ICLR 2019 Reproducibility-Challenge Discriminator-Actor-Critic: Addressing Sample Inefficiency and Reward Bias in Adversarial Imitation Learning

Sheldon Benard, Vincent Luczkow, & Samin Yeasar Arnob

COMP-652 Machine Learning
McGill University
Montreal, Quebec, Canada
{sheldon.benard,vincent.luczkow,samin.arnob}@mail.mcgill.ca

Abstract

As part of the ICLR 2019 Reproducibility Challenge we attempted to replicate the results of *Discriminator-Actor-Critic: Addressing Sample Inefficiency and Reward Bias in Inverse Reinforcement Learning* (Anonymous, 2019). Discriminator-Actor-Critic (DAC) is an adversarial imitation learning algorithm. It uses an off-policy reinforcement learning algorithm to improve upon the sample efficiency of existing methods, and it extends the learning environment with absorbing states and uses a new reward function in order to achieve unbiased rewards. We were able to reproduce some but not all of the claimed results, achieving comparable rewards and sample efficiency on two of four environments. All of our code is available at: **https://github.com/vluzko/dac-iclr-reproducibility**

1 Introduction

Within reinforcement learning, imitation learning is the problem of training a policy that imitates an existing expert policy, rather than training a policy directly. In the adversarial imitation learning framework, a discriminator network is used to distinguish expert behaviour from learner behaviour, and the learner policy is rewarded for fooling this discriminator. In the DAC paper, the authors note two problems with existing adversarial imitation learning algorithms (with the generative adversarial imitation learning (GAIL) algorithm (Ho & Ermon, 2016) being a pivotal example).

First, the algorithms have low sample efficiency, requiring a very large number of environmental interactions in order to reach convergence. This is undesirable, since interactions can be expensive. To mitigate this issue, DAC utilizes the TD3 off-policy learning algorithm (Fujimoto et al., 2018) to train the policy; this significantly increases sample efficiency.

Second, the reward functions used by these algorithms are implicitly biased either for or against survival. As such, the policy may not be able to correctly learn the expert's policy, since the bias could incentivize detracting behaviours. DAC modifies the reward function to be able to return both positive and negative rewards, and converts all terminal states in the environment to absorbing states. Combined, these changes remove the survival bias.

To evaluate their algorithm, the authors give empirical demonstrations of DAC's increased performance on five MuJoCo environments and a virtual Kuka IIWA arm. They show that, under DAC, the learner policy converges much faster when compared to baseline models. For baseline benchmarks, the authors use the behavioral cloning algorithm (Bain & Sammut, 1995) and the GAIL algorithm.

In order to reproduce the DAC algorithm, we:

- Used the original TD3 algorithm to train the policy
- Implemented the GAIL binary discriminator and modified the reward function
- Added a gradient penalty and entropy regularization to the discriminator and gradient clipping to the actor network
- Altered the Markov decision process by replacing existing terminal states with absorbing states, while incorporating importance weights.

We tested our implementation on four of the five MuJoCo environments used in the original paper, and we were able to achieve comparable results on two of them. In these experiments, we considered two separate loss functions for the discriminator: the authors‘ original loss function, which does not average the expert and policy discriminator loss, and the loss function which averages these losses (see 3.3.1). Additionally, we tested the authors' auxiliary claim that their new reward function performs well with a naive discriminator. To this, our learning curve failed to retain the same reward trajectory, with the original GAIL reward function performing more favorably.

2 Background

In the standard reinforcement learning paradigm, a policy selects actions, within a particular environment, over time. These actions determine how the environment changes, and the environmental changes determine a reward given to the policy. The goal is to create a policy which achieves the maximum reward.

Formally, the underlying environment is a *Markov decision process* $(S, A, P, \rho, r, \gamma)$. S is the set of states the environment can be in (for instance it could be positions of pieces on a chess board). A is the set of actions a policy can take. $P : (S \times A) \to S$ is a function that maps the current state and the action selected by the policy to the next state. ρ is a distribution over S that determines the initial state of the environment. $r : (S \times A \times S) \to \mathbb{R}$ is the reward function, which takes the current state, the current action, and the next state and gives the policy a reward. γ is a discount factor, used to reduce the value of future rewards. (Specifically the reward at time step i is actually $\gamma^i r(s_i, a_i, s_{i+1})$, not just $r(s_i, a_i, s_{i+1})$.) A policy $\pi : S \to A$ is a function that takes the current state as input and produces an action (frequently the policy will actually be stochastic).

Imitation learning is an extension of the standard reinforcement learning problem. Given a Markov decision process (S, A, P, r, γ) *and* an expert policy $\pi_E : S \to A$, the goal is to produce a learner policy $\pi_L : S \to A$ which is as close as possible to π_E. This is in contrast to standard reinforcement learning, where the goal is simply to produce a policy that achieves high reward on the MDP.

Adversarial imitation learning is an imitation learning framework in which a discriminator network is trained to distinguish between actions taken by the learner policy and actions taken by expert policy. The reward function used to train the learner is then a measure of the learner's ability to fool the discriminator. The discriminator and learner are then trained in alternate steps until the learner consistently fools the discriminator into believing it is the expert.

3 Discriminator-Actor-Critic: Technical Details

3.1 Motivation: Issues with Adversarial Imitation Learning

3.1.1 Sample Inefficiency

In GAIL, an agent requires as few as 200 expert transitions from 4 expert trajectories in order to robustly imitate the expert and achieve expert-like trajectories and rewards. However, in order to reach this point the agent requires as many as 25 million policy transitions sampled from the environment to reach convergence. This is sample inefficient and isn't suitable for many real-world applications.

3.1.2 Reward Bias

Adversarial algorithms often utilize the reward functions $r(s,a) = -\log(1 - D(s,a))$ or $r(s,a) = \log(D(s,a))$. The former assigns only positive rewards. This is great for environments with a survival bonus, since the per-step positive reward encourages the agent to survive longer and collect additional rewards. However, if the task is time-sensitive and must be done quickly, the agent can behave sub-optimally, as the positive reward may incentivize the agent to remain close to the expert but move in loops or take a longer path. The latter, in contrast, assigns only negative rewards. The reward function assigns per-step penalties, which encourages shorter paths, but is not suitable for tasks with survival bonuses. As such, a new reward function is needed to generalize the imitation learning paradigm across tasks.

Furthermore, adversarial methods improperly handle terminal states. This introduces implicit reward priors that can affect the policy‘s performance. In particular, many imitation learning implementations and MDPs omit absorbing states s_a. Thus, they implicitly assign 0 reward to terminal/absorbing states and bias the reward learning process. If the reward formulation for a given task is $r(s,a) = -\log(1 - D(s,a))$, an agent is never incentivized to explore an absorbing state. The strictly positive reward function rewards the policy for avoiding absorbing states, regardless of how the discriminator classifies the state-action pairs. Conversely, a strictly negative reward pushes the policy to end the trajectory as soon as possible, exploring absorbing states that the expert may never visit.

3.2 DAC Implementation

During training, all learner policy actions are added to a replay buffer R, while expert policy actions are placed in a replay buffer R_E. The algorithm is trained on samples from these replay buffers, rather than the exact trajectories of either policy.

To address the reward bias, the DAC algorithm utilizes a new reward function, $r(s,a) = log(D(s,a)) - log(1 - D(s,a))$, and converts terminal states to absorbing states. All terminal state-action-state tuples $(s_T, \cdot, \cdot)$ are replaced with a transition (s_T, a_a, s_a) to an absorbing state (s_a, a_a, s_a). This absorbing state is then appended to the expert rollout and the policy rollout[1]. Now, when training the discriminator and the policy, sampling from the replay buffer R may yield tuples involving the absorbing state, and the discriminator is able to learn whether reaching an absorbing state is desirable from the expert‘s perspective.

To increase sample-efficiency, the authors use off-policy methods to train the policy and the discriminator. The discriminator (which the authors specify is the GAIL discriminator) undergoes off-policy training via sampling from the replay buffer R. TD3 is then used to train the learner policy (Fujimoto et al., 2018), using the reward function given in the previous paragraph.

The GAIL discriminator is a 2-layer multilayer perceptron (MLP) with 100 hidden units per layer and $tanh$ activation functions. Gradient penalties (Gulrajani et al., 2017) regularize the network with $\lambda = 10$. The

[1] s_a is the absorbing (zero) state, s_T is terminal state, and a_a absorbing (zero) action

actor and critic networks are also 2-layer MLPs with 400 and 300 hidden units and $ReLU$ activations. Gradient clipping with a clipping value of 40 was used on the actor network.

All networks were trained with the Adam optimizer with initial learning rate 10^{-3}, which decayed by $\frac{1}{2}$ every 10^5 actor network training steps. Furthermore, as noted by the authors in their OpenReview comments and communications with us (Appendix B), the replay buffer R keeps all transitions, the batch size is 100, and importance weights were used when subsampling expert trajectories.

3.3 Our Implementation: Uncertainties and Differences

In order to implement the DAC algorithm, we followed the pseudocode provided in the paper‘s appendix and the description of the networks and the implementation found in Section 4 and Section 5. However, in our implementation and our experiments, we were met with and addressed some uncertainties.

3.3.1 Discriminator Loss Function

In the paper‘s pseudocode (see Appendix A), the discriminator's loss is:

$$L = \sum_{b=1}^{B} \log(D(s_{R,b}, a_{R,b})) - \log(1 - D(s_{E,b}, a_{E,b})) \tag{1}$$

The state-action tuples are samples from the replay buffer R and expert trajectory E. Here, the loss is minimized if *both* $D(s_E, a_E) = 0$ and $D(s_R, a_R) = 0$. We suspected this was an error, reached out to the authors (see Appendix B), and they confirmed that the subtraction should be an addition, giving the new loss function:

$$L = \sum_{b=1}^{B} \log(D(s_{R,b}, a_{R,b})) + \log(1 - D(s_{E,b}, a_{E,b})) \tag{2}$$

This loss function has the correct convergence properties, however we found that the loss explodes to $-\infty$ and NaNs begin to occur during training[2]. In OpenAI Baselines the loss function is instead binary cross entropy:

$$L = -[y_E \log(D(s_E, a_E)) + (1 - y_R)\log(1 - D(s_R, a_R))] \tag{3}$$

Where the target label $y_E = 1$ and the target label $y_R = 0$. Thus, L → 0, as $D(s_E, a_E) \rightarrow 1$ and $D(s_R, a_R) \rightarrow 0$.

Our results are generated with a regularized form of equation (3). The authors report (in the OpenReview comments) that the pseudocode omits entropy regularization (Ziebart et al., 2008), to simply the exposition. Including this term, the regularized loss function is:

$$L = -\sum_{b=1}^{B} [\log(D(s_{E,b}, a_{E,b})) + \log(1 - D(s_{R,b}, a_{R,b}))] + \lambda_{Entropy} H(\pi) \tag{4}$$

Where $\lambda_{Entropy} H(\pi)$ is entropy regularization with $\lambda_{Entropy} = 0.001$.

[2] Loss is minimized by pushing $D(a, b)$ towards 0 for learner samples and 1 for expert samples, thus we have $\log(D(a, b)) \rightarrow -\infty$ for learner samples and $\log(1 - D(a, b)) \rightarrow -\infty$ for expert samples.

Lastly, in OpenAI baselines[3], the GAIL discriminator takes the mean of the generator loss and expert loss. So, we also explored the loss function:

$$L = -\frac{1}{B}\sum_{b=1}^{B}[\log(D(s_{E,b}, a_{E,b})) + \log(1 - D(s_{R,b}, a_{R,b}))] + \lambda_{Entropy}H(\pi) \tag{5}$$

3.3.2 Decay

Two interpretations of the Adam optimizers learning rate decay were possible: (a) Decay the learning rate by $\frac{1}{2}$ for only the actor network, or (b) Decay the learning rate by $\frac{1}{2}$ for all networks. We used with the latter interpretation.

3.3.3 MuJoCo

For our experiments, we evaluated DAC in four environments: Hopper, Ant, Walker2d, and Half-Cheetah. However, we could not find which MuJoCo and Gym version the authors used. Since TD3 and OpenAI imitation (the original GAIL implementation) use *v1* (i.e. Hopper-v1), we evaluated the algorithm with this version of MuJoCo.

3.3.4 Sampling and importance weights

To generate the batch of expert state-action tuples from 4 expert trajectories, we subsampled 24 state-action pairs from each of the trajectories, with an absorbing state-action tuple appended to the end. An importance weight of 1 is assigned to expert tuples and $\frac{1}{N}$, $N = 25$, for absorbing tuples. These importance weights are used during the training of the discriminator.

3.3.5 Baselines

To gather the baseline results (GAIL baseline and Behavioral Cloning baseline), we utilized OpenAI baselines, whereas the authors used the original GAIL implementation (OpenAI imitation). However this does not affect the reported results for DAC, which used OpenAI imitation to generate the expert trajectories.

3.3.6 Evaluation

To evaluate the policy‘s training, we record the mean episode reward over 3 episodes utilizing the policy with no random noise. However, due to our limited time and computational power, we were only able to perform the evaluation for 1 seed (i.e 1 full run to 1 million environmental interactions per MuJoCo environment).

4 Experiments

All algorithms were implemented in the PyTorch framework (Paszke et al., 2017).

In order to implement the DAC algorithm, we used the original implementation[4] of TD3. Furthermore, we used a PyTorch implementation[5] of WGAN-GP for the discriminator gradient-penality, the original OpenAI

[3] https://github.com/openai/baselines

[4] https://github.com/sfujim/TD3

[5] https://github.com/EmilienDupont/wgan-gp

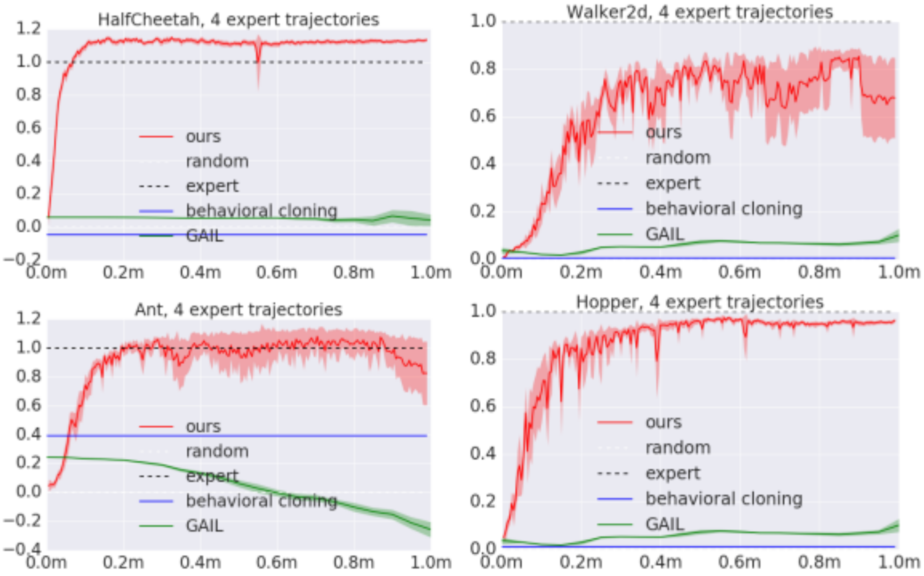


Figure 1: The MuJoCo results reported in the DAC paper.

GAIL code[6] to generate the expert trajectories for DAC, and OpenAI baselines[7] for the behavioural cloning and GAIL benchmarks.

4.1 DAC Experiments

Figure 1 depicts the original results obtained by the authors of the paper.

Referring to Figure 2, we were able to retrieve the expert reward in under 1 million timesteps for both the Ant-v1 and HalfCheetah-v1 MuJoCo environments. For Ant-v1 the experiment with the Equation 4 loss function dropped off significantly around the 400,000 timestep mark, but this could be mitigated with early stopping.

However we obtained minimal learning for the Hopper-v1 and Walker2d-v1 MuJoCo environments, with both loss functions (Equation 4 and Equation 5) performing similarly (it does appear that taking the mean of the generator and expert loss improved training slightly for Walker2d-v1).

Referring to Figure 3, we attempted to reproduce an auxiliary claim made by the authors. The authors found that even without training the discriminator the reward functions can perform well on some tasks. For a naive discriminator, their reward function, $r(s,a) = log(D(s,a)) - log(1 - D(s,a))$, was able to retrieve up to around 0.3 normalized reward in the Hopper MuJoCo environment, and the original GAIL reward, $r(s,a) = -log(1 - D(s,a))$, reached up to 0.2 normalized reward before tapering off to around 0. In contrast, we found the original GAIL reward was able to retrieve around 0.3 normalized reward, whereas their reward function varied around the 0 mark.

[6] https://github.com/openai/imitation

[7] https://github.com/openai/baselines

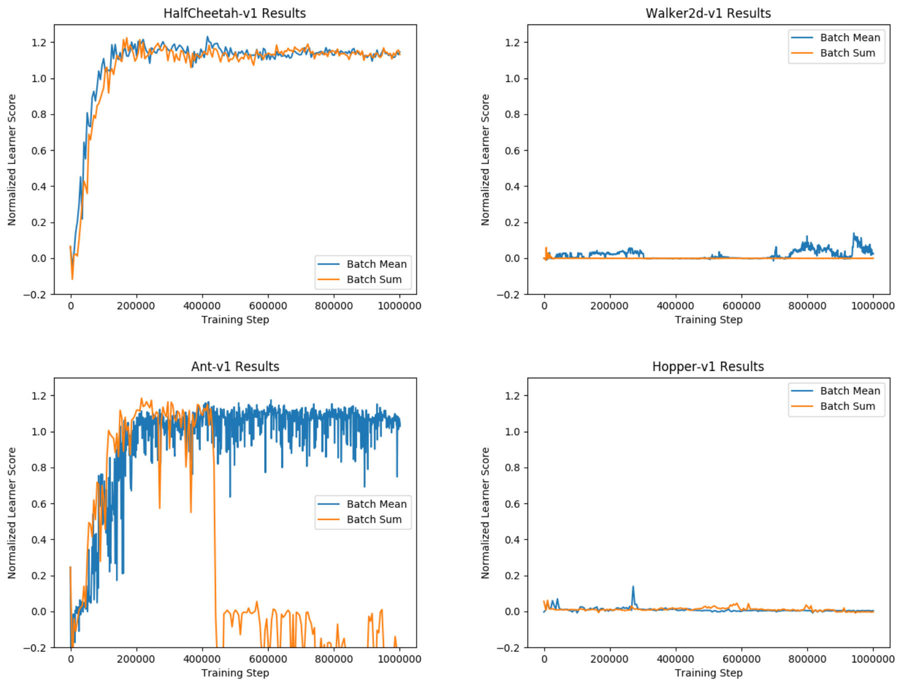


Figure 2: Normalized reward for our DAC implementation.

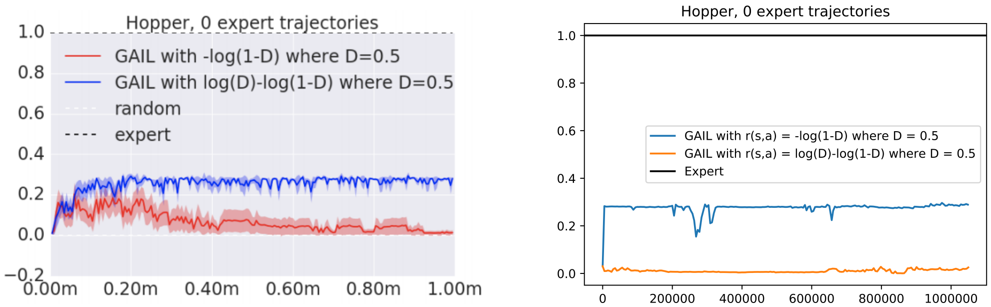


Figure 3: Comparison of reward function with the naive discriminator. Left: original results. Right: our results

5 CONCLUSION

We were able to reproduce the authors' results in two MuJoCo environments, but fell short in reproducing the results in all 4 MuJoCo environments. Further, we failed to demonstrate the behaviour of their new reward function with a naive discriminator. As such, our results are inconclusive; however, since we had limited time and budget, further experiments should be conducted, involving several runs with different random seeds per MuJoCo environment.

ACKNOWLEDGMENTS

We would like to thank Google Cloud for providing Google Deep Learning Virtual Machines to the Reproducibility project.

REFERENCES

Anonymous. Discriminator-actor-critic: Addressing sample inefficiency and reward bias in adversarial imitation learning. In *Submitted to International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Hk4fpoA5Km`. under review.

Michael Bain and Claude Sammut. A framework for behavioural cloning. 1995.

Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL `http://arxiv.org/abs/1802.09477`.

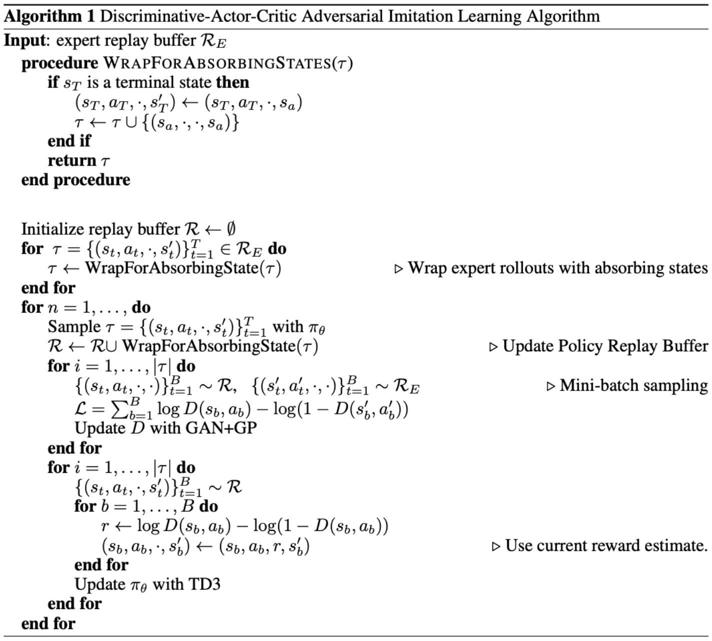
Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pp. 5767–5777, 2017.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *CoRR*, abs/1606.03476, 2016. URL `http://arxiv.org/abs/1606.03476`.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

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

A DAC Pseudocode

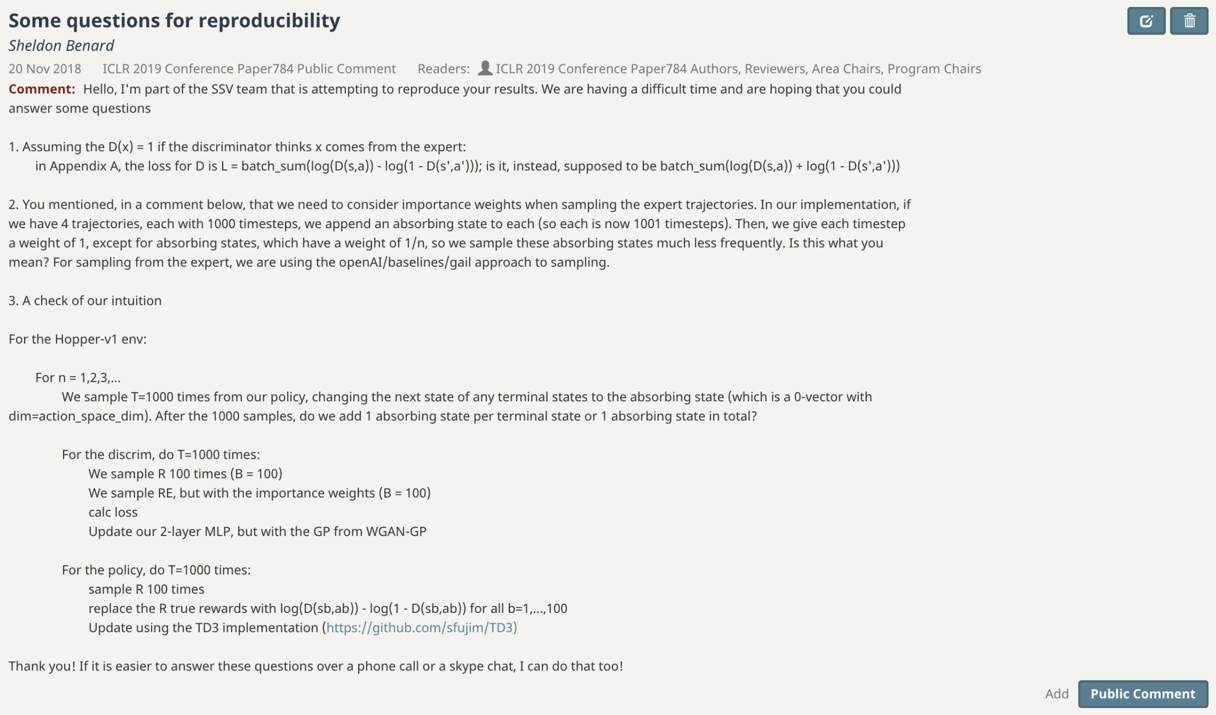


```
Algorithm 1 Discriminative-Actor-Critic Adversarial Imitation Learning Algorithm
Input: expert replay buffer $\mathcal{R}_E$
  procedure WrapForAbsorbingStates($\tau$)
    if $s_T$ is a terminal state then
      $(s_T, a_T, \cdot, s'_T) \leftarrow (s_T, a_T, \cdot, s_a)$
      $\tau \leftarrow \tau \cup \{(s_a, \cdot, \cdot, s_a)\}$
    end if
    return $\tau$
  end procedure

  Initialize replay buffer $\mathcal{R} \leftarrow \emptyset$
  for  $\tau = \{(s_t, a_t, \cdot, s'_t)\}_{t=1}^{T} \in \mathcal{R}_E$ do
    $\tau \leftarrow$ WrapForAbsorbingState($\tau$)                ▷ Wrap expert rollouts with absorbing states
  end for
  for $n = 1, \ldots,$ do
    Sample $\tau = \{(s_t, a_t, \cdot, s'_t)\}_{t=1}^{T}$ with $\pi_\theta$
    $\mathcal{R} \leftarrow \mathcal{R} \cup$ WrapForAbsorbingState($\tau$)          ▷ Update Policy Replay Buffer
    for $i = 1, \ldots, |\tau|$ do
      $\{(s_t, a_t, \cdot, \cdot)\}_{t=1}^{B} \sim \mathcal{R}, \quad \{(s'_t, a'_t, \cdot, \cdot)\}_{t=1}^{B} \sim \mathcal{R}_E$        ▷ Mini-batch sampling
      $\mathcal{L} = \sum_{b=1}^{B} \log D(s_b, a_b) - \log(1 - D(s'_b, a'_b))$
      Update $D$ with GAN+GP
    end for
    for $i = 1, \ldots, |\tau|$ do
      $\{(s_t, a_t, \cdot, s'_t)\}_{t=1}^{B} \sim \mathcal{R}$
      for $b = 1, \ldots, B$ do
        $r \leftarrow \log D(s_b, a_b) - \log(1 - D(s_b, a_b))$
        $(s_b, a_b, \cdot, s'_b) \leftarrow (s_b, a_b, r, s'_b)$          ▷ Use current reward estimate.
      end for
      Update $\pi_\theta$ with TD3
    end for
  end for
```

B Communication with the Authors

Private question to openReview:



Some questions for reproducibility

Sheldon Benard

20 Nov 2018 ICLR 2019 Conference Paper784 Public Comment Readers: ICLR 2019 Conference Paper784 Authors, Reviewers, Area Chairs, Program Chairs

Comment: Hello, I'm part of the SSV team that is attempting to reproduce your results. We are having a difficult time and are hoping that you could answer some questions

1. Assuming the D(x) = 1 if the discriminator thinks x comes from the expert:
    in Appendix A, the loss for D is L = batch_sum(log(D(s,a)) - log(1 - D(s',a'))); is it, instead, supposed to be batch_sum(log(D(s,a)) + log(1 - D(s',a')))

2. You mentioned, in a comment below, that we need to consider importance weights when sampling the expert trajectories. In our implementation, if we have 4 trajectories, each with 1000 timesteps, we append an absorbing state to each (so each is now 1001 timesteps). Then, we give each timestep a weight of 1, except for absorbing states, which have a weight of 1/n, so we sample these absorbing states much less frequently. Is this what you mean? For sampling from the expert, we are using the openAI/baselines/gail approach to sampling.

3. A check of our intuition

For the Hopper-v1 env:

```
    For n = 1,2,3,...
        We sample T=1000 times from our policy, changing the next state of any terminal states to the absorbing state (which is a 0-vector with
dim=action_space_dim). After the 1000 samples, do we add 1 absorbing state per terminal state or 1 absorbing state in total?

        For the discrim, do T=1000 times:
            We sample R 100 times (B = 100)
            We sample RE, but with the importance weights (B = 100)
            calc loss
            Update our 2-layer MLP, but with the GP from WGAN-GP

        For the policy, do T=1000 times:
            sample R 100 times
            replace the R true rewards with log(D(sb,ab)) - log(1 - D(sb,ab)) for all b=1,...,100
            Update using the TD3 implementation (https://github.com/sfujim/TD3)
```

Thank you! If it is easier to answer these questions over a phone call or a skype chat, I can do that too!

Response:

”Thank you for your interest in reproducing results reported in our paper.

1) Yes, thank you for pointing that out! We will fix it in the supplementary materials.

2) We used dataset subsampling from the original implementation (https://github.com/openai/imitation). In the original implementation the authors process as follows: given a trajectory they subsample taking every Nth transition starting from a random offset from 0 to N-1. For the experiments, they use N=20. Therefore, after subsampling for a trajectory of 1000 thansitions they have only 50 transitions left (200 for 4 trajectories). Due to the fact that in our implementation we add absorbing state after subsampling, we use importance weights for the discriminator loss. In particular, for 50 original transitions we use importance weight of 1 while for 1 added absorbing state we use an importance weight of 1/20.

3) You need to add 1 terminal state - absorbing state and 1 absorbing state - absorbing state transitions to the replay buffer after every terminal state. Please take into account that it has to be done only for episodic tasks (e.g. when episode is terminated not because of a time limit. See this line for a better intuition: https://github.com/sfujim/TD3/blob/master/main.py#L123). We might be able to provide more help if you share your current implementation of our paper (if it complies with ICLR rules).

We will be happy to answer any questions via Skype, we can create an anonymous account. However, could you please check whether it complies with the ICLR rules?”

NOTE: this was the only communication; we did not call or have a video conference with the authors.