# Comparative Analysis of PPO and SAC for Continuous Control in Locomotion Tasks

**By Nishanth Chockalingam Veerapandian and Sai Nithish Mahadeva Rao**
**chockalingamveerap.n@northeastern.edu and mahadevarao.s@northeastern.edu**

Khoury College of Computer Sciences, Northeastern University
360 Huntington Ave,
Boston, MA 02115

**Code Repository:** https://github.com/saiMahadevaRao/Comparative-Analysis-of-PPO-and-SAC

## Abstract

This study compares two leading reinforcement learning algorithms—Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC)—in continuous control tasks. Using the OpenAI Gymnasium environments Hopper-v4 and HalfCheetah-v4, the project evaluates the rewards achieved and visualizes the locomotion policies via rendered videos. Hopper-v4 emphasizes single-legged balance and forward motion, while HalfCheetah-v4 focuses on maximizing velocity in a multi-joint system.

PPO's clipped objective function provides stable policy updates, while SAC's off-policy learning and entropy-based exploration enable efficient sample usage and potentially diverse solutions. The results showcase the differences in reward optimization and locomotion strategies generated by each algorithm. Through rendered video analysis, this research highlights their practical capabilities in continuous control tasks, offering insights into their performance trade-offs and applicability.

## Introduction

Reinforcement learning (RL) has emerged as a cornerstone in solving complex control problems, particularly in the realm of continuous control tasks. These problems, such as robotic locomotion, require intelligent agents to learn nuanced control strategies that balance stability, efficiency, and adaptability. Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) are two widely recognized algorithms that have shown exceptional promise in addressing such challenges. This project seeks to explore and compare their performance in two benchmark environments: Hopper-v4 and HalfCheetah-v4 from the OpenAI Gymnasium framework.

The Hopper-v4 environment models a single-legged robotic agent tasked with balancing and moving forward, emphasizing stability and energy-efficient control. In contrast, the HalfCheetah-v4 environment simulates a multi-jointed robotic system designed for high-speed locomotion, challenging the agent's ability to optimize velocity while managing control complexity. These environments are particularly suitable for evaluating RL algorithms due to their continuous state and action spaces and their ability to simulate real-world locomotion dynamics.

The motivation for this comparative study stems from the practical need to identify the strengths and weaknesses of different RL approaches in solving continuous control tasks. PPO, with its on-policy learning and clipped objective function, is known for its stable updates and robust policy performance. SAC, leveraging an off-policy framework and entropy-based exploration, offers sample efficiency and the potential for discovering diverse control strategies. By comparing these algorithms through reward metrics and rendered video analyses, we aim to provide actionable insights into their suitability for various control scenarios.

This project addresses two key research questions:

1. How do PPO and SAC compare in terms of performance, measured by rewards obtained, in environments with differing complexities?
2. What qualitative insights can be drawn from the locomotion behaviors rendered in video outputs?

By tackling these questions, this study contributes to a deeper understanding of the practical applications and trade-offs between state-of-the-art RL algorithms in the domain of continuous control.

## Background

### 1. Environments

This study uses two benchmark environments from the OpenAI Gymnasium framework: **HalfCheetah-v4** and **Hopper-v4**. These environments simulate robotic locomotion tasks with continuous state and action spaces, making them ideal for evaluating reinforcement learning algorithms.

### 1.1 HalfCheetah-v4

The HalfCheetah environment simulates a multi-jointed agent resembling a two-legged creature moving forward. The task requires optimizing stability, velocity, and energy efficiency.

**Action Space** The action space consists of six torques applied to the joints, each bounded within the range $[-1, 1]$:

- Back thigh rotor: Torque controlling the backward and forward motion of the back thigh.

- Back shin rotor: Torque controlling the flexion and extension of the back shin.
- Back foot rotor: Torque controlling the rotation of the back foot.
- Front thigh rotor: Torque controlling the backward and forward motion of the front thigh.
- Front shin rotor: Torque controlling the flexion and extension of the front shin.
- Front foot rotor: Torque controlling the rotation of the front foot.

Positive torque values move the corresponding joint in one direction (e.g., forward rotation for a thigh or extension for a shin), while negative values move it in the opposite direction (e.g., backward rotation for a thigh or flexion for a shin). These torques must be coordinated to achieve smooth locomotion.

**Observation Space** The observation space is a 17-dimensional continuous vector capturing the agent's state. Key features and their ranges include:

- Positions: $z$-coordinate of the front tip and joint angles, bounded within $[-\infty, \infty]$.
- Velocities: Angular velocities of the back and front joints (thigh, shin, foot), bounded within $[-\infty, \infty]$.

**Rewards** The reward structure consists of:

- **Forward Reward**: Proportional to the forward velocity of the agent, incentivizing faster movement.
- **Control Cost**: Penalizes the magnitude of actions to encourage energy-efficient locomotion.

The total reward is calculated as:

$$\text{reward} = \text{forward\_reward} - \text{ctrl\_cost}.$$

### 1.2 Hopper-v4

The Hopper environment simulates a single-legged robotic agent tasked with achieving stable forward motion. This requires balancing, energy efficiency, and adaptability.

**Action Space** The action space consists of three torques applied to the joints, each bounded within the range $[-1, 1]$:

- Thigh joint: Torque controlling the forward and backward rotation of the thigh.
- Leg joint: Torque controlling the flexion and extension of the leg.
- Foot joint: Torque controlling the rotation of the foot.

Positive torque values move the corresponding joint in one direction (e.g., forward for a thigh or extension for a leg), while negative values move it in the opposite direction (e.g., backward for a thigh or flexion for a leg). The agent uses these actions to dynamically balance and propel itself forward.

**Observation Space** The observation space is an 11-dimensional continuous vector containing features with the following ranges:

- Positions: $z$-coordinate of the top (representing the agent's height) and joint angles, bounded within $[-\infty, \infty]$.
- Velocities: Angular velocities of the thigh, leg, and foot joints, bounded within $[-\infty, \infty]$.

**Rewards** The reward structure includes:

- **Healthy Reward**: A fixed reward for maintaining a stable state.
- **Forward Reward**: Proportional to the forward velocity of the agent.
- **Control Cost**: Penalizes excessive torque to encourage efficient movement.

The total reward is computed as:

$$\text{reward} = \text{healthy\_reward} + \text{forward\_reward} - \text{ctrl\_cost}.$$

## 2. Policy Gradient Methods: Shared Foundation for PPO and SAC

Policy Gradient (PG) methods form the theoretical basis for both PPO and SAC. These methods optimize a parameterized policy $\pi_\theta(a|s)$ by directly maximizing the expected return $J(\theta)$. The Policy Gradient Theorem establishes the update rule:

$$\nabla_\theta J(\theta) = \mathbb{E}_t[\nabla_\theta \log \pi_\theta(a_t|s_t) A_t],$$

where $A_t$ is the advantage function. The resulting objective is:

$$L^{PG}(\theta) = \mathbb{E}_t[\log \pi_\theta(a_t|s_t) A_t].$$

Despite its theoretical elegance, traditional PG methods suffer from instability and inefficiency, necessitating advancements like **Trust Region Policy Optimization (TRPO)**, which laid the groundwork for PPO, and **Deterministic Policy Gradient (DPG)**, which evolved into SAC.

## 3. Proximal Policy Optimization (PPO)

**3.1 Trust Region Policy Optimization (TRPO)** TRPO improves the stability of policy gradient updates by solving the following constrained optimization problem:

$$\max_\theta \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t \right],$$

subject to:

$$\mathbb{E}_t \big[ \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \big] \leq \delta,$$

where $\delta > 0$ limits the Kullback-Leibler (KL) divergence between the old and new policies. While theoretically robust, TRPO is computationally intensive, requiring conjugate gradient and line search methods.

**3.2 Proximal Policy Optimization (PPO)** PPO builds on TRPO by simplifying the trust region concept with a clipped surrogate objective. This enables stable policy updates while avoiding the computational complexity of TRPO. The objective is:

$$L^{CLIP}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and $\epsilon$ controls allowable policy changes. PPO is particularly well-suited for on-policy learning in continuous control tasks, delivering competitive performance with efficient updates.

## 4. Soft Actor-Critic (SAC)

SAC combines entropy-augmented objectives with off-policy learning to encourage exploration and enhance sample efficiency. Originating from DDPG and TD3, SAC introduces several innovations:

**4.1 Deep Deterministic Policy Gradient (DDPG)** DDPG adapts deterministic policy gradient methods to deep learning by parameterizing both the policy ($\mu_\theta$) and the value function ($Q_\phi$):

$$J_{\text{DDPG}}(\theta) = \mathbb{E}_{s\sim\rho^\beta}\big[Q_\phi(s, \mu_\theta(s))\big],$$

where $\rho^\beta$ is the state distribution under the behavior policy. While effective, DDPG suffers from overestimation bias and poor exploration due to deterministic policies.

**4.2 Twin Delayed DDPG (TD3)** TD3 addresses DDPG's limitations with:

- **Clipped Double Q-Learning**: Uses two $Q$-value networks to mitigate overestimation:

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i'}(s', \mu_{\theta'}(s') + \epsilon),$$

  where $\epsilon$ adds exploration noise.
- **Delayed Policy Updates**: Updates the policy less frequently than the value networks to improve stability.

**4.3 Soft Actor-Critic (SAC)** SAC introduces entropy regularization to balance exploration and exploitation:

$$J_\pi(\theta) = \mathbb{E}_t[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))],$$

where $\mathcal{H}$ is the policy entropy, and $\alpha$ controls the exploration-exploitation trade-off. SAC also dynamically adjusts $\alpha$ to maintain target entropy, ensuring robust performance in stochastic and continuous control tasks.

# Implementation and Algorithmic Details

## Proximal Policy Optimization (PPO)

PPO improves upon traditional policy gradient methods by introducing a clipped objective that stabilizes policy updates, avoiding drastic changes that could degrade performance.

**Key Components**

**Policy and Value Networks** PPO uses two neural networks:

- **Policy network** ($\pi_\theta$) outputs a Gaussian distribution over actions with a state-dependent mean $\mu_\theta(s)$ and fixed standard deviation $\sigma$.
- **Value network** ($V_\phi$) estimates the expected discounted returns for each state.

Both networks are fully connected and employ ReLU activations.

**Advantage Estimation** PPO uses Generalized Advantage Estimation (GAE) to compute advantages $\hat{A}_t$, balancing bias and variance:

$$\hat{A}_t = \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}, \quad \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t),$$

where $\lambda \in [0, 1]$ controls the trade-off between value function reliance and trajectory returns.

**Clipped Surrogate Objective** The PPO objective constrains policy updates to stabilize training:

$$L^{CLIP}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio.

**Value Function Loss** The value network minimizes the squared error:

$$L^{VF}(\phi) = \mathbb{E}_t\big[(V_\phi(s_t) - R_t)^2\big],$$

where $R_t$ represents the cumulative rewards for each trajectory.

**Entropy Bonus** An entropy term promotes exploration and prevents premature convergence:

$$L = L^{CLIP} - c_1L^{VF} + c_2S[\pi_\theta],$$

where $S[\pi_\theta]$ is the policy entropy, and $c_1, c_2$ are weighting coefficients.

**Implementation Highlights**

- **Normalization**: Advantage estimates $\hat{A}_t$ are normalized for numerical stability.
- **Mini-batch Training**: Trajectories are divided into mini-batches for efficient gradient updates.
- **Multiple Epochs**: Each mini-batch is processed for multiple epochs, maximizing sample utility while clipping ensures stability.

## Soft Actor-Critic (SAC)

SAC combines entropy maximization with off-policy learning, encouraging exploration while achieving robust performance.

**Key Components**

Algorithm 1: Proximal Policy Optimization (PPO)

**Initialize:** Policy network $\pi_\theta$, value network $V_\phi$, and replay buffer $\mathcal{D}$
**for** each iteration **do**
  Collect trajectories using $\pi_\theta$ and store in $\mathcal{D}$
  Compute rewards-to-go $R_t$ and advantages $\hat{A}_t$ using GAE
  **for** each epoch **do**
    Sample mini-batches from $\mathcal{D}$
    Update $\pi_\theta$ using the clipped surrogate objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big]$$

    Update $V_\phi$ by minimizing:

$$L^{VF}(\phi) = \mathbb{E}_t\big[(V_\phi(s_t) - R_t)^2\big]$$

  **end for**
**end for**

**Maximum Entropy Objective** SAC optimizes a policy to maximize both expected returns and entropy:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\big[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\big],$$

where $\mathcal{H}(\pi)$ is the entropy, and $\alpha$ is a temperature parameter dynamically adjusted to maintain target entropy.

**Actor-Critic Architecture**

- **Actor Network**: Outputs a stochastic policy $\pi_\theta(a|s)$, parameterized as:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)).$$

- **Critics**: Two Q-value networks $Q_{\phi_1}$ and $Q_{\phi_2}$ estimate action-state values to mitigate overestimation:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s')}\big[(Q_{\phi_i}(s,a) - (r + \gamma(Q_{\phi_i'}(s',a') - \alpha\log\pi_\theta(a'|s'))))^2\big]$$

**Automatic Temperature Tuning** The temperature parameter $\alpha$ is updated to regulate exploration:

$$L(\alpha) = \mathbb{E}_{a_t\sim\pi}\big[-\alpha(\log\pi(a_t|s_t) + \mathcal{H}_{\text{target}})\big].$$

**Implementation Highlights**

- **Replay Buffer**: Off-policy samples are stored and reused for efficient training.
- **Critic Updates**: Target Q-values are computed using:

$$Q_{\text{target}} = r + \gamma(1-d)\big(\min_{i=1,2} Q_{\phi_i'}(s',a') - \alpha\log\pi(a'|s')\big),$$

  where $d$ indicates terminal states.
- **Actor Updates**: The actor maximizes the expected Q-value adjusted for entropy:

$$L_\pi = \mathbb{E}_{s\sim\mathcal{D}}\big[\alpha\log\pi(a|s) - Q(s,a)\big].$$

Algorithm 2: Soft Actor-Critic (SAC)

**Initialize:** Actor $\pi_\theta$, Critics $Q_{\phi_1}$, $Q_{\phi_2}$, Target Critics $Q_{\phi_1'}$, $Q_{\phi_2'}$, Replay Buffer $\mathcal{D}$, Temperature $\alpha$
**for** each episode **do**
  Receive initial state $s_0$
  **for** each step $t$ **do**
    Sample action $a_t \sim \pi_\theta(a_t|s_t)$
    Execute action $a_t$, observe reward $r_t$, and next state $s_{t+1}$
    Store $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
    Sample $(s, a, r, s', d)$ from $\mathcal{D}$
    Compute target Q-value:

$$Q_{\text{target}} = r + \gamma(1-d)(\min_{i=1,2} Q_{\phi_i'}(s',a') - \alpha\log\pi_\theta(a'|s'))$$

    Update critics by minimizing:

$$L_Q = \sum_{i=1,2} (Q_{\phi_i}(s,a) - Q_{\text{target}})^2$$

    Update actor by maximizing:

$$L_\pi = \mathbb{E}_{s\sim\mathcal{D}}\big[\alpha\log\pi(a|s) - Q(s,a)\big]$$

    Update temperature:

$$L_\alpha = -\alpha(\log\pi(a|s) + \mathcal{H}_{\text{target}})$$

    Update target networks:

$$Q_{\phi_i'} \leftarrow \tau Q_{\phi_i} + (1-\tau)Q_{\phi_i'}$$

  **end for**
**end for**

# Experiments

## Proximal Policy Optimization (PPO)

**Experimental Setup** We evaluated PPO on the two MuJoCo-based environments from Gymnasium: **HalfCheetah-v4** and **Hopper-v4** mentioned in Background.

**Implementation Details** Two PPO variants were tested:

1. **Base PPO**: Standard implementation with basic advantage estimation, applied to HalfCheetah-v4.
2. **PPO with GAE**: Enhanced version incorporating Generalized Advantage Estimation (GAE) and mini-batch updates, applied to Hopper-v4.

**Architecture** The network architecture for both variants:

- **Actor Network**: State $\rightarrow$ FC(64) $\rightarrow$ ReLU $\rightarrow$ FC(64) $\rightarrow$ ReLU $\rightarrow$ FC(action_dim).
- **Critic Network**: State $\rightarrow$ FC(64) $\rightarrow$ ReLU $\rightarrow$ FC(64) $\rightarrow$ ReLU $\rightarrow$ FC(1).

The policy uses a Gaussian distribution with a fixed standard deviation.

**Hyperparameters**

- **Shared Parameters**:

- Learning Rate ($\alpha$): $3 \times 10^{-4}$
- Discount Factor ($\gamma$): 0.99
- PPO Clip Range ($\epsilon$): 0.2
- Action Standard Deviation: 0.5
- Training Episodes: 1,000
- Independent Trials: 5

- **PPO with GAE Parameters (Hopper-v4)**:
  - GAE Lambda ($\lambda$): 0.95
  - PPO Update Epochs: 10
  - Mini-batch Size: Timesteps/32
  - Timesteps per Rollout: 2,048
  - Value Loss Coefficient: 0.5
  - Entropy Coefficient: 0.01

**Results**

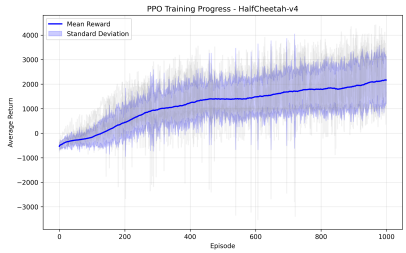


Figure 1: Training curves for PPO on HalfCheetah-v4, showing rewards as a function of training episodes.

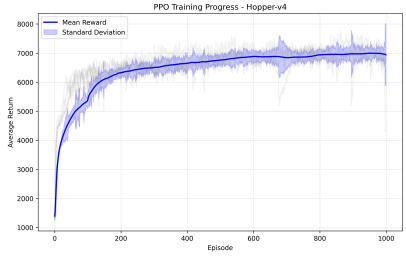


Figure 2: Training curves for PPO on Hopper-v4, showing rewards as a function of training episodes.

**Quantitative Analysis** Figure 1 and Figure 2 illustrates the training curves for **HalfCheetah-v4** and **Hopper-v4** respectively, highlighting distinct learning patterns observed in each environment:

1. **Hopper-v4**:
   - Returns increased rapidly from approximately $1,000$ to $6,000$ during the initial 400 episodes, demonstrating the agent's ability to learn basic hopping strategies efficiently.
   - After 400 episodes, returns gradually stabilized between $6,000$ and $7,000$, indicating robust policy refinement and consistent hopping behavior.
   - The standard deviation decreased steadily, reflecting improved training stability as the agent converged to a periodic motion pattern.
2. **HalfCheetah-v4**:
   - Initial rewards were negative, starting around $-1,500$, but improved rapidly to approximately $2,000$ within the first 200 episodes, signaling early progress in forward locomotion.
   - Between 200 and 1,000 episodes, rewards increased steadily, eventually stabilizing near $2,500$, albeit with higher variance compared to Hopper-v4.
   - The higher variability suggests the presence of multiple local optima, as the agent explores different locomotion strategies.

**Behavioral Analysis - Inference from video**

1. **HalfCheetah-v4**:
   - Average running time of 30 minutes per trial.
   - 60% of trials (3/5) demonstrated consistent forward locomotion.
   - 20% (1/5) converged to a reverse locomotion strategy.
   - 20% (1/5) failed due to stability loss, resulting in tumbling behavior.
2. **Hopper-v4**:
   - Average running time of 30 minutes per trial.
   - 20% of trials (1/5) achieved sustained hopping for the full episode duration.
   - 40% (2/5) maintained stability for part of the episode but failed due to balance loss.
   - 40% (2/5) were unable to establish initial stability.

## Soft Actor-Critic (SAC)

**Experimental Setup** SAC was also evaluated on the same two continuous control benchmarks from the MuJoCo suite.

**Implementation Details**

**Training Parameters**

- Learning Rate: $3 \times 10^{-4}$
- Discount Factor ($\gamma$): 0.99
- Soft Update Coefficient ($\tau$): 0.005
- Batch Size: 256
- Replay Buffer Capacity: $1,000,000$
- Training Duration: 1,000 episodes

**Modified Reward Function** To promote stable locomotion, a modified reward function was used:

$$r_{modified} = r_{original} + \begin{cases} -2.0 & \text{if } h < 0 \\ 1.0 & \text{otherwise} \end{cases}$$

where $h$ represents the torso height.

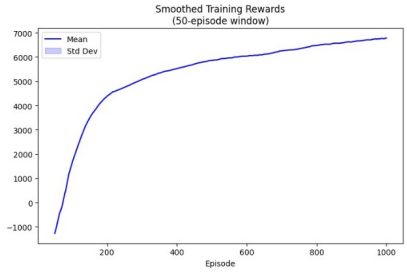


Figure 3: Training curves for SAC on HalfCheetah-v4, showing rewards as a function of training episodes.

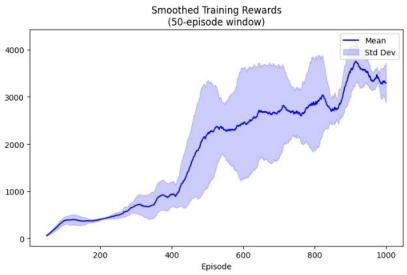


Figure 4: Training curves for SAC on Hopper-v4, showing rewards as a function of training episodes.

**Quantitative Analysis** Figure 3 and Figure 4 illustrates the training curves for **HalfCheetah-v4** and **Hopper-v4** respectively, highlighting distinct learning patterns observed in each environment:

1. **HalfCheetah-v4** SAC achieved consistent progress with three learning phases:
   - Initial Learning (0–200 episodes): Rewards improved from $-2,000$ to 4,000.
   - Intermediate Refinement (200–400 episodes): Steady improvement from 4,000 to 6,000.
   - Final Optimization (400–1,000 episodes): Rewards stabilized at $6,700 \pm 300$.
2. **Hopper-v4** SAC faced challenges in Hopper-v4:
   - Early Stage (0–400 episodes): Rewards remained below 1,000.
   - Transition Phase (400–600 episodes): Sharp increase in performance.
   - Stabilization Period (600–1,000 episodes): Rewards stabilized around $3,500 \pm 800$.

**Behavioral Analysis - Inference from video**

1. **HalfCheetah-v4**:
   - Due to computational intensity, only one trial was conducted for SAC (taking approximately 6 hours)
   - The single trial demonstrated robust forward locomotion with consistent gait patterns
2. **Hopper-v4**:
   - Average running time of 1 hour per trial.
   - for SAC 100% of trials (3/3) achieved sustained hopping for the full episode duration

## Conclusion

Our comparative analysis of PPO and SAC in continuous control tasks reveals distinct performance characteristics and trade-offs between these algorithms. Through experimentation on the Hopper-v4 and HalfCheetah-v4 environments, we have identified several key insights:

**Performance Characteristics** PPO demonstrated robust learning with stable policy updates, achieving consistent performance in the Hopper-v4 environment with returns stabilizing between 6,000 and 7,000. However, its on-policy nature required more samples for convergence. SAC, leveraging off-policy learning and entropy maximization, showed superior sample efficiency and performance stability, particularly in the HalfCheetah-v4 environment, reaching higher final rewards of approximately 6,700 ± 300.

**Computational Requirements** A significant finding of our study concerns the computational demands of these algorithms. For the HalfCheetah-v4 environment, PPO required approximately 30 minutes per trial, while SAC demanded substantially more computation time at around 6 hours per trial. This disparity was less pronounced but still notable in the Hopper-v4 environment, where PPO trials took 30 minutes each compared to SAC's 1 hour per trial. This substantial difference in computational requirements can be attributed to:

- SAC's off-policy nature, requiring maintenance and sampling from a large replay buffer
- The twin critic architecture in SAC, necessitating additional forward and backward passes
- The automatic entropy temperature tuning mechanism, which adds computational overhead
- More complex gradient computations due to the maximum entropy objective

These computational demands significantly influenced our experimental design, particularly for the HalfCheetah-v4 environment, where we conducted fewer trials for SAC (one trial) compared to PPO (five trials).

**Environment-Specific Behaviors** In the HalfCheetah-v4 environment, while SAC achieved impressive results in its single trial with consistent forward locomotion, PPO's multiple trials revealed varied behaviors, with 60% achieving successful forward locomotion. For Hopper-v4, SAC demonstrated superior stability maintenance (100% success in sustained hopping across three trials) versus PPO's mixed results (20% full success rate across five trials). These findings suggest that SAC's entropy-regularized objective provides advantages in maintaining stable policies for complex locomotion tasks, albeit at a significant computational cost.

**Algorithmic Trade-offs** PPO's simplicity, straightforward implementation, and computational efficiency make it an attractive choice for environments where rapid prototyping and multiple trials are desired. However, SAC's sophisticated approach to exploration and value estimation,

while computationally intensive, offers better performance in terms of both final rewards and learning stability. The automatic entropy tuning mechanism in SAC proved particularly valuable in adapting to different environment dynamics without manual parameter adjustment.

**Future Directions** Several promising research directions emerge from this study:

- Development of hybrid approaches combining PPO's computational efficiency with SAC's performance benefits
- Investigation of optimization techniques to reduce SAC's computational overhead while maintaining its performance advantages
- Exploration of parallel implementation strategies to mitigate the computational constraints of SAC
- Analysis of the algorithms' robustness to environmental perturbations and parameter sensitivity
- Research into simplified versions of SAC that maintain its key benefits while reducing computational complexity

This comparative study provides valuable insights for practitioners selecting algorithms for continuous control tasks, highlighting the crucial trade-off between computational resources and performance benefits. The significant difference in computational requirements between PPO and SAC emphasizes the importance of considering both algorithmic performance and practical implementation constraints in real-world applications.

## References


[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[2] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning* (pp. 1861-1870).

[3] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning* (pp. 1889-1897).

[4] Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning* (pp. 1587-1596).

[5] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. *arXiv preprint arXiv:1606.01540*.

[6] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[7] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 32, No. 1).

[8] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.