Goal-conditioned Imitation Learning (goalGAIL) Replication

Anonymous

Abstract

Reinforcement learning algorithms coupled with deep learning techniques have achieved many successes in solving robotic tasks. In this paper, we present a study of reproducing Goal-conditioned Imitation Learning (goalGAIL) [1], an algorithm that learns to reach any desired goals from a few expert demonstrations using binary rewards and self-supervision.

1 Introduction

Reinforcement Learning (RL) has shown impressive performance in diverse simulated scenarios, while it still struggles to transfer its successes in simulation to real world tasks. One of the critical assumptions that are hard to obtain in the real world are the access to a dense reward function. Self-supervised methods help mitigate the problem of having access to only binary reward signals.

Learning from Demonstrations, or Imitation Learning (IL), is such an alternative to reward crafting that learns desired behaviors from human demonstrations. Two main methods for IL are Behavioral Cloning (BC) and Inverse Reinforcement Learning (IRL). However, both methods suffer in the low data regime which makes data relabeling an interesting topic to explore. In addition, we often expect the agent to be able to reach a wide range of configurations without training a specific policy for each one, which reveals the importance of goal-conditioned setting during the training process.

The GoalGAIL paper[1] aims to find a goal-conditioned policy that seeks to obtain the indicator reward of having the observation exactly match the goal under self-supervision. It first describes how IL methods can be extended to the goal-conditioned setting, and utilizes Hindsight Experience Replay (HER) for data augmentation. Then, the paper illustrates and analyzes goalGAIL algorithm, empirically proving its outstanding performance and robustness over the baseline methods. Finally, it shows the model's ability to leverage state-only demonstrations that do not include expert actions.

2 Background

2.1 Imitation Learning (IL)

Imitation learning (IL) techniques aim to mimic expert behavior in a given task. A RL agent is trained to perform a task from demonstrations by learning a mapping from observations to actions. Generic imitation learning methods could potentially reduce the problem of teaching a task to that of providing demonstrations; without the need for explicit programming or designing reward functions specific to the task [2].

The problem of imitation learning can be formalized as the followings. An environment is described by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, the horizon H, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, a transition probabilities distribution $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_+$, and a discount factor $\gamma \in [0, 1]$.

Then we define a discrete-time finite-horizon discounted Markov decision process (MDP) by a tuple $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho_0, \gamma, H)$. Our objective is to find a stochastic policy π_θ that maximizes the expected discounted reward within the MDP:

$$\eta(\pi_\theta) = \mathbb{E}_\tau[\Sigma_{t=0}^{T} \gamma^t r(s_t, a_t, s_{t+1})] \tag{1}$$

We denote by $\tau = (s_0, a_0, ...,)$ an entire state-action trajectory, where:

$$s_0 \sim \rho_0(s_0), a_t \sim \pi_\theta(a_t|s_t), s_{t+1} \sim \mathcal{P}(s_{t+1}|s_t, a_t)$$

We also assume that expert demonstrations are provided as a dataset of state-action pairs $\mathcal{D} = \{(s_i, a_i)\}_{i=1}^{N}$. [3]

Some methods, like Behavioral Cloning (BC) [4], train a parameterized policy $\pi_\theta : \mathcal{S} \to \mathcal{A}$ to maximize the likelihood of the given human demonstrations (expert actions), such that $\theta^* = \arg\max_\theta \Sigma_N \log \pi_\theta(a_i|s_i)$. While those methods' performance largely depend on the quantity of the demonstrations [5], which is costly and time-consuming for robotic tasks. In more general imitation learning, loss function takes in an expert policy generated by the given demonstrations rather than the demonstrations set during the learning process. Following the more general path, goalGAIL leverages the discriminator of Generative Adversarial Imitation Learning (GAIL) [6] for self-supervision to improve sample efficiency.

2.2 Related Work

2.2.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradients (DDPG) [7] is a model-free RL algorithm for continuous action space. In DDPG we maintain two neural networks representing *a target policy* (also called an actor) $\pi : \mathcal{S} \to \mathcal{A}$ and an action-value function approximator (called the critic) $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The critic's job is to approximate the actor's action-value function Q^π.

Episodes are generated using a behavioral policy which is a noisy version of the target policy, e.g. $\pi_b(s) = \pi(s) + \mathcal{N}(0, 1)$. The critic is trained in a similar way as the Q-function in Deep-Q Networks (DQN), which maintains a neural network Q that approximates Q^*. Q is defined as an ϵ-greedy policy with probability ϵ takes a random action sampled uniformly from $\mathcal{A}$ and takes the action $\pi_Q(s) = \arg\max_{a\in\mathcal{A}} Q(s, a)$ with probability $1 - \epsilon$. For formal definition and more details about Deep-Q Networks, see [8]. The main difference here is targets y_t are computed using actions outputted by the actor, i.e. $y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}))$. The actor is trained with mini-batch gradient descent on the loss $\mathcal{L}_a = -\mathbb{E}_s Q(s, \pi(s))$, where s is sampled from the replay buffer. The gradient of actor parameters w.r.t. $\mathcal{L}_a$ can be computed by backpropagation through the combined critic and actor networks. [9]

2.2.2 Hindsight Experience Replay (HER)

Hindsight Experience Replay (HER) [9] is a replay technique used when learning with sparse and binary rewards in order to improve data efficiency. The idea behind HER is very simple: after experiencing some episodes $s_0, s_1, ..., s_T$ we store in the replay buffer every transition $s_t \to s_{t+1}$ not only with the original goal used for this episode but also with a subset of other goals. Note that only the agent's actions are influenced but not the environment, so that each trajectory with an arbitrary goal can be used for replay when we use an off-policy RL algorithm. In this work, we use DDPG.

Follow the approach from *Universal Value Function Approximators* [10] we train policies and value functions which take as input not only a state s but also a goal $g \in \mathcal{G}$. We assume that every goal $g \in \mathcal{G}$ corresponds to some predicate $f_g(s) = \mathcal{S} \to \{0, 1\}$ and that the agent's goal is to achieve any state s that satisfies $f_g(s) = 1$. When we want to exactly specify the desired states in the system, we can define $\mathcal{S}$ and $f_g(s)$ accordingly.

Given an off-policy RL algorithm $\mathbb{A}$, a strategy $\mathbb{S}$ for sampling goals for replay, a reward function $r : \mathcal{S}\times\mathcal{A}\times\mathcal{G} \to \mathbb{R}$, for every timestep in every episode, HER stores the transition $(s_t \parallel g, a_t, r_t, s_{t+1} \parallel g)$ in the replay buffer R, samples a set of additional goals for replay $G := \mathbb{S}(\textbf{current episode})$, and then for every $g' \in G$, stores the transition $(s_t \parallel g', a_t, r'_t, s_{t+1} \parallel g')$ in R. Specifically, $r_t := r(s_t, a_t, g)$ and $r'_t := r(s_t, a_t, g')$.

Without having any control over the distribution of initial environment states, HER provides additional training data by replaying transitions stored in the replay buffer R. An off-policy RL algorithm like DDPG can be trained with these data to efficiently overcome sparse and uninformative rewards or data insufficiency. For more details about HER, please refer to [9].

2.2.3 Generative Adversarial Imitation Learning (GAIL)

With the same formalized imitation learning preliminaries in Section 2.1, Generative Adversarial Imitation Learning (GAIL) [6] uses an expectation with respect to a policy $\pi \in \Pi$ to denote an expectation with respect to the trajectory it generates: $\mathbb{E}_\pi[r(s,a)] \triangleq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$ where Π is the set of all stationary stochastic policies that take actions in $\mathcal{A}$ given states in $\mathcal{S}$. $\widehat{\mathbb{E}}_\tau$ denotes empirical expectation with respect to trajectory samples τ, and π_E denotes the expert policy.

Adopted from maximum causal entropy Inverse Reinforcement Learning (IRL) [11, 12], given an expert policy π_E, the optimization problem can be defined as:

$$\max_{c \in \mathcal{C}} \left(\min_{\pi \in \Pi} -H(\pi) + \mathbb{E}_\pi[c(s,a)] \right) - \mathbb{E}_{\pi_E}[c(s,a)] \tag{2}$$

where $H(\pi) \triangleq \mathbb{E}_\pi\left[-\log \pi(a|s)\right]$ is the γ-discounted causal entropy of the policy π and $c \in \mathcal{C}$ is the cost function from a set of all possible cost functions. Maximum causal entropy IRL looks for a cost function $c \in \mathcal{C}$ that assigns low cost to the expert policy and high cost to other policies, thereby allowing the expert policy to be found via a certain reinforcement learning procedure:

$$RL(c) = \arg\min_{\pi \in \Pi} - H(\pi) + \mathbb{E}_\pi[c(s,a)] \tag{3}$$

which maps a cost function to high-entropy policies that minimize the expected cumulative cost. Notify that in practice, π_E will only be provided as a set of trajectories sampled by executing π_E in the environment, so the expected cost of π_E in Eq.2 is estimated using these samples.

In GAIL, a discriminator D_ψ is trained to distinguish expert transitions $(s,a) \sim \tau_{expert}$ from agent transitions $(s,a) \sim \tau_{agent}$, while the agent is trained to "fool" the discriminator into thinking itself is the expert. Formally, the discriminator is trained to minimize:

$$\mathcal{L}_{GAIL} = \mathbb{E}_{(s,a)\sim\tau_{agent}}\left[\log D_\psi(s,a)\right] + \mathbb{E}_{(s,a)\sim\tau_{expert}}\left[\log(1 - D_\psi(s,a))\right] \tag{4}$$

while the agent is trained to maximize $\mathbb{E}_{(s,a)\sim\tau_{agent}}\left[\log D_\psi(s,a)\right]$ by using the output of the discriminator $\log D_\psi$ as reward. For more details about GAIL, please refer to [6].

3 GoalGAIL Algorithms

3.1 Preliminaries

GoalGAIL solves the same MDP described in Section 2.1 with changes specific to goal-conditioned tasks, where the deterministic policy and the binary reward function are now conditioned on a goal sampled uniformly from all feasible states, i.e. $a_t \sim \pi_\theta(a_t|s_t, g)$, $r(s_t, a_t, s_{t+1}|g) = \mathbb{1}[s_{t+1} == g]$. Given that the transition probability is not affected by the goal, g can be "relabeled" in hindsight, so a transition $(s_t, a_t, s_{t+1}, g, r = 0)$ can be treated as $(s_t, a_t, s_{t+1}, g' = s_{t+1}, r = 1)$. [1]

To solve the learning objective in Eq. 1 in goal-conditioned settings, goalGAIL extends GAIL [6] to be conditioned on the goal and combines it with HER. [9] Therefore, the loss described in Eq. 4 becomes:

$$\mathcal{L}_{GAIL}(D_\psi, \mathcal{D}, \mathcal{R}) = \mathbb{E}_{(s,a,g)\sim\mathcal{R}}\left[\log D_\psi(s,a,g)\right] + \mathbb{E}_{(s,a,g)\sim\mathcal{D}}\left[\log(1 - D_\psi(s,a,g))\right] \tag{5}$$

One major contribution of goalGAIL is the use of GAIL discriminator D_ψ as an additional reward term so the reward function used to train an off-policy RL algorithm, in our case DDPG, is a weighted sum of the binary reward provided by the environment and the discriminator loss function, i.e. $r(s_t, a_t, s_{t+1}) = (1 - \delta_{gail})\mathbb{1}[s_{t+1} == g] + \delta_{gail} \log D_\psi(s_t, a_t, g)$.

3.2 Data Generation Algorithms

By inspecting the released code, we find that the policy used to generate expert demonstrations is greedy, i.e. it chooses the shortest path in the state space to reach a desired goal.

To sufficiently use both the agent and expert's trajectories, goalGAIL employs HER for data augmentation, which is described in Section 2.2.2. In the goalGAIL algorithm, HER is used to relabel all agent's transitions and selected expert's transitions, sampled from their corresponding replay buffers.

As a result, goalGAIL learns to improve its policy using two sets of trajectories, the set of relabelled expert demonstrations and the set of relabelled trajectories from executing its own policy.

3.3 GoalGAIL Algorithm

The goalGAIL algorithm consists of a loop of the following steps, the DDPG actor first executes its policy and stores the obtained trajectories $(s_0, a_0, s_1, \dots |g)$ in a replay buffer. Then goalGAIL samples a minibatch of agent and expert's trajectories from their corresponding buffers, and relabel the agent trajectories and optionally the expert trajectories. It trains the GAIL discriminator using the minibatch. The DDPG critic is trained using a weighted sum of discriminator loss and the binary reward function. GoalGAIL then does a gradient update step on DDPG's policy network.

The improvement achieved by goalGAIL is three-fold,

- goalGAIL accelerates learning and improves performance by leveraging GAIL's discriminator for self-supervision and the relabeling technique HER.
- goalGAIL is more robust to noise in the expert demonstrations and can outperform the demonstrator.
- goalGAIL can leverage state-only demonstrations for learning.

Our replication efforts focus on demonstrating the first result, i.e. goalGAIL achieves a better policy than GAIL and converges to a similar optimal performance but faster than HER with DDPG.

4 Implementation Details

4.1 Contributions

The major components of goalGAIL algorithm are HER, DDPG and GAIL. Combining the strengths of HER and GAIL, we implement our system as shown in Figure 1. Note that many public repositories contain full or partial implementations of these three algorithms, e.g. OpenAI baselines.[1] In this work, we uses a customized version of HER from OpenAI, GAIL from Google research together with other public libraries such as OpenAI gym and rllab.[2]

The high-level idea of our implementation design is that we create three parameter dictionaries, which are DDPG parameter dictionary, GAIL parameter dictionary and expert buffer parameter dictionary. Each dictionary stores initialization parameters of its corresponding component. The GAIL discriminator, which is in fact a fully connected neuron network with two hidden layer of 256 units, will measure the distinction of observations, goals and actions between agent and expert. The DDPG policy, unlike the normal one that just gains the immediate reward from reaching successive state, will interpolate the reward that related to the similarity of action between its policy and expert policy. The expert policy will generate demonstrations and store them in the expert replay buffer prior to the training process. Those demonstrations will later be sampled from the replay buffer to train the GAIL discriminator. The discriminator will then compute a reward and store it in the agent replay buffer which will eventually be used to update the agent policy.

As these models are all highly separable and elastic, the training of different algorithms can be simply adjusted by different hyperparameters. We can switch to run just HER with DDPG by simply removing the reward given by the GAIL discriminator, or we can ignore the transitions relabeled by HER in order to run GAIL only.

4.2 Observations

The authors imported and modified several baselines from other public repositories so that different parts can be better combined and easily invoked. In GAIL, the reward function was modified from its originated form $-\log(1 - sigmoid(x))$ to $\log(sigmoid(x))$ which changes the backbone function from a convax function to a concave function. It could be a factor which makes a difference in the final results since it changes the probability distribution and expectation of the reward, however, this is not mentioned in the paper.

[1]`https://github.com/openai/baselines`.

[2]OpenAI gym: `https://github.com/openai/gym`, rllab: `https://github.com/rll/rllab`.

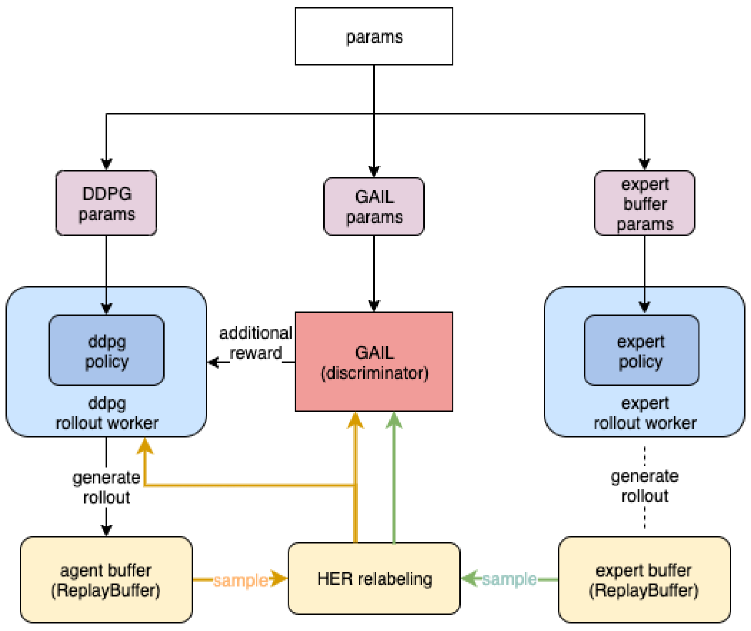
4.3 Hyperparameters

In the four tasks in our experiments, i.e. Continuous Four Rooms, Pointmass Block Pusher, Fetch Pick & Place and Fetch Stack Two, we set the same hyperparameters as those reported in the paper.

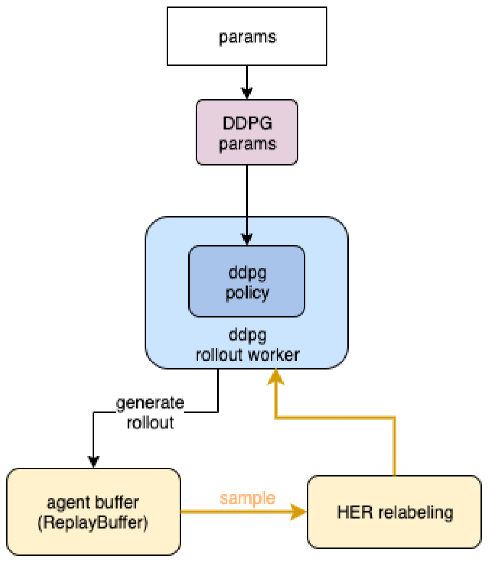
5 Experiments

5.1 Environment Setup

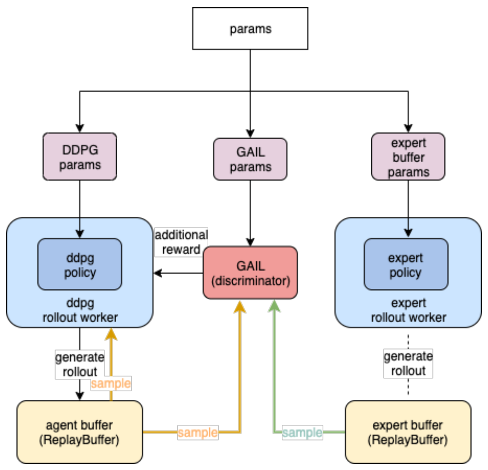
Though the author offered an environment configuration file in their repository, we failed to build a compatible virtual environment directly from it using miniconda-3 on Ubuntu 16.04. We observed



(a) GoalGAIL



(b) HER



(c) GAIL

Figure 1: Flowchart of HER, GAIL and GoalGAIL

multiple deprecated packages (i.e. rllab) and several conflicts among packages. To use the same environment setup as provided in the paper for experiments, we reconfigured our own virtual environment manually. A new conda configuration file works for miniconda-3 on Ubuntu 16.04 can be found in our repository.

5.2 Reproducibility Cost

Given that there was no mention in the paper about compute requirement, we describe the computational cost required for running the experiments. In order to keep the training process invariant for comparison, we run experiments with close number of steps as shown by the authors.

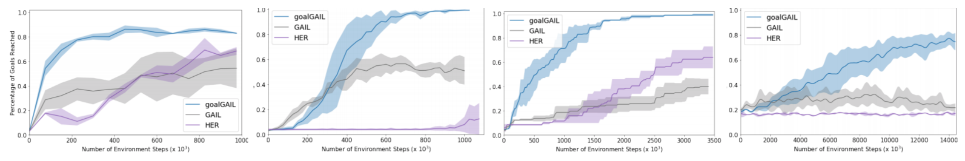
To complete all the experiments, we create 12 instances (Machine Type: n2-standard-2, with 4 vCPUs, 8GB memory, and 40GB disk size) on Google Cloud Platform for these 12 lines. The first two tasks, i.e. Continuous Four Room and Pointmass Block Pusher, would take around 20 hours for each line, including baselines and goalGAIL. The Fetch Pick & Place task, would take around 40 hours for each line, including baselines and goalGAIL. The Fetch Stack Two, would take around 100 hours for each line, including baselines and goalGAIL. In our case, the computing time is about 540 hours in total.

5.3 Results

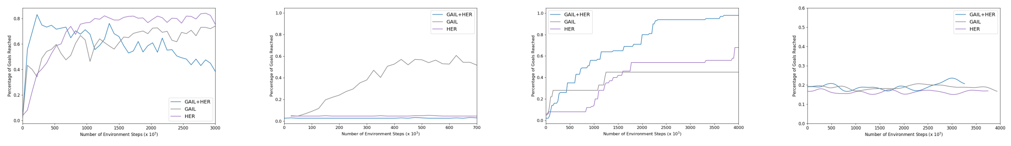
We choose to replicate Figure 3 and think it captures one essence of the algorithm, i.e. goalGAIL achieves a better policy than GAIL and converges to a similar optimal policy but faster than HER with DDPG.

As shown in Figure 3 from the original paper, we run three different algorithms, HER with DDPG, GAIL and goalGAIL in four simulated environments. For each environment, we compare and analyze the results reported in the paper, reproduced by the authors' code, and generated by our reimplementation.

We observe 3 random seeds being used with one standard deviation to generate the graphs reported in the paper. However, with limited computing resources and time, we only complete the experiments with one of the three original random seeds.



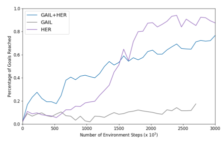
(a) Continuous Four rooms (b) Pointmass block pusher (c) Fetch Pick & Place (d) Fetch Stack Two



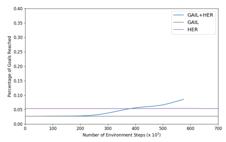
(a) Continuous Four Rooms (b) Pointmass Block Pusher (c) Fetch Pick & Place (d) Fetch Stack Two

Figure 2: Plots generated by running author's code

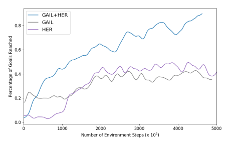
For the Continuous Four Rooms task, we run author's code, shown in 2(a), and our implementation, shown in 3(a), respectively for 3×10^6 steps. From 2(a), we can see that the early growth of goalGAIL is very fast, but after 10^6 steps, the performance of goalGAIL starts to decline, while HER rises steadily and finally converge to a policy with the similar performance of goalGAIL. We observe that the reported diagram only shows the result within 10^6 steps. The results from our implementation 3(a) show consistency with the results shown in the paper before 10^6 steps, however, the goal-reaching percentage of HER rises quickly between 10^6 to 2×10^6 steps and outperforms goalGAIL in our experiments at the end. From our perspective, the variance in Continuous Four Rooms task might be



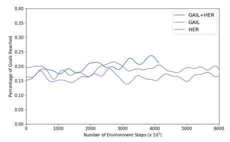
(a) Continuous Four Rooms



(b) Pointmass Block Pusher



(c) Fetch Pick & Place



(d) Fetch Stack Two

Figure 3: Plots generated by running our code

caused by the import of tensorflow and numpy. The authors imported them in caller environment, which may disable the effect of random seed.

For the Pointmass Block Pusher task, we run author's code shown in 2(b), and our implementation, shown in 3(b) respectively for 7×10^5 steps. However, the figure we get is apparently different from the author's. We are assuming that this might be caused by the reason that we contaminated the author's code during the initial setup phase. Our implementation is also different from the paper. Neither goalGAIL nor GAIL learns the policy well.

For the Fetch Pick & Place task, we run author's code shown in 2(c) and our implementation shown in 3(c) for 5×10^6 steps. The trend of all three algorithms in our implementation is similar to the figure presented in the paper. GoalGAIL learns more quickly than the other two baselines and converges faster. The plots agree with author's claim that goalGAIL is able to outperform HER and GAIL for Fetch Pick & Place task. One thing needs to notify is that the authors' code only logs the maximum success rate over epoches, and we takes the actual success rate, which makes their curves to increase steadily during training and also results in the slight fluctuations in our curves. The only difference between the figures is that the author's code actually logs the maximum success rate, which makes their lines rise in steps.

For the Fetch Stack Two task, we run author's code shown in 2(d) and our implementation shown in 3(d). Due to the limited time and computing resources, we only run both experiments with 4×10^6 steps. The plots from our implementation agree with 2(d) within the first 4×10^6 steps. Since we do not get as much data as the author, it is hard to tell which algorithm performs better in current condition.

6 Conclusion and Future Work

Based on our existing data, We could partially confirm the results reported in the original paper. Three tasks, Continuous Four Room, Fetch Pick & Place and Fetch Stack Two, basically conform to the author's results under the current condition. It is worth mentioning that in the Continuous Four Room task, the author only showed the first 10^6 steps, but we find that after 3×10^6 steps, the performance of goalGAIL experiences a significant decline, which the author did not mention in the paper. For Pointmass Block Pusher task, the algorithms perform much worse. We will first focus on further testing the author's code to ensure whether the figures match each other. Meanwhile, we will look into our own implementation and determine the reasons for its poor performance. Also, we will run Fetch Stack Two task for more steps and **obverse** whether the future trend matches author's conclusions for this task.

In the future work it will be interesting to validate this goal-conditioned method in real world settings, expand it to more robotic domains (e.g. drone navigation), or to integrate with natural language instructions.

Acknowledgement

We would like to express our gratitude to the authors for releasing their code and provide us with the opportunity to learn from experimenting with the code base.

References

[1] Yiming Ding, Carlos Florensa, Pieter Abbeel, and Mariano Phielipp. Goal-conditioned imitation learning. In *Advances in Neural Information Processing Systems*, pages 15298–15309, 2019.

[2] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):21, 2017.

[3] Yuke Zhu, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, et al. Reinforcement and imitation learning for diverse visuomotor skills. *arXiv preprint arXiv:1802.09564*, 2018.

[4] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[5] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[6] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016.

[7] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[9] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[10] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015.

[11] Brian D Ziebart, Andrew Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. 2008.

[12] Brian D Ziebart, J Andrew Bagnell, and Anind K Dey. Modeling interaction via the principle of maximum causal entropy. 2010.