DAC Replication Report

Yichen Chai
yichen_chai@brown.edu

Zhe Hu
zhe_hu@brown.edu

Cancan Huang
cancan_huang@brown.edu

Yan Huang
yan_huang@brown.edu

Ziyao Huang
ziyao_huang@brown.edu

Zezhi Wang
zezhi_wang@brown.edu

Abstract

This paper is a reproduction of DAC: The Double Actor-Critic Architecture for Learning Options by Zhang and Whiteson ((2019)). It tried to apply the actor-critic technique to hierarchical reinforcement learning. We re-implemented the DAC and some other baselines the author used for experiments and evaluated the reproducibility of this work. We conducted empirical studies on 4 Mujoco games the authors used with their hyper-parameters. In this setting, our results partially support their claims.

1 Introduction

This DAC paper focuses on the hierarchical reinforcement learning area. The author examines the existing hierarchical reinforcement learning methods and addresses two issues: (reference here)

- Although policy-based methods are often preferred in the MDP setting, theoretical study for learning a master policy with policy-based intra-option methods is limited, and its empirical success has not been witnessed so far.
- Despite the recent successes of gradient-based option learning algorithms, most of them customize the original algorithm to the option-based SMDP. Consequently, we cannot directly leverage recent advances in gradient-based policy optimization from MDPs.

To address these issues, this paper reformulates the SMDP of the option framework as two augmented MDPs: one master MDP to select option, and several lower level MDPs for deciding action given a specific option. The author applies actor-critic algorithm on each augmented MDP. Under this setting, any policy optimization algorithms can be easily applied without specific customization for the policy learning. The author also ran experiments on a few OpenAI Mujoco tasks, using their DAC architecture combined with the Proximal Policy Optimization algorithm.

This paper contains two main experiments: single task learning and transfer learning. We will focus on replicating the single task learning experiments first.

The conclusions from the single task learning experiments are:

- DAC+PPO outperform other algorithms (OC, IOPG, DAC+A2C). This shows that the performance boose mainly comes from the PPO algorithm, which is the advantage of DAC and AHP: to use policy optimization algorithms off the shelf to learn options.
- DAC+PPO performs similarly to vanilla PPO in 3 out of 4 single tasks. This is because options aren't necessary for single tasks for good performance.

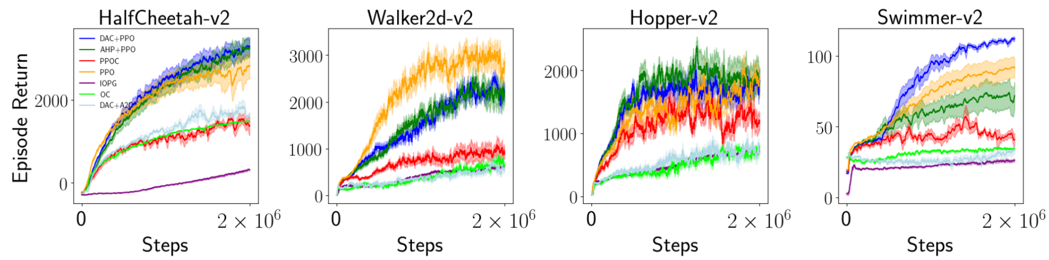


Figure 1: DAC single task learning result

The original single task learning experiment results are shown in Figure 1.

Therefore, we would like to rerun the experiments using these algorithms:

DAC+PPO, PPO, DAC+A2C, OC, PPOC

to see if our results are consistent with their conclusion. Since OC and IOPG have similar performance, we will only do OC instead of both as a baseline.

2 Algorithms Explanation

2.1 Double Actor-Critic Architecture

The Double Actor-Critic architecture (DAC) reformulates the traditional option framework into two augmented MDPs. One high level MDP M^H is used to learn options, and a low level MDP M^L learns actions for a given option. Both MDPs share same samples from the environment for higher sample efficiency.

The agent's learning process is shown in Algorithm 1:

Algorithm 1: Double Actor Critic algorithm

Initialize networks weight;
Reset the task env;
Get the initial state S_1 of the task, and initial option O_0;
while *current step < max step* **do**
  for *timestep=1,2,$\cdots$* **do**
    Sample an option O_t from $\pi_h(O|(O_{t-1}, S_t))$;
    Sample an action A_t from lower level policy $\pi_l(A|(O_t, S_t))$;
    Apply the sampled action in the task to get next state S_{t+1} and reward R_t;
    Update current step number and record the reward R_t for plotting;
  Apply some policy gradient algorithm to update networks weights;

In our actual training, we set the max step to be two million for all of our agents, as the author did.

2.2 Proximal Policy Optimization

Proximal Policy Optimization(PPO) is an algorithm proposed by Schulman et al. ((2017)), having state-of-the-art performance but is easy to implement and tune compared to other algorithms. It stems from the idea of off-policy policy gradient, utilizing importance sampling in the process of updating the policy so that the same set of data can be used to train the agent for several epochs, largely improving the data efficiency. Besides, it improves upon Trust Region Policy Optimization (TRPO) in that it uses a much simpler constraint while achieving similar performance.

For PPO, the objective function is

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \quad r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$$

In contrast to TRPO where a constraint of KL-divergence between the old and new policy is applied, PPO simply clips the loss function to serve the purpose of ensuring the stability of the training process, and it turns out to work pretty well in many kinds of tasks.

The Pseudo Code of PPO is shown in Algorithm 2.

Algorithm 2: Proximal Policy Optimization

Run $\pi_{\theta_{old}}$ for T timesteps ;
Compute advantage estimates $\hat{A}_1, \cdots, \hat{A}_T$, and record $\pi_{\theta_{old}}(a_1|s_1), \cdots \pi_{\theta_{old}}(a_T|s_T)$;
for *epoch=1,2,$\cdots$* **do**
    for *minibatch=1,$\cdots$T/minibatch_size* **do**
        Sample minibatch_size of experience steps;
        Compute $L^{\mathrm{CLIP}}(\theta)$ with current π_θ;
        Compute the critic error and add them together;
        Backprop to update π_θ;

2.3 Option Critic

The Option-Critic Architecture proposed by Bacon et al. ((2017)) is an architecture built upon the combination of option framework and the classical actor-critic algorithm, where they derived the policy gradient theorems for options, i.e.

$$\begin{aligned}
\nabla_\nu v_\pi(S_0) &= \sum_{s,o} \rho(s, o|S_0, O_0) \sum_a q_\pi(s, o, a) \nabla_\nu \pi_o(a|s), \\
\nabla_\phi v_\pi(S_0) &= -\sum_{s',o} \rho(s', o|S_1, O_0)(q_\pi(s', o) - v_\pi(s')) \nabla_\phi \beta_o(s')
\end{aligned}$$

where $\rho(s, o|S_0, O_0) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s, o_t = o|S_0, O_0)$. Based on these theorems, they constructed an algorithm known as option-critic, where the actor consists of the intra-option policies, termination functions, while the critic contains the calculation of q_π (and v_π), and a ϵ-greedy master policy is applied over options.

The pseudo code of OC is shown in Algorithm 3:

Algorithm 3: Option-critic with Q-learning

$s \leftarrow s_0$;
Choose ω according to an ϵ-soft policy over options $\pi_\Omega(s)$;
while *s is non-terminal* **do**
    Choose action α according to $\pi_{\omega,\theta}(\alpha \mid s)$;
    Execute α in s, observe s', r;
    $\delta \leftarrow r - Q_U(s, \omega, \alpha)$;
    if *s' is non-terminal* **then**
        $\delta \leftarrow \delta + \gamma(1 - \beta_{\omega,\vartheta}(s'))Q_\Omega(s', \omega) + \gamma\beta_{\omega,\vartheta}(s') \max_{\bar{\omega}} Q_\Omega(s', \bar{\omega})$;
    $Q_U(s, \omega, \alpha) \leftarrow Q_U(s, \omega, \alpha) + \alpha\sigma$;
    $\theta \leftarrow \theta + \alpha_\theta \frac{\partial \log \pi_{\omega,\theta}(\alpha|s)}{\partial \theta} Q_U(s, \omega, \alpha)$;
    $\vartheta \leftarrow \vartheta - \alpha_\vartheta \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta}(Q_\Omega(s', \omega) - V_\Omega(s'))$;
    if *$\beta_{\omega,\vartheta}$ terminates in s'* **then**
        choose new ω according to ϵ-soft($\pi_\Omega(s')$);
        $s \leftarrow s'$;

2.4 Advantage Actor-Critic

Advantage Actor-Critic (A2C) released by OpenAI is a synchronous, deterministic variant of Asynchronous Advantage Actor-Critic proposed by Mnih et al. ((2016)) with equal performance. The advantage of a state-action pair is the difference between the state-action Q value, and the state's value:

$$A(s,a) = Q(s,a) - V(s)$$

In this algorithm, first initialize network parameters θ with random values. Second, play N steps in the environment using the current policy π_θ, saving state s_t, action a_t and reward r_t. Third, $R = 0$ if the end of the episode is reached or $V_\theta(s_t)$. Fourth, for $i = t-1...t_{start}$, accumulate the PG $\delta\theta_\pi \leftarrow \delta\theta_\pi + \nabla_\theta log\pi_\theta(a_i|s_i)(R - V_\theta(s_i))$ and the value gradients $\delta\theta_v \leftarrow \delta\theta_v + \frac{\delta(R-V_\theta(s_i))^2}{\delta\theta_v}$. Finally, update parameters using the accumulated gradients, moving in the direction of PG $\delta\theta_\pi$ and in the opposite direction of the value gradients $\delta\theta_v$. Repeat from second step until the convergence is reached.

3 Challenges

Throughout the process of trying to understand and replicate this work, we encountered some challenges.

- Understanding the DAC architecture required us to have deep understandings of some prior works that were new to us.
- Implementing several baselines is a considerable amount of work.
- The way policy gradient algorithms are used to optimize the two MDPs separately in DAC architecture is confusing to us, so we had to look at the author's implementation as a reference to help us understand how DAC and policy gradient are integrated together.

4 Implementation

Our code for experiment can be found at our Github Repository[1].

4.1 Fully Connected Network

Each of our agents utilizes multiple fully connected networks in their implementation. They all have similar architecture, with the difference mostly being in the output layer.
They all have an input layer with dimension being the state dimension, two hidden layers with 64 nodes each, and one output layer. The author used the Tanh activation function for the first three layers in Mujoco tasks, but we found a ReLU activation function generates better results. That's what we use for our replicate experiments.

4.2 PPO

Our implementation of PPO is based on higgsfield's implementation [2]. In the real implementation, a parametrized Gaussian policy is employed as the actor in the continuous situation, and Generalized Advantage Estimation(GAE) by Schulman et al. ((2015)) is used to compute the advantage function. 10 epochs are applied to train the data from every 2048 steps, with a minibatch size of 64.

Also, we find an interesting phenomenon that the orthogonal initialization by Saxe et al. ((2013)) with an appropriate weight scale plays an important role in achieving good performance for the PPO algorithm. For example, in the HalfCheetah task, the PPO algorithm without orthogonal initialization can only achieve an average episodic reward of 2500 to 3000 in 2e6 steps, while with orthogonal initialization (weight scale 1) the PPO algorithm is able to achieve an average episodic reward over 3000 to even 4000.

4.3 DAC Architecture

Our DAC agent keeps track of $2k + 2$ separate fully connected networks, where k is the number of options.

[1] https://github.com/DAC-Prime/supreme-waffle

[2] https://github.com/higgsfield/RL-Adventure-2/blob/master/3.ppo.ipynb

We use 2 networks for high level MDP learning, one to predict master policy, the other one to predict the values of the current state. Both networks have the number of options as their output dimension. The policy net uses softmax as output layer activation function, and the value net output the prediction directly without any activation function.

For each option, we utilize two more networks for action selection. The first one generates an action policy for its corresponding option given a state, and the other one predicts the termination probability. The policy net outputs a tensor with the same shape as the Mujoco task action. It has Tanh as the final activation function. The termination net predicts a single number, using sigmoid as final activation.

4.4 OC

For OC, we here implement a slightly different version of option-critic based on the implementation of DAC's authors'[3] for comparison, which uses multiple (16) workers to train the option-critic synchronously, and only compute the gradients and update the policy every 5 steps.

Built upon option-critic, we also implement the Proximal Policy Option Critic (PPOC) algorithm proposed by Klissarov et al. ((2017)), which is a combination of PPO and OC. PPOC employs the PPO loss function to train the intra-option policies, and include the master policy (softmax) over options in its actor so that it will also be trained by backpropagation.

4.5 A2C

We consulted the OpenAI baselines for A2C [4].

5 Environment Setup

5.1 OpenAI Gym and Mujoco

The experiments in this paper consist of two parts: single task learning and transfer learning. The single task learning part, which we focus on, uses four Mujoco environments from OpenAI Gym:

HalfCheetah-v2, Walker2d-v2, Hopper-v2, Swimmer-v2

In the appendix, the author mentioned that the states from environments are normalized using running std and mean as preprocessing steps. This is also confirmed in the OpenAI Gym Github repo that Mujoco environments states require this normalization preprocess for it to be learned properly. We utilized the existing normalizer provided in the OpenAI Baseline Github repo[5] for this purpose.

5.2 Platform

The experiments are all ran on the cluster of Center for Computation Vision at Brown University. Anaconda is used for package control and environment setup. The activation file of Mujoco is provided by Prof. Michael Littman. To maintain the consistency between different systems, we also create a Docker recipe from which a Docker image can be built and ran in any system.

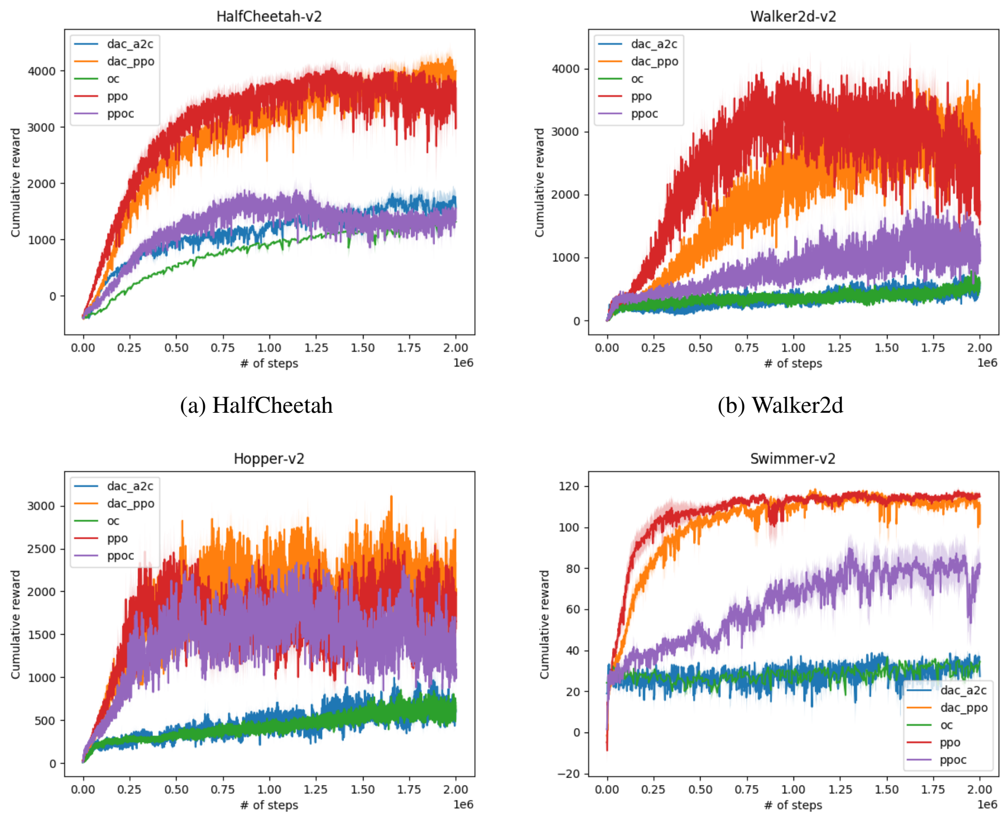
5.3 Plotting

For our experiments, to avoid coincidence and extend the universality, we have trained 10 agents for each of the four Mujoco tasks(HalfCheetah-v2, Walker2d-v2, Hopper-v2, and Swimmer-v2) to plot our episodic cumulative rewards graph. The results of the mean value with the standard error were plotted for the agents we have implemented.

6 Experiments Result

Our replicated performance of the DAC-PPO, DAC-A2C, OC, PPO and PPOC agent roughly matches the result reported in the original DAC paper.

[3] `https://github.com/ShangtongZhang/DeepRL/blob/DAC/deep_rl/agent/OC_agent.py`
[4] `https://github.com/openai/baselines/blob/master/baselines/a2c`
[5] `https://github.com/openai/baselines`



(a) HalfCheetah

(b) Walker2d

(c) Hopper

(d) Swimmer

Figure 2: Experimental results for four Mujoco games

Specifically,

1. Our implementations of DAC-A2C and OC turn out to have approximately the same performance as the authors', which is worse than that of PPO and DAC-PPO. The author concluded that the advantage mainly comes from PPO, and our results are in line with this claim.

2. For HalfCheetah-v2 and Hopper-v2, our DAC-PPO agent and PPO agent achieve approximately the same performance after two million steps of learning. This matches the original result.

3. For Walker-v2 task, the author concluded that the DAC-PPO and PPO agents perform similarly, although the graph showed that the vanilla PPO performs a little better than DAC-PPO. Our agents perform pretty close to each other at the end of training.

4. For Swimmer-v2 setting, the author reported a better performance from DAC-PPO agent compared to vanilla PPO. Our DAC-PPO agent achieved similar cumulative rewards to the author's result, but our vanilla PPO performed better than theirs. So our result doesn't have that large margin in the original result.

We have also noticed a larger variation in our replicated agents' performance compared to the graph reported in the DAC paper. The reason could be due to a bug inside the original implementation. The author unintentionally fixed the Mujoco environments random seed across multiple runs, making their experiment less stochastic.[6]

[6] `https://github.com/ShangtongZhang/DeepRL/issues/67`

We did not have the chance to re-implement and experiment the Augmented Hierarchical Policy (AHP) architecture, so our replication could not verify the author's comparison between DAC and AHP.

7 Reflection

Our experiment results roughly match those reported by the author. We think our replication experience partially supports the paper.

About the first point, our replication generates state-of-art performance in the four Mujoco tasks. This is an empirical success of master policy learning with policy-cased intra-option methods.

Our replication verifies that the proposed DAC architecture is able to utilize PPO and A2C algorithms for optimization. However, it's hard for us to figure out the way DAC architecture and policy gradient algorithms are integrated together from the paper alone. It's probably due to our lack of background knowledge and experience, but that's still not intuitive from our perspective.

The choice of activation function can influence the results significantly, which might be worth digging in the underneath reasons.

References

Shangtong Zhang and Shimon Whiteson. Dac: The double actor-critic architecture for learning options. *ArXiv*, abs/1904.12691, 2019.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *arXiv preprint, arXiv:1707.06347.*, 2017.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 2016.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Martin Klissarov, Pierre-Luc Bacon, Jean Harb, and Doina Precup. Learnings options end-to-end for continuous action tasks, 2017.