DPO Reproducibility Challenge Report

Jiuyang Bai
Brown University
jiuyang_bai@brown.edu

Gregory Cho
Brown University
gregory_cho@brown.edu

Linlin Liu
Brown University
linlin_liu@brown.edu

Xingchi (Miles) Yan
Brown University
miles_yan@brown.edu

Liu Yang
Brown University
liu_yang@brown.edu

Abstract

This project attempted to reproduce some of the experimental findings presented by Tessler et al. (2019). In the paper, the authors present a novel reinforcement learning algorithm called the Generative Actor Critic (GAC), an implementation of distributional policy optimization (DPO), for continuous control problems. The authors evaluate GAC on several MuJoCo environments and obtain competitive results when compared to state of the art policy gradient baselines. The replicated GAC algorithm was ultimately successful in reproducing the learning curves for GAC on the MuJoCo Humanoid task, using an autoregressive implicit quantity network (AIQN) and implicit quantile network (IQN) as the actor. These findings, when compared with other algorithms, support the author's claim that GAC could be an alternative approach for continuous control. In addition to reproducing the original experiments, this report also elucidates on diagnostic investigations conducted throughout this project with the objective of presenting a better understanding of certain critical details in the algorithm. The replicated implementation of GAC used in this report can be found in the following public git repository: https://github.com/gwbcho/dpo-replication.

1 Introduction and Objective

The original paper describes a method to represent arbitrary quantile distribution functions over a continuous action space allowing for the construction of an optimal stationary stochastic policy. Without making assumptions about the underlying distribution, their generative algorithm, called the Generative Actor Critic, can overcome some limitations of more traditional policy gradient methods. The paper both proves the efficacy of their approach formally and through empirical evidence in the form of experimental results which show competitive cumulative rewards in numerous complex continuous control tasks.

In order to validate the paper's claims on the algorithm's efficacy, this project initially aimed to reproduce Figure 4 which plots the training curves of 6 different methods on 6 DeepMind MuJoCo control suites. Unfortunately, due to limited computational resources and time available, the project's scope was restricted to reproducing a single graph. Specifically, this project reproduced the graph of the algorithm's performance on the Humanoid challenge using a Boltzmann AIQN and IQN. The primary reason for this graph's selection was simply because the paper indicated that their approach resulted in a significant boost to performance when compared to traditional policy gradient methods. One thing to note is that GAC is computationally more expensive than current policy gradient methods resulting in slow rates of completion.

2 Background

2.1 Quantile Regression

Given a one dimensional probability measure μ with cumulative distribution function $F_\mu(x) = \mu((-\infty, x])$, the quantile function Q_μ of μ is the inverse of F_μ, formally

$$Q_\mu(\tau) = \inf\{x \in R : \tau \leq F_\mu(x)\} \quad \forall \tau \in [0,1] \tag{1}$$

One valuable property of the quantile function is that it can act as a generator that takes uniform random noise as input and outputs the samples from μ.

Quantile regression aims to learn the quantile function Q_μ with samples of μ as training data. It is easy to verify that

$$Q_\mu(\tau) \in \operatorname*{argmin}_x \mathrm{E}_{\xi\sim\mu}[(\xi - x)(\tau - 1_{\xi - x<0})], \quad \forall \tau \in [0,1] \tag{2}$$

Thus the quantile regression function can be represented as

$$L^\tau(x) = \mathrm{E}_{\xi\sim\mu}[(\xi - x)(\tau - 1_{\xi - x<0})] \tag{3}$$

To smooth gradients, in practice, it is common to use the Huber quantile loss function:

$$L_h^\tau = \mathrm{E}_{\xi\sim\mu}[L_\kappa(\xi - x)|\tau - 1_{\xi - x<0}|] \tag{4}$$

where L_κ is the Huber loss function defined as

$$L_\kappa(\theta) = \begin{cases} \frac{\theta^2}{2} & \text{if } |\theta| \leq \kappa \\ \kappa(|\theta| - \frac{\kappa}{2}) & \text{otherwise} \end{cases} \tag{5}$$

2.2 Actor Critic Framework

The Actor Critic framework is widely used in model-free reinforcement learning algorithms. In the GAC framework, the value, critic and actor networks aim to approximate the value function, critic function, and the policy function respectively. To resolve issues with numerical and gradient instability, in practice, two copies of these networks are created and identically initialized for each function, usually referred to as the network and the target network. During each training iteration, the actor network is trained using values determined by the target network with the target network being updated by the trained values of the actor using a set update rate. In addition, as suggested by Fujimoto et al. (2018), two critic networks are used to address the usual issues of variance with network gradients.

3 Model Implementation

The model that this project attempted to reproduce was the Generative Actor Critic (GAC) which uses either an AIQN or IQN (specified by the experimenter) to sample from some implied quantile network, learned via an actor critic approach. This section presents the implementation details of this project's replicated GAC algorithm. This reproduced version of the GAC algorithm was written in Tensorflow 2.0 as opposed to the original framework, PyTorch.

3.1 Main Loop

The main loop utilizes the agent and corresponding actor critic networks to iterate through the game environment, collect samples for the replay buffer, train network parameters, and evaluate the current performance of the learned stochastic policy. For the sake of consistency with the original experiment environment, all action selected during the replay buffer population phase were perturbed by random noise to allow for local exploration. This is a necessary component to avoid the algorithm being stuck in a local optimum. In order to ensure that sufficiently new transitions are stored in the buffer before training, the authors of the original paper implemented a "roll out" period where the agent would sample states and actions without updating the network parameters to populate the buffer. This process provides more information for the network to train on, ideally decreasing the time necessary till an optimal policy is determined. It should be noted that GAC is still theoretically correct without "roll out" periods and, while it was a convenient component to use while reconstructing the original experiment results, there were also signs during runs in simpler environments (i.e. LunarLanderContinuous, Pendulum, etc.) that indicated the GAC agent was learning without it.

3.2 Value Network Training

Given the state s and K actions $\tilde{a}$ sampled from the delayed actor, the minimum value of two critics were used to update the value network.

$$y_v \leftarrow \min_{i=1,2} \frac{1}{K} \sum_{j=1}^{K} Q_{\theta_i'}(s, \tilde{a}_j)$$

$$\phi \leftarrow \phi - \frac{1}{N} \nabla_\phi \sum (y_v - v_\phi(s))^2$$

An iteration of training ends with a soft update on the respective target network. Formally represented below.

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

Note that a soft update in this context is when the value of τ has the property $0 < \tau < 1$.

3.3 Critic Network Training

The value network produces the expected value for the critics given the transitions.

$$y_Q \leftarrow r + \gamma v_{\phi'}(s')$$

Although only the minimum value between two critics are used for policy and value updates, both critic networks are updated based on the expected value produced by the value network.

$$\theta \leftarrow \theta - \frac{1}{N} \nabla_{\theta_i} \sum (y_Q - Q_{\theta_i}(s,a))^2$$

While not documented in Algorithm 2, the authors implicitly regularized the critic network in the code, by applying the following update rule:

$$\theta \leftarrow \theta - \frac{1}{N} \nabla_{\theta_i} \sum (y_Q - Q_{\theta_i}(s, \text{clip}(a+\epsilon)))^2,$$

where ϵ is uniform noise from -0.01 to 0.01, and "clip" means clipping the perturbed actions to fit the bound for actions.

An iteration of training ends with a soft update on the respective target network. Formally represented below.

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$$

Note that, as before, a soft update in this context is when the value of τ has the property $0 < \tau < 1$.

3.4 Actor Network Training

The target distribution of the generative actions conditioned on input state is induced from the set of advantage values evaluated via the critic and value networks. In general, the training of the actor network consists of three steps. The first step consists of sampling actions as candidates for each state in the training batch. In this step, randomly sampled actions and policy sampled actions are concatenated as candidates. The second step involves picking out the state and action pairs with positive advantage from the previously collected candidates. The advantage is evaluated using the target critic networks and the target value network. Finally, the last step is to determine the gradient for the the parameters of the actor network using the Huber quantile loss function so that the distribution of the generated actions approaches the target distribution.

4 Efforts

The efforts of this project were primarily focused on implementing Algorithm 2 from Tessler et al. (2019) which comprehensively details the GAC process. The core components of the algorithm were sufficiently documented to make a relatively faithful replica from scratch, though, there were several components that were helpful in this challenge yet were mentioned neither implicitly nor explicitly in the author's paper. These components, instead, were only discovered after analyzing the original GAC code. One such detail was the regularization of critic networks which aided significantly in speeding up the training process. Regularization in this context being the process of perturbing the actions sampled from the buffer during the training of the critic networks such that similar actions were forcibly assigned similar state action values. It is recognized by the members of this project that this practice is similar to the critic regularization used in the TD3 algorithm and is likely common knowledge in the field. Furthermore, for numerical stability, when running the algorithm on certain complex tasks it was useful to implement the normalization of rewards, states, and actions.

In addition to missing information regarding regularization and normalization, this project encountered several minor issues while implementing the reproduced algorithm in Tensorflow 2.0. Most of the issues encountered were due to features in Tensorflow 2.0 that group members were unaware of. One issue was with the initialization of trainable variables for the actor, critic, and value networks. In the algorithm, the target networks should be initialized with the same initialization values as the fast networks, as indicated in line 3 of Algorithm 2. In the replicated implementation, the actor, critic, and value functions are all Tensorflow modules, allowing for easy access to trainable variables during the training procedure. Unfortunately, Tensorflow 2.0 does not initialize the trainable_variables list until they are actually used, resulting in the update function being unable to forcibly set the parameters of the target networks to those of the fast networks. A solution was found by simply feeding dummy inputs into the networks thereby populating the trainable_variables list before updating them to be the same value. The final issue encountered during our implementation was with the Huber Loss Function which automatically reduced the loss values into a singleton as opposed to a vector of losses. This issue was easily rectified by setting the reduction variable to tf.keras.losses.Reduction.NONE.

As stated earlier, in order to fully replicate the GAC algorithm described in the paper, the author's code was consulted for clarification. No code from their original repository was actually used, however, with their implementation serving only as a guide to refine the replica. Code from the OpenAI baseline repository was used, though, to implement noise and normalization functionality. In addition, this group contacted the author of the paper with minor questions about some of its more esoteric subjects. These dialogues never covered contents pertaining to their implementation beyond what their paper described and were mostly conceptual.

5 Results

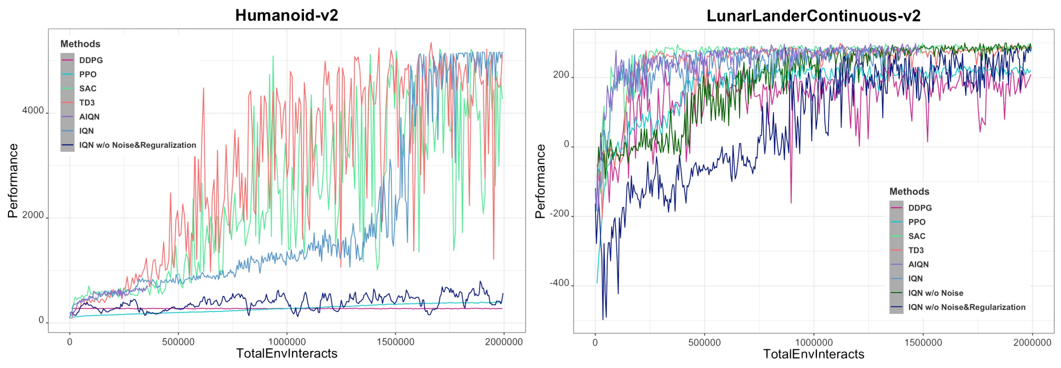

Figure 1: Results from the Humanoid experiment (left) and the LunarLanderContinuous experiment (right) run on this project's replicated GAC algorithm with the number of environment interactions on the x axis and cumulative reward on the y axis. The AIQN data for the Humanoid experiment is only partially complete due to limitations on time and resources. In addition, the AIQN GAC method used a smaller batch size than the IQN version (64 as opposed to 128) simply to reduce the time necessary till sufficient data for a graph was collected. The exploration noise for the Humanoid experiment is drawn from an i.i.d Gaussian distribution and an Ornstein–Uhlenbeck process for LunarLanderContinuous.

The replicated code was tested on two OpenAI Gym (Brockman et al., 2016) environments, LunarLanderContinuous-v2 and MuJoCo's Humanoid-v2. The selection of the Humanoid task out of the six other original experiments was done simply due to the dramatic increase in performance achieved by the author's IQN approach when compared to other baseline methods. To test other state of the art RL methods and acquire data for a baseline comparison, this project utilized the code from https://github.com/openai/spinningup to run DDPG, PPO, SAC, and TD3 algorithms and plot their performance. The results of both experiments are illustrated in Figure 1. To provide a greater understanding of critical algorithm components, the figures also illustrate the results of an IQN actor without the exploratory noise in line 6 of Algorithm 2 and the regularization of critic networks.

As seen in the resulting data, this project was successfully able to reproduce the core findings of the original paper's experiment. Indeed, compared with the original results, the experiments run on the replicated algorithm actually achieved higher rewards than was previously reported. This success in reproducing the results of an IQN agent on the Humainoid-v2 problem, in particular, supports the empirical findings and claims made by the original paper; indicating that GAC could be considered a legitimate alternative approach to solving continuous control tasks. However, while competitive, the results garnered by the GAC algorithm were not significantly better than the state of the art policy gradient methods (at least not to the same degree seen in the paper) with many of the alternative methods achieving similar cumulative rewards compared to the IQN GAC approach. Specifically, the TD3 algorithm used in this experiment achieved substantially greater rewards than was seen in the original paper. It is likely that this deviation from the original paper's findings is due to differences in hyperparameter values for both GAC and other policy gradient methods.

It should be noted that initial trials using an IQN actor on the Humanoid environment, run without certain components, were unsuccessful at reproducing the results seen in the original paper's experiments. As mentioned before, one necessary component for replicating the success of the GAC algorithm was the regularization of the critic function. As a result of adding regularization, the algorithm was not forced to learn new values for similar states and actions, allowing for GAC to more quickly converge to an optimal policy. In addition, the original replicated implementation neglected to use perturbed actions for exploration under the assumption that there was implicit noise in the model. After implementing Gaussian noise with standard deviation 0.2 during the training phase consistent with line 6 in Algorithm 2, though, the model trained significantly faster in both problems. Intuitively, this action is akin to the model being more exploratory during the training phase. While the noise used in this project's experiment was sampled from a Gaussian distribution with greater standard deviation than the original (0.2 as opposed to 0.1), it appears the results are still consistent with those of the paper.

6 Reflection

The novelty of the GAC algorithm mainly lies in the introduction of a generative actor to sample policies, and training said actor so that the distribution of generated actions approaches a target distribution with support over a set of advantageous actions. The resulting stochastic policy from GAC is thus more flexible than deterministic policies or policies of parametric distributions. This approach clearly represents a step in an interesting and promising direction towards the use of distributional policy optimization for complex reinforcement learning tasks. That being said, there were points in the algorithm description which were slightly confusing to this group as a whole which, while not a significant issue, could likely be clarified or improved on in some fashion. It appears that there was a mistake on line 14 in Algorithm 2 as, despite representing the mean, $\frac{1}{K}$ is missing. Furthermore, in the algorithm, the authors introduced both value networks and critic networks for use in policy optimization. However, only advantage, i.e. the difference between critic and value networks, is used to train the actor. It is therefore likely possible to replace the value and critic networks with a single value network, similar to traditional A2C algorithms, thereby reducing the computational cost of training. In addition to these minor issues within the algorithm description, the utilization of random noise for exploration was slightly confusing as such randomness should, in theory, be inherent to a generative actor. This difficulty with regards to mathematical interpretation extends to the concatenation of uniformly sampled actions with those that are policy sampled while acquiring states and actions of positive advantage.

7 Conclusion

DPO provides an interesting alternative approach to continuous control methods. Using a generative actor as a stochastic policy, DPO – and GAC by extension – is more flexible in action sampling than traditional policy gradient approaches, allowing it to perform as well as if not better than state of the art RL solutions. This project, by reproducing the key results of the author's experiment, is in support of the empirical findings and claims made by the original paper. Despite this, there are still some observed limitations to this method. Primarily, the computational cost associated with this algorithm is notably higher than baseline methods, potentially limiting its application scope.

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Scott Fujimoto, Herke van Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*.

Chen Tessler, Guy Tennenholtz, and Shie Mannor. 2019. Distributional policy optimization: An alternative approach for continuous control. *arXiv preprint arXiv:1905.09855*.