PPO vs. SAC: A Comparative Study on Learning Stability and Performance

Mischa Mez
EPFL
Robotics
mischa.mez@epfl.ch

Guillaume Spahr
EPFL
Robotics
guillaume.spahr@epfl.ch

Abstract

This paper presents a comparative analysis of two prominent actor-critic algorithms in Deep Reinforcement Learning (Deep RL): Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). Deep RL leverages deep neural networks to enhance the function approximation capabilities of traditional reinforcement learning (RL). PPO addresses the limitations of prior RL methods such as deep Q-learning and vanilla policy gradient by using a clipped surrogate objective to balance sample efficiency and runtime performance. On the other hand, SAC, designed for both discrete and continuous action spaces, maximizes policy entropy to avoid local minima and employs a soft update mechanism for stability. This study evaluates the performance and learning stability of PPO and SAC across various benchmarks, including Cartpole, Acrobot, Pendulum, and MountainCarContinuous. The results highlight the strengths and trade-offs of each algorithm, providing insights into their applicability for different RL tasks.

Keywords: Deep Reinforcement Learning, Proximal Policy Optimization, Soft Actor-Critic, Actor-Critic Methods, Policy Entropy, Learning Stability, Sample Efficiency.

1 Introduction

Reinforcement Learning methods can be broadly categorized into three main types: value-based, policy-based, and actor-critic methods. Our goal is to detail two actor-critic algorithms, namely Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). We classify PPO within the framework of actor-critic methods, even though most papers discuss it as a policy gradient algorithm. While PPO is indeed a policy gradient algorithm, it also belongs to the actor-critic methods. PPO learns the optimal policy via the actor network and the value function via the critic network. However, unlike SAC, the actor in PPO is learned via policy gradient (not Q-learning). Therefore, it is also a policy-based method.

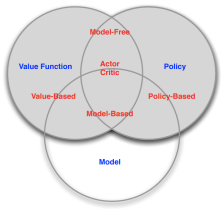

Figure 1: Reinforcements models

Preprint. Under review.

2 Related Work

Proximal Policy Optimization, introduced by Schulman et al. (2017) [3], has become a very popular algorithm in policy-based reinforcement learning. By using a clipped probability ratio in its objective function, PPO effectively controls how much the policy can be updated, thus enhancing stability and improving sample efficiency. Our approach builds on this by implementing the concept highlighted in the paper and making a real implementation for various Gym environments. On the other hand, Soft Actor-Critic, developed by Haarnoja et al. (2018) [1], is a model-free, off-policy algorithm originally designed for continuous action spaces. SAC's key innovation lies in its entropy-maximizing objective, which maintains a degree of randomness in the policy and prevents premature convergence to suboptimal policies. This entropy term, combined with a soft update mechanism for target networks, contributes to SAC's stability during training. Our work adapts SAC to discrete action spaces and simplifies the original algorithm by omitting value functions, relying instead on dual Q-functions to express new targets. This adaptation is inspired by the work on Spinning Up by OpenAI [2] and seeks to retain SAC's core benefits while extending its applicability. Additionally, we wrote the code with the original algorithm for the discrete case to verify that this adaptation shows the same results as the one proposed by the paper.

3 Methodology

Our approach aims to compare both of these algorithms by specifically focusing on their stability and performance in both discrete and continuous action spaces. We follow the guidelines of the papers, implementing the variance-reduced advantage-function estimator such as the generalized advantage estimation (GAE) for PPO to further enhance sample efficiency, and using the reparameterization trick in SAC to improve learning dynamics.

Our methodology involves a series of steps designed to evaluate and compare the performance and stability of these algorithms across various tasks and environments. To ensure a comprehensive comparison, we implemented both algorithms in several benchmark environments with discrete and continuous action spaces. The environments chosen for our experiments include:

- **CartPole**: A classic control problem where the goal is to balance a pole on a cart by applying discrete forces.
- **Acrobot**: A two-link pendulum system where the objective is to swing the lower link up to a specified height.
- **Pendulum**: A single pendulum where the goal is to swing it up and balance it.
- **MountainCarContinuous**: A continuous state environment adapted for discrete actions, where the goal is to drive a car up a steep hill.

These environments were selected for their varying levels of complexity and the different challenges they present, allowing us to thoroughly assess the algorithms' capabilities.

3.1 Implementation Details

In the following section, we detail our algorithm implementation using pseudocodes and visual diagrams.

Algorithm 1 PPO-Clip with Continuous and Discrete Action Spaces

Input: Initial policy parameters θ_0, initial value function parameters ϕ_0
Initialize: Environment env, memory buffer memory
for $k = 0, 1, 2, \ldots$ **do**
    Reset environment and get initial state s_0
    for each step in the environment **do**
        Select action a_t according to the current policy $\pi(\theta_k)$
        Execute action a_t in the environment
        Observe next state s_{t+1}, reward r_t, and done
        Store $(s_t, a_t, r_t, \text{done}, \log \pi_{\theta_k}(a_t|s_t), V_{\phi_k}(s_t))$ in memory
        if done is true **then**
            Reset environment and get initial state s_0
        end if
    end for
    Compute returns R_t and advantage estimates $\hat{A}_t$ using Generalized Advantage Estimation
    Update policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \hat{A}^{\pi_{\theta_k}}(s_t, a_t), \text{clip}(\epsilon, \hat{A}^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - R_t)^2$$

    typically via some gradient descent algorithm.
    Plot training history every k^{th} interval
    Clear memory buffer
end for

Algorithm 2 Soft Actor-Critic (SAC)

Input: Initial policy parameters θ, initial Q-function parameters ϕ_1, ϕ_2, and initial target parameters copied from ϕ_1, ϕ_2
for each iteration **do**
    Collect set of trajectories:

$$a_t \sim \pi_\phi(a_t|s_t)$$
$$s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)$$
$$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$$

    if must update **then**
        for each gradient step **do**
            Sample a batch of random elements from replay buffer $\mathcal{D}$: $B = \{(s, a, r, s')\}$
            Update the two Q-functions by minimizing:

$$\frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(Q_{\theta_i}(s, a) - \hat{Q}(s, a) \right)^2$$

            with $\hat{Q}(s, a) = r + \gamma \left(\min_i Q_{\textbf{target},i}(s', a') - \alpha \log(\pi_\theta(a'|s')) \right)$
            Update the policy by maximizing:

$$\frac{1}{|B|} \sum_{(s,a) \in B} \left(\min_i Q_{\theta_i}(s, a) - \alpha \log(\pi_\theta(a|s)) \right)$$

        via gradient ascent algorithm
            if must update Q target **then**
                Perform a soft update on the parameters:

$$\phi_{\text{target},i} \leftarrow \tau \phi_i + (1 - \tau) \phi_{\text{target},i}$$

            end if
        end for
    end if
end for

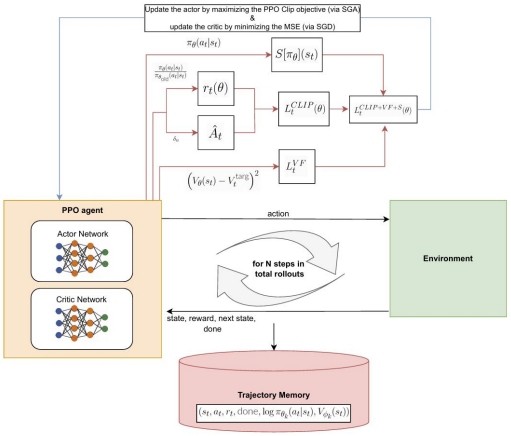

Figure 2: High-level diagram of the proximal policy optimization algorithm

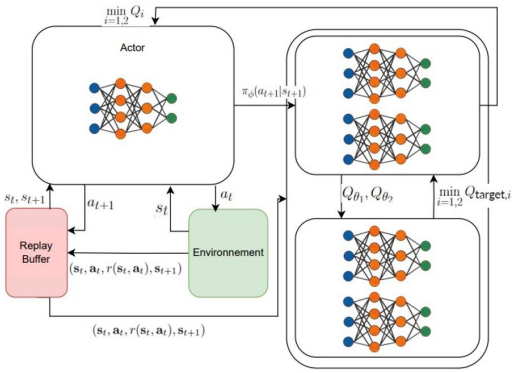

Figure 3: High-level diagram of the soft actor-critic algorithm

3.2 Evaluation Metrics

The primary metrics for evaluation were:

- **Learning Stability**: Measured by the variance in cumulative rewards across episodes.
- **Sample Efficiency**: Evaluated by the number of episodes required to reach a predefined performance threshold.
- **Final Performance**: The highest cumulative reward achieved by each algorithm.

We conducted the experiments using a consistent computational setup (use of seeds) to eliminate hardware-induced variability. Each algorithm was run 3 times to account for stochasticity in the training process, and the results were averaged.

4 Results

This section presents the experimental results for the comparative study of Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) algorithms. The experiments were conducted on four different environments: CartPole, Acrobot, Mountain Car Continuous, and Pendulum. Each environment was chosen to test the algorithms' performance across various complexities and action spaces.

4.1 CartPole

- **Performance & stability:**
  - PPO: Higher learning rates enable rapid learning with quick convergence to optimal performance and good stability, as shown in Figure 4; however, using lower or higher learning rates can cause instability. The entropy coefficient significantly affects stability, where excessive exploration leads to considerable fluctuations even after achieving high scores. Higher critic learning rates and entropy coefficients lead to excessive exploration and poor performance. Additionally, a lower batch size improves performance for simpler problems (Figure 5).
  - SAC: The plot shows that SAC can learn rapidly, but it comes with a high computational cost when increasing the gradient steps. Increasing the training interval, the score has difficulties reaching a high score. In fact, the best value was 1, meaning that at each step the critics and the actor are updated. It probably means that each step gives a high level of information to the agent. Despite fast convergence and a relatively high batch size of 128, we observe that the variance can be high even in the

last episodes. This is probably due to the number of gradient steps and the update's frequency of the weights.

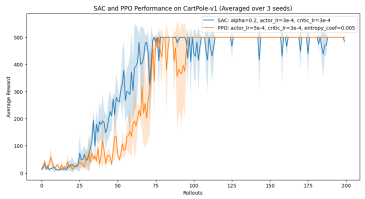

Figure 4: CartPole: best parameters

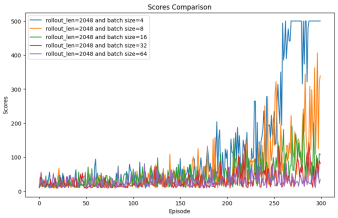

Figure 5: CartPole Bad Parameters (PPO)

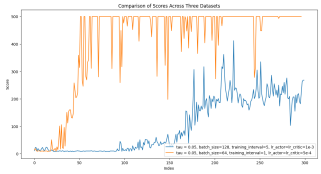

Figure 6: CartPole Bad Parameters (SAC)

4.2 Acrobot

- **Performance & stability:**
  - PPO: In this environment, the overall performance and stability were highly influenced by the following hyperparameters: Higher learning rates initially caused instability but improved performance as the policy stabilized, whereas lower learning rates provided more stable and responsive learning early on. The larger batch size and longer rollout length contributed to more stable gradient estimates over time, though initial instability was noted. Conversely, smaller batch sizes and shorter rollout lengths balanced exploration and exploitation more effectively, with Figure 7 achieving the best scores and long-term performance due to reduced exploration and more refined policy learning.
  - SAC: For this discrete environment, the SAC algorithm achieved good performance even if it takes 2 gradient steps per update, which was done at each environment step. However, even if it is more computationally intensive with 2 gradient steps, it took less time than CartPole because the episode stops when the goal is reached, so it counter-balances this disadvantage. We also increased the exploration importance. Additionally, we observe a low variance after a few episodes, even with a relatively low batch size of 64.

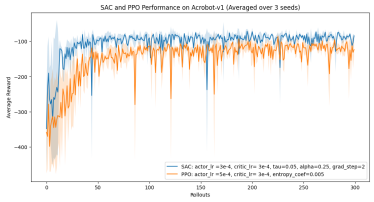

Figure 7: Acrobot: best parameters

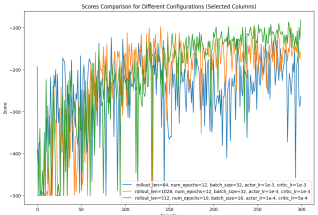

Figure 8: Acrobot Bad Parameters (PPO)

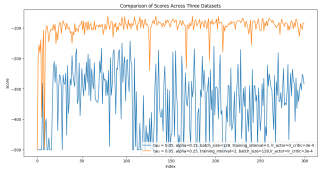

Figure 9: Acrobot Bad Parameters (SAC)

4.3 Mountain Car Continuous

- **Performance & stability:**
  - PPO: In this environment, the performance and stability were significantly impacted by the entropy coefficient and batch size. A lower entropy coefficient led to faster convergence by focusing the policy on exploiting known good actions, despite slightly noisier updates due to the smaller batch size. Poor performance and instability could be seen when using high learning rates and smaller batch sizes, resulting in large fluctuations and unstable training.

– SAC: In this environment, we did not achieve to make the score reach the maximum. We tried to vary the hyperparameters a large number of times, but nothing made the agent learn. We also tried to increase the randomness in the decision by increasing the Gaussian noise from the sampling function, but even with that, it didn't change. We hypothesize that the problem comes from the reward system that makes the agent incapable of learning.

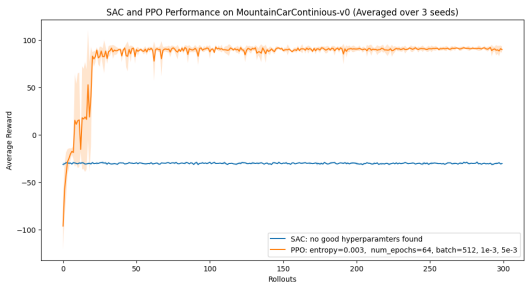

Figure 10: Mountain Car Continuous: best parameters (only PPO)

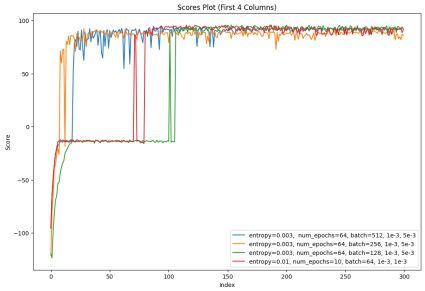

Figure 11: Mountain Car Continuous bad Parameters (PPO)

4.4 Pendulum

- **Performance & stability:**
  - PPO: In the Pendulum environment, using a low entropy coefficient of 0.005 and a value function coefficient of 0.5 promoted faster initial learning by focusing on exploiting known good actions. The numbers of epochs, the rollout length, and the batch size were also important. Choosing larger values enabled better learning, while still never reaching perfect stability and always oscillating between -400 and 0 reward scores.
  - SAC: The first and most important hyperparameter was the training interval. Surprisingly, when taking the value of 1, the agent didn't learn anything. We had to choose a higher update step to significantly increase the score. We hypothesize that the actor should take more information from its most current states to get better feedback, allowing it to adapt better to the environment. Choosing a batch size of 128 makes a good match with that observation because it is directly linked to the explanation we have just given. The soft coefficient was also very decisive in keeping the agent stable and efficient.

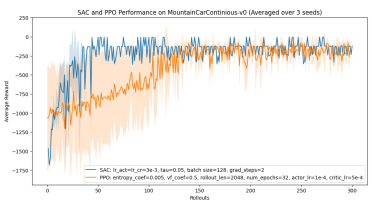

Figure 12: Pendulum: best parameters

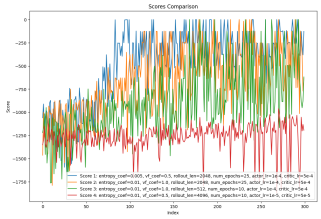

Figure 13: Pendulum Bad Parameters (PPO)

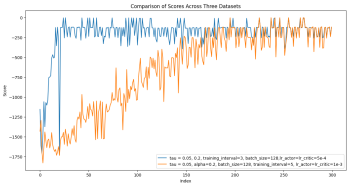

Figure 14: Pendulum Bad Parameters (SAC)

4.5 Summary of Hyperparameters

The overall hyperparameters used across experiments were consistent for comparison. For PPO, two hidden layers were used for both the actor and critic with dimensions of 64. Default values included gamma = 0.99, lambda = 0.95, entropy_coef = 0.01, epsilon = 0.2, vf_coef = 0.5, rollout_len = 256,

total_rollouts = 3000, num_epochs = 10, and batch_size = 16. For SAC, we changed the number of layers and dimensions slightly, but it rarely influenced the results. For the hyperparameters, we generally took 1 or 2 gradient steps to avoid useless and endless training time. We kept the update interval of the targets to one because it wasn't the most influential hyperparameter and didn't change the results much. We usually stayed with a value of 0.2 for the entropy coefficient as proposed in the paper. The other hyperparameters' values depended more on the environment we trained on.

5 Discussion

The results in the previous section indicate that hyperparameter tuning is critical for achieving optimal performance and stability. Higher learning rates can speed up convergence but may cause instability, while lower rates provide stable learning but slow convergence. Entropy coefficients play a significant role in balancing exploration and exploitation, impacting both learning speed and stability. Batch sizes and rollout lengths (in the case of PPO) affect gradient estimates and computational requirements, influencing the overall training process. Furthermore, for the SAC algorithms, one of the main challenges is to find a good balance between performance and computational cost with the numerous hyperparameters.

While SAC's overall performance seems to be better than PPO's, especially as it takes fewer epochs to achieve an optimal solution, it is less stable in environments such as CartPole. PPO, once it achieves optimal performance, remains very stable. Furthermore, we also need to consider not only the number of episodes but also the specific training time. In our setup, SAC took much longer to train, requiring up to 1 hour to train on 3 seeds, whereas PPO managed to do it in under 20 minutes. We can also add that for PPO, initial variance is much higher than SAC, and for some environments, such as Pendulum, it still exhibits high variance compared to SAC which has really low variance.

SAC is more computationally expensive per iteration compared to PPO. This is because SAC requires updating multiple Q-functions and the policy frequently, leading to increased computational overhead. The need for soft updates and entropy calculations further adds to its complexity. PPO stores the policy more compactly compared to SAC. PPO uses a single policy network with an associated value function network, while SAC maintains dual Q-functions and a policy network, increasing the overall storage requirements. SAC scales better for continuous action spaces. Its design, which includes maximizing entropy, allows it to explore the action space more effectively and avoid local minima, leading to better performance in continuous domains. This was evident in the Pendulum environment, where SAC outperformed PPO in terms of stability and final performance. SAC makes efficient use of off-policy data. As an off-policy algorithm, SAC can leverage data collected from previous policies, enhancing sample efficiency and reducing the need for fresh data collection each iteration. This is in contrast to PPO, which primarily relies on on-policy data, limiting its sample efficiency but simplifying its data handling and storage requirements.

Overall, both algorithms exhibited sensitivity to hyperparameters, but the degree and impact varied. PPO was sensitive to learning rate, entropy coefficient, batch size, and rollout length. SAC, on the other hand, was less sensitive to initial hyperparameter settings, providing greater flexibility. Key hyperparameters include training interval, gradient steps, and entropy coefficient, which need careful adjustment to balance computational cost and performance.

This comprehensive analysis highlights the strengths and trade-offs of PPO and SAC, providing insights into their applicability for different RL tasks. The detailed results from each environment emphasize the importance of hyperparameter optimization in reinforcement learning algorithms.

6 Conclusion

In conclusion, the choice between PPO and SAC depends on the specific requirements of the task at hand. PPO is suitable for environments where computational efficiency and straightforward policy storage are critical, and where stability is a primary concern. SAC excels in continuous action spaces and scenarios where leveraging off-policy data is advantageous, despite its higher computational cost and complexity. Both algorithms have their unique advantages and limitations, and their performance can be significantly influenced by hyperparameter tuning. Understanding these aspects can guide the selection of the appropriate algorithm for different reinforcement learning tasks.

References

[1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[2] OpenAI. Spinning up in deep reinforcement learning, 2018.

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