exactly this computation, i.e. it chooses and outputs directly the continuous action.

3.a Models: Actor and Critic

To implement the algorithm we use two models, that are two neural networks that are chained together. The image below clearly defines the architecture of the Actor and Critic networks.

Actor and Critic networks architecture.

In our implementation of the networks we build the actor module as a neural network composed of 2 fully connected hidden layers, both defined with 256 units and in which we use the Rectified Linear Unit (ReLU) as activation function. In input to the actor we give the observation and the goal (because in this problem there is more than one goal we may try to achieve and for this reason we want to specify which goal we're referring to) and in the last layer since we want to output a continuous action, we use the hyperbolic tangent function which is then rescaled so that it lies in the range of valid actions. Concerning the Critic model,we build the critic neural network in a very similar way with respect to the Actor network, because it has 2 dense hidden layers with 256 units and as input we give to the network not only the observation and the goal but also the action that has been chosen by the actor network. As output the critic returns the Q-value, which is the value that represents how good or bad is the action that the actor takes out.

3.b Algorithm implementation

In a few words, the DDPG algorithm is composed of an initial step where we initialize both Actor and Critic networks, (in particular we initialize the parameters) and both actor target and critic target networks which are delayed networks that are used to be compared with the main networks. The weights of targets are "softly" updated periodically based on the principal actor and critic networks, "softly" in the sense that only a part of the actual networks weights are transferred to the target ones (according to a parameter that typically is close to 1). We also in this first phase initialize the replay buffer R in which we'll put the past experiences. Now, there is the second phase in which we start M number of episodes, each consisting of T timesteps. At each timestep the next action is selected according to the current policy and exploration noise and after we the agent execute it, we'll observe the reward and the new state. After this, we store this latter transition in the replay buffer R. Then we sample a minibatch of transitions from R in which we recall that we don't have only the transitions in which the goal to be achieved is the original goal, but also that transitions that belong to HER (transitions in which we consider additional goals, with positive reward). In the end, we update both critic and actor parameters by minimizing their loss functions.

4 HINDSIGHT EXPERIENCE REPLAY (HER)

The idea behind HER is from the human experience: one ability humans have is to learn

almost as much from achieving an undesired outcome as from the desired one.
A standard RL algorithm doesn't learn anything from the sequence of actions that doesn't lead us to a successful conclusion: when there is a situation of unreached goal, with HER, the agent learns the sequence of actions that would be a successful path if the goal was the final state of the sequence.

4.a Replay strategy

Each episode is composed by a sequence of states $s_1, ..., s_T$ and there is a goal $g \neq s_1, ..., s_T$: this situation implies that the agent, at each step, gets a reward of -1: now we can replay the experience where we replace the goal g , in the replay buffer, with the state s_T getting a positive reward (by the way we can still replay the original goal g because it is left intact in the replay buffer). To replay each episode with a different goal than the one the agent was trying to achieve, there are several ways to sample these additional goals:

- final: the additional goal is the one corresponding to the final state of the episode.
- future: replay with k random states which come from the same episode as the transition being replayed and were observed after it.
- episode: replay with k random states coming from the same episode as the transition being replayed.
- random: replay with k random states encountered so far in the whole training procedure.

The factor k is very important because the rate of goals to be replaced by additional goals depends on it; in our implementation we set k equals to 4 (the more k, the more is the number of goals to be replaced).

4.b Algorithm implementation

The HER algorithm can be summed up in six steps:

1. execute an episode using DDPG
2. for each transition in the episode, got from the step 1, compute the reward and store the transition in the replay buffer
3. sample a set of additional goals
4. for each additional goal, got from the step 3, compute the reward and store this transition in the replay buffer
5. sample a minibatch from the replay buffer and train the network with this subset using DDPG
6. repeat M times from step 1 to step 5 (M=number of episodes)

5 IMPLEMENTATION

We have divided the implementation in three folders: the most important one is called DDPG_HER where we can find the code about DDPG agent, HER implementation and actor-critic models.
There is a file to define the parameters for the system, e.g environment's name, epochs number, replay strategy and some hyperparameter for the neural network.
The 'demo' file is used to play a demo with MuJoCo and a pretrained model.
The 'train' file is used to train the model.
When the model finishes its training, we save it in the folder 'saved_model' and we generate a log file to store the success rate permanently at the end of each epoch.

It is important to say that we have used some code from openAI baselines, in particular to work with mpi, with the normalizer and with the replay buffer.

6 CONCLUSIONS

We have trained the model for 50 epochs and each epoch is composed of 50 cycles where each cycle consists of 16 episodes. Since we've worked with 8 cores, the number of episodes assigned to each core is 2.
The reduced number of training epochs is due to our limited available technology: training for 50 epochs + evaluation have taken 3.30h. At the end of the process, the highest success rate we've reached is 0.712.
As we can notice, the success rate of learning starting from the 1st to the last epoch is ascending, thus it seems that if we had increased the epoch number, we would have reached a higher success rate.
In conclusion, the HER novel technique permits us to implement a learning algorithm in an efficient way with a quite high performance (even with only 50 epochs of training) despite we're facing a very challenging and wide reinforcement learning problem with sparse and binary rewards.