Improving Hindsight Experience Replay (HER) on the Fetch Robotics Environments

Hanlu Li
Electrical and Computer Engineering Dept.
University of California, San Diego
La Jolla, USA
hal037@ucsd.edu

Allen Zeng
Electrical and Computer Engineering Dept.
University of California, San Diego
La Jolla, USA
azeng@ucsd.edu

***Abstract*—For complex robotics environments, Hindsight Experience Replay (HER) [1] can be used to help off-policy reinforcement learning algorithms like Deep Deterministic Policy Gradient (DDPG) learn from failure in addition to successes. This enables reinforcement learning algorithms to train in environments that have sparse and binary rewards–often the only rewards given in real-world scenarios. We extend the work of HER in two directions: (1) we replace the DDPG portion of the baseline algorithm with a newer off-policy algorithm called Twin Delayed DDPG (TD3); (2) we incorporate pre-trained agents' successful demonstrations into training new agents. In this work, we primarily train and test our modifications on the Fetch robotics environments for the tasks of puck-sliding and block pick-and-placing. Our results show that each individual modification and the combined modifications improve upon the baseline algorithm. The agents trained using HER and demonstrations end up outperforming the demonstrating agents trained on HER alone.**

I. Introduction

Nowadays various industries are using industrial robots to do repetitive tasks such as fetch, move and place objects in a preprogrammed trajectory. However, in the reality, robots actually have limited ability to handle unseen or complex environments. The purpose of this project is to design an algorithm that can let robot complete complicated task in new environments. The main environment that is used for experiment is Fetch robotics environments from OpenAI gym [2], [3], and specifically are FetchSlide-v1 and FetchPickAndPlace-v1. The challenge of this projects comes from the sparse rewards of the complicated environment dynamics. Agents rarely see reward signals when searching randomly, making it difficult to obtain learning signals in low-reward environments. Here is where reinforcement learning comes in to use, the method we use to solve this problem is to train with Hindsight Experience Replay (HER) [1] combined with off-policy reinforcement learning algorithm TD3 [4].

II. Background

A. Reinforcement Learning

Reinforcement Learning is a type of machine learning which allows agents to learn in an interactive environment by using rewards and punishments as signals for positive and negative behavior. The agent needs to collect sufficient information to make the best overall decisions in the future. Markov Decision Processes (MDP) is mathematical framework to describe an environment in reinforcement learning and we will formulate our problem using MDP. An MDP consists of a finite set of environmental states S, a set of possible actions A(s) in each state, a reward function R(s), a transition model $P(s', s|a)$, and a discount factor $\gamma \in [0, 1]$.

At the beginning of every episode, initial state s_0 will be sampled, and at each timestep t, the agent generates an action based on the current state: $a_t = \pi(s_t)$. Then a reward $r_t = r(s_t, a_t)$ will be given and the new state is sampled from the transition model $P(s', s|a)$. A return is a discounted sum of future rewards which can be formulated as $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. And the goal of the agent is to learn a policy $\pi : S \rightarrow A$ to maximize expected returns $E_{s_0}[R_0|s_0]$.

B. Fetch Environments

In this work, we primarily focus on the FetchSlide-v1 and FetchPickAndPlace-v1 environments of the Fetch robotics environments [3]. The Fetch environments are implemented in the OpenAI Gym [2], and use the Mujoco [5] physics simulator. The Fetch environments are based on the 7-DoF Fetch robotics arm which has a two-fingered gripper [6]. There are four Fetch environments: FetchReach-v1, FetchPush-v1, FetchSlide-v1, and FetchPickAndPlace-v1. In FetchReach-v1, the robot must move its gripper to a randomized target position. In FetchPush-v1, the robot must push a block to a randomized target position on a tabletop. In FetchSlide-v1, the robot must hit a puck across a long table such that it slides and comes to rest on the desired, randomized goal. For FetchPickAndPlace-v1, the robot needs to pick up an object from a table using its gripper and move it to a desired, randomized goal on or above the table.

In all of these environments, the robot must move the object to within a 5 cm tolerance of the goal. The rewards in these environments are sparse and binary: the agent obtains a reward of 0 upon reaching the goal, and -1 otherwise. This is intended to be a difficult but realistic reward signal to the real world, where tasks are often described only as successful or unsuccessful.

FetchSlide-v1 and FetchPickAndPlace-v1 are considered to be the more difficult environments to solve. In the former, the goal locations are out-of-reach of the robotic arm. So, the robot

has to hit the slippery puck such that it comes to a stop with a tolerance of the goal position. This is difficult because the robot must figure out it needs to reach the puck, and move in a manner that the gripper hits the puck with enough force that it slides and stops at the goal due to friction; all with the sparse, binary reward signal. In FetchPickAndPlace-v1, the robot must similarly reach the block, and then keep the gripper closed on the block as it moves to the goal position. With standard algorithms like DQN [7] and DDPG [8], random exploration is not enough to train an agent to solve these environments.

C. DQN

Deep Q-Networks (DQN) [7] is a model-free reinforcement learning algorithm that is used for discrete action spaces. It is similar to Q-learning which are based on value iteration. In ordinary Q-learning, when the state and action space are discrete and the dimension is not high, Q-Table can be used to store the Q value of each state-action pair. However, when the state and action space are high-dimensional and continuous, it is very difficult to use the Q-Table. Therefore, DQN solves this problem by using 4 techniques. The first technique is using experience replay to stores experiences including state transitions, rewards and actions, which are used to perform Q learning and makes mini-batches to update neural networks. The second technique is using target network to fix parameters of target function and replaces them with the latest network every set number of steps, where $Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a)]$. Third technique is clipping rewards by making all positive rewards as +1 and all negative rewards as -1 which makes training more stable. The last technique is skipping frames, DQN calculates Q values every 4 frames and use the previous 4 frames as inputs. It reduces computational cost while also accumulating more experience.

D. DDPG

Deep Deterministic Policy Gradients (DDPG) [8] is also a model-free reinforcement learning algorithm that learns Q functions and a policy at the same time. It uses off-policy data and Bellman equations to learn the Q-function, and use the Q function to learn the policy.

There are two neural networks used in DDPG, one is a target policy which is called an actor $\pi : S \rightarrow A$, another is a critic function $Q : S \times A \rightarrow R$ which is used to approximate the action-value function of actor Q^{π}.

A noisy version of the target policy is used to generate the episode $\pi_b(s) = \pi(s) + N(0, 1)$ and the target is computed using the actions output from actor $y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}))$. The loss for this combined actor and critic networks are $\mathcal{L}_a = -\mathbb{E}_s Q(s, \pi(s))$. The state s is sampled from the replay buffer. The gradients with respect to the actor network are computed through back-propagation through the critic and then the actor network.

E. TD3

Although DDPG has good performance in some cases, it also has a drawback which is Q-function sometimes begins to dramatically overestimate Q-values. It will exploits the errors in the Q-function and lead to the policy breaking. And there are three improvements that Twin Delayed DDPG (TD3) [4] has made to address the issue. The first one is TD3 uses two twin Q-functions to learn, and the smaller of the two Q-values is used to form the targets in the Bellman error loss functions. Unlike the targets in DDPG, here we have $y_t = r_t + \gamma \min_{i=1,2} Q_{\theta_i}(s_{t+1}, \pi_{\phi_i}(s_{t+1}))$. The second improvement is that TD3 updates the policy for every two Q-function. The third idea is by smoothing out Q along changes in action, TD3 adds noise to the target action, making it more difficult for the policy to exploit Q-function errors. We refer to [4] for more in-depth discussion about these improvements.

F. Hindsight Experience Replay (HER)

Let us consider the FetchSlide-v1 environment. This is a task where the robot needs to learn how to push and slide the puck on the table to hit the target. With the initial random exploration, the robot is unlikely to succeed in accomplishing the goal. In prior reinforcement learning algorithms, the sparse binary reward means that only a -1 is rewarded at all timesteps during these episodes with failed goals, and so the agent does not learn anything from this negative learning signal.

The main idea of Hindsight Experience Replay (HER) [1] is to pretend we reached an alternative goal even if we failed the original task. Using this substitution, we can still get a learning signal from mistakes, rather than getting only -1 at each timestep. By relabeling these episodes as successes, the RL algorithm gets a learning signal that can be used to train the agent. HER can be combined with any off-policy RL algorithm, such as DDPG and TD3. In HER, we store both the original episode and the goal-substituted episode into the replay buffer. Having alternate goals helps the agent learn during the random exploration phase of RL training, while recording real goals helps the agent converge to a policy that actually solves the original environment.

G. Incorporating Demonstrations for Complex Environments

Previous work such as [9] have incorporated demonstrations (imitation learning) with reinforcement learning to guide agents into solving complex environments. The use of a demonstration buffer, a second replay buffer containing episodes recorded from human demonstrators, was introduced in [9]. Their work shows that demonstrations help short-circuit the random exploration phase of reinforcement learning, which may be prohibitively difficult in environments such as FetchSlide-v1, and multi-goal environments.

The work of [10] extends HER by incorporating demonstrations. They introduce two ideas that we use and summarize here: the behavior cloning loss L_{BC} and the Q-filter $L_{BC,QF}$:

$$L_{BC} = \sum_{i=1}^{N_D} ||\pi(s_i|\theta_\pi) - a_i||^2$$

$$L_{BC,QF} = \sum_{i=1}^{N_D} ||\pi(s_i|\theta_\pi) - a_i||^2 \mathbb{1}_{Q(s_i,a_i) > Q(s_i,pi(s_i))}$$

The behavior cloning loss is a standard loss used in imitation learning [10] for training the actor, as it encourages the agent to mirror the demonstrator policy. However, the demonstrator may have sub-optimal movements and we do not want to learn from this. The Q-filter solves this problem by only applying the behavior cloning loss where the critic $Q(s, a)$ determines that the demonstrator action would be better than the actor action. This behavior cloning loss with Q-filter is balanced with the appropriate RL algorithm loss (DDPG and TD3 in our case), so that the agent is able to explore and potentially learn better actions as well.

III. Methods

A. Overview

We first started with the OpenAI Baselines [11] implementation of HER, which uses DDPG as its off-policy RL algorithm. Together, this is referred to as DDPG+HER. Using DDPG+HER, we obtain baseline results using the computational hardware available to us. More experimental details can be found in Section VI.

B. TD3+HER

Our first method of improving on the baseline method was to substitute the DDPG algorithm with TD3. In this TD3+HER algorithm, we kept as many of the hyperparameters the same as possible to DDPG+HER in order to make a fair comparison of results.

C. Simulated Demonstrations

As suggested by [10], learning from demonstrations using their method could lead to an agent outperforming the demonstrator policy. In this second method of improving on the baseline method, we augment HER with learning from a demonstration buffer.

Unlike [10], we do not use data collected from real demonstrations. Instead, we assume the agents trained from the baseline DDPG+HER and TD3+HER are our expert demonstrators. During their training, we collect all successful test episodes and save them into a demonstration buffer. Episodes that are not successful are not included, and this is independent of the alternative goals from HER. Then we train a new agent using this demonstration buffer, HER, and the corresponding off-policy RL algorithm. In this method, we implement the same ideas a introduced in [10] in utilizing our demonstration buffer, namely the behavior cloning loss and the Q-filter.

When training an agent with demonstrations and DDPG+HER, we filled the demonstration buffer using episodes collected from test rollouts of an agent trained with only DDPG+HER. Correspondingly for training an agent with demonstrations and TD3+HER, the demonstration buffer was filled using an agent trained with only TD3+HER.

IV. Results

During initial experimentation, we attempted to reproduce the results of the baseline DDPG+HER algorithm on the four Fetch environments available to us [2], [3]. Due to our low computational budget compared to the authors', we trained with less MPI workers, meaning less environment timesteps per epoch. However, we were able to see trends similar to those shown in [3]. DDPG+HER reached 100% test success rate in our experiments as expected. (Although the authors compared sparse and dense reward settings, we only use the sparse rewards.) For FetchSlide-v1 and FetchPickAndPlace-v1, we got slightly lower results using DDPG+HER than in [3], which is expected because of the difference in MPI worker count. Initial results comparing DDPG+HER and TD3+HER on the Fetch environments are shown in Fig. 1 and Fig. 2. We also tested vanilla DDPG and TD3 on FetchSlide-v1 and FetchPickAndPlace-v1, but test success rates were stuck at 0. These unsuccessful runs are omitted from the plots.

Comparing the results in Fig. 1 about the FetchSlide-v1 environment, there does not seem to be any significant improvement switching from DDPG to TD3 as the RL al-

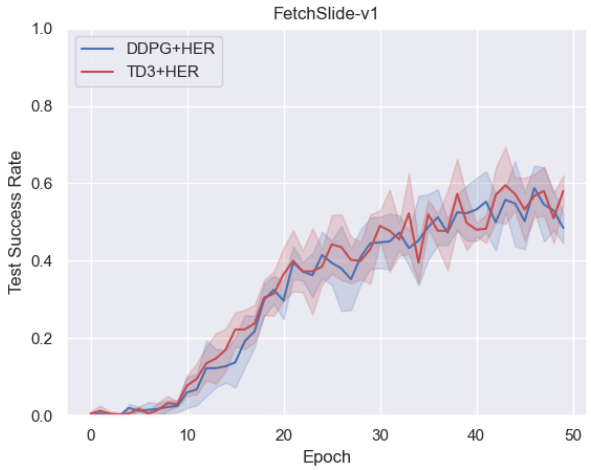

Fig. 1. Performance comparison of baseline HER algorithms on the FetchSlide-v1 environment.

Fig. 2. Performance comparison of baseline HER algorithms on the FetchPickAndPlace-v1 environment.

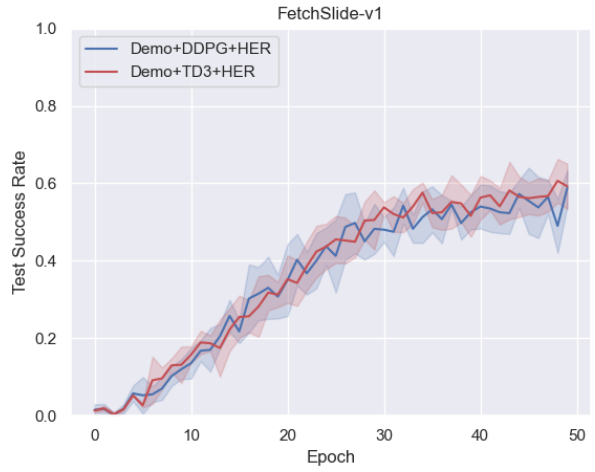

Fig. 3. Performance comparison of demonstration HER algorithms on the FetchSlide-v1 environment.

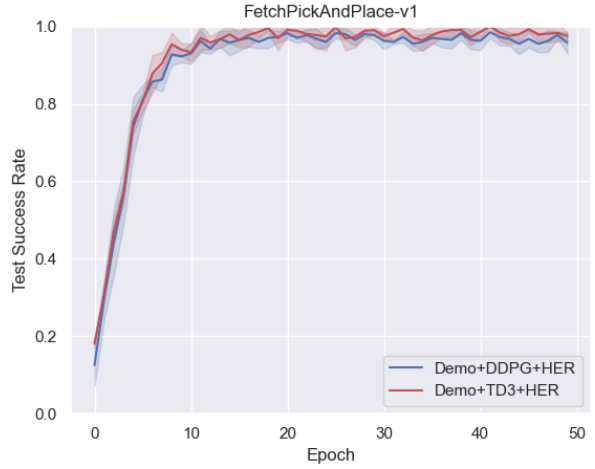

Fig. 4. Performance comparison of demonstration HER algorithms on the FetchPickAndPlace-v1 environment.

gorithm. Both algorithms improve at roughly the same rate, and reach roughly the same test success rate during training. Comparing the results in Fig. 1 about the FetchPickAndPlace-v1 environment however, TD3+HER shows clear improvement over DDPG+HER. Although the two algorithms had similar performance initially, TD3+HER improved more rapidly starting at Epoch 10 and reaches a higher test success rate by the end of training, on average. With DDPG+HER, we reach an average test success rate of roughly 0.8, and for TD3+HER a rate of roughly 0.9. This is on par with the previous ≈ 0.9 DDPG+HER result shown in Fig. 3 of [3], which was trained with many more MPI workers (and episodes).

Results from the demonstration variants of DDPG+HER and TD3+HER are shown in Fig. 3 and Fig. 4. On the FetchSlide-v1 environment, our method of using demonstrations marginally improves the test success rate of both algorithms compared to their baseline versions in Fig. 1 during the first 20 epochs. However, the algorithms all converge to roughly the same test success rate by epoch 50. Notably on the FetchPickAndPlace-v1 environment, Fig. 4, our method of using demonstrations greatly improves the test success rate compared to the baselines in Fig. 2. Within the first 10 epochs, we reach nearly a 100% test success rate. This is an improvement even over the previously-mentioned ≈ 0.9 DDPG+HER result of [3]. These improvements in the early epochs make sense given that the purpose of the demonstration method is to short-circuit the random exploration phase early in the learning.

Using the demonstrations and TD3+HER together, the trained agent outperforms the baseline DDPG+HER on the FetchPickAndPlace-v1 environment. Improvements are seen both in the rate of increase in the test success rate (learning speed), and in the final achieved test success rate. On the more complicated FetchSlide-v1 task, there are minimal improvements using the modified algorithms. There is only a slight improvement in learning speed.

V. Conclusion

Our results show that HER helps augment the performance of RL algorithms like DDPG and TD3, with TD3+HER performing better than DDPG+HER on the FetchPickAndPlace-v1 task. On the more difficult FetchSlide-v1 task, the two algorithms perform similarly.

Our results also show that incorporating demonstrations, even from a RL-trained agent and not from humans, can improve performance versus RL-alone or imitation learning alone. However since our demonstrations are simulated, this method may not extend well to even more complex environments. It still relies on the initial demonstrating agent to be able to solve the environment on its own first. HER naturally enables complex and multi-goal environments to be solved by off-policy RL algorithms, and using better RL algorithms or using demonstrations enables even better performance.

In our implementation of the demonstration buffer, we collect success episodes from all test runs of the DDPG+HER and TD3+HER. This includes runs in early epochs where, although the robot successfully reaches the goal in the episode, the robot may move sub-optimally or noisily. In future work, we would like to try only recording successful episodes after the agent has converged, or after the 50 training epochs. Perhaps these new demonstrations are more informative in training new agents, even accounting for the Q-filter in the behavior cloning loss.

Given more time and resources, we would also like to train our algorithm using the exact same hyperparameters as in the original HER paper [1], and compare our results to theirs directly. Similarly, we would train the algorithm for the *Shadow Dexterous Hand* environments [3], which have also been shown to be difficult to solve for vanilla RL algorithms but easier when they are combined with HER. In future work, we would also like to explore using even more advanced off-policy RL algorithms. Furthermore, we would like to examine

how well our trained agents generalize to a real-life Fetch robots performing similar tasks.

REFERENCES

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2018.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research, 2018.

[4] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[5] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE, 2012.

[6] Inc. Fetch Robotics. Autonomous mobile robots that improve productivity - fetch robotics. https://fetchrobotics.com/.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[8] 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, 2019.

[9] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards, 2018.

[10] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations, 2018.

[11] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

VI. Experiment Details

A. Hyperparameters

We use similar hyper parameters as in [1], [3] in training all of our agents, with the one difference indicated in bold. The lower MPI worker count was set according to the authors' available computational hardware.

- Actor and critic networks: 3 layers with 256 units each and ReLU non-linearities
- Adam optimizer [12] with 10^{-3} learning rate for training both actor and critic
- Buffer size: 10^6 transitions
- Polyak-averaging coefficient: 0.95
- Action L2 norm coefficient: 1.0
- Observation clipping: $[-200, 200]$
- Batch size: 256
- Rollouts per MPI worker: 2
- **Number of MPI workers:** 4
- Cycles per epoch: 50
- Batches per cycle: 40
- Test rollouts per epoch: 10
- Probability of random actions: 0.3
- Scale of additive Gaussian noise: 0.2
- Probability of HER experience replay: 0.8
- Normalized clipping: $[-5, 5]$

Using this configuration, each experiment on the FetchSlide-v1 and FetchPickAndPlace-v1 environments trains with 250,000 timesteps. This corresponds to 50 epochs.

B. Training & Testing

We use the same training and testing procedure as described in [1], using the OpenAI Baseline [11] code as a starting point.

During training, each epoch consists of 50 cycles of MPI worker rollouts. Each worker has 2 rollouts. After generating and storing the episodes in the replay buffer, the networks are trained with 40 batches sampled from the replay buffer. For training with demonstrations, 128 of the 256 samples in batch are replaced with samples from the demonstration buffer instead. Also for demonstrations, we weigh the primary RL training loss equally with the secondary Behavior Cloning loss with Q-filter, adding them together directly before backpropagation.

Each MPI worker performs 10 deterministic test rollouts, and the test success rate is averaged from all of these rollouts. In our case, the test success rate is calculated from the number of successful episodes from 40 total episodes.