Analysis of Soft Actor-Critic in BipedalWalker-v3 (Normal and Hardcore Mode)

Sreehari Premkumar
Master's Candidate in Robotics
Northeastern University, Boston 02120
premkumar.sreehari@northeastern.edu

Abstract

This project addresses the challenge of efficiently solving continuous control problems in complex environments, using the BipedalWalker-v3 task as a testbed. The task involves training a virtual agent to walk across a flat terrain in normal mode and navigate randomized obstacles in hardcore mode, providing a diverse set of challenges for reinforcement learning algorithms.

A custom implementation of the Soft Actor-Critic (SAC) algorithm was developed and trained in both modes. The SAC implementation was evaluated and compared with the optimized SAC version from Stable-Baselines3 to understand the performance differences. Key metrics such as average episodic reward, convergence rates, and training stability were analyzed, revealing areas where the custom implementation falls short of the Stable-Baselines3 version due to optimization differences and computational efficiency.

This work highlights the robustness of SAC in handling continuous action spaces while emphasizing the importance of algorithmic and engineering optimizations for achieving state-of-the-art performance in reinforcement learning.

Introduction

The BipedalWalker-v3 task presents a challenging problem in continuous control, requiring an agent to walk stably and efficiently in a high-dimensional continuous action space. The task consists of two modes: normal, where the terrain is flat, and hardcore, which introduces randomized obstacles, ramps, and pits that test an agent's ability to adapt to dynamic and complex environments. These challenges make it an excellent domain to evaluate reinforcement learning (RL) algorithms.

Reinforcement learning is particularly well-suited for such tasks due to its ability to learn optimal policies through trial-and-error interactions with the environment. Among RL algorithms, Soft Actor-Critic (SAC) is a state-of-the-art approach known for its stability and efficiency in continuous action spaces. SAC leverages entropy regularization to balance exploration and exploitation, making it a strong candidate for solving tasks like BipedalWalker-v3.

The objective of this project is to implement a custom SAC algorithm and train it to achieve stable walking on both modes of the BipedalWalker-v3 task. Beyond achieving stable walking, the project aims to analyze the algorithm's training efficiency, examine reward trends over episodes, and compare the custom SAC implementation with the more optimized version from Stable-Baselines3. This comparison highlights the strengths and weaknesses of the custom implementation while uncovering critical insights into the role of algorithmic optimizations in performance.

In summary, this project contributes by (1) implementing SAC for BipedalWalker-v3, (2) analyzing reward trends and training dynamics, and (3) comparing performance with a highly optimized SAC implementation to identify key factors influencing RL success in continuous control environments.

Background

BipedalWalker-v3 Environment

The BipedalWalker-v3 environment is a continuous control problem in which the agent learns to walk using a two-legged robotic structure. Below are the specifics of the action and observation spaces, with a detailed explanation of their ranges and meaning:

Action Space

Type: `Box(-1.0, 1.0, (4,), float32)`

Shape: 4-dimensional continuous space.

Range: Each action dimension is between [-1.0, 1.0].

- **Left Hip Joint Torque**
- **Left Knee Joint Torque**
- **Right Hip Joint Torque**
- **Right Knee Joint Torque**

Key Details:

- Values closer to 1.0 represent maximum positive torque, while values closer to -1.0 represent maximum negative torque. This normalized range ensures compatibility with most reinforcement learning algorithms.
- The agent must apply these torques to maintain balance, move forward, and handle obstacles without falling.

Observation Space

Type:

```
Box([-3.1415927, -5.0, ...],
[3.1415927, 5.0, ...], (24,), float32)
```

Shape: 24-dimensional continuous vector.

The state vector in the environment consists of the following 24 features, grouped into four categories with their respective ranges:

- **Hull-related features (4 features):**

  Range: $[-\pi, \pi]$ for angles, $[-5.0, 5.0]$ for velocities.

- **Joint-related features (8 features):**

  Range: $[-\pi, \pi]$ for joint angles, $[-5.0, 5.0]$ for joint angular velocities, $[0.0, 5.0]$ for joint torques.

- **Ground contact indicators (2 features):**

  Range: $[0.0, 5.0]$ for both legs.

- **Lidar readings (10 features):**

  Range: $[-1.0, 1.0]$ for all 10 readings.

Rewards

The reward function incentivizes behaviors that promote efficient and stable walking while discouraging falling and inefficient energy use:

- **Forward Movement:** Positive reward proportional to the horizontal distance covered by the agent.
- **Energy Efficiency:** Penalty for excessive torque use, encouraging smooth and efficient motion.
- **Falls:** A penalty of -100 is applied when the agent falls, terminating the episode prematurely.

Hardcore Mode

Randomized Terrain: Introduces pits, slopes, stairs, and other obstacles requiring dynamic adjustments.
Objective: The agent must learn adaptive strategies for diverse terrain while maintaining robustness and efficiency.

Soft Actor-Critic (SAC) Details

Replay Buffer

Purpose: To store past experiences in the form of tuples (s, a, r, s', d) for off-policy learning, enabling efficient sample reuse and decorrelation of data.
Structure:

State (s): The 24-dimensional observation vector, stored in a matrix of shape $(\text{max\_size}, 24)$.

Action (a): The 4-dimensional torque values, stored in a matrix of shape $(\text{max\_size}, 4)$.

Reward (r): A scalar value for the current step, stored as a vector of shape $(\text{max\_size},)$.

Next State (s'): The next 24-dimensional observation vector after the action, stored in a matrix of shape $(\text{max\_size}, 24)$.

Done (d): A boolean value indicating episode termination, stored as a vector of shape $(\text{max\_size},)$.

Key Functions:

Store Transition: Adds a transition (s, a, r, s', d) to the buffer. The index is determined in a cyclic manner to overwrite older transitions when the buffer is full.

Sample Buffer: Randomly samples a batch of size B from the stored transitions, ensuring decorrelation of training data. The sampled batch includes:

- States: A batch of shape $(B, 24)$.
- Actions: A batch of shape $(B, 4)$.
- Rewards: A batch of shape $(B,)$.
- Next States: A batch of shape $(B, 24)$.
- Done Flags: A batch of shape $(B,)$.

Benefits:

- **Decorrelation:** Prevents learning directly from correlated samples by randomly sampling from the buffer.
- **Data Reuse:** Improves sample efficiency by reusing past transitions multiple times.

Related Work

Various reinforcement learning (RL) algorithms, including Soft Actor-Critic (SAC), Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), and Twin Delayed Deep Deterministic Policy Gradient (TD3), have been applied to robotic control tasks. Below, I compare SAC with these algorithms and explain why it was chosen for BipedalWalker.

Comparison of SAC with Other RL Methods

- **SAC vs. DDPG:** DDPG is sensitive to hyperparameters and lacks entropy regularization, which can lead to instability. In contrast, SAC improves stability and sample efficiency through entropy regularization, addressing DDPG's shortcomings.
- **SAC vs. PPO:** While PPO is an on-policy method that requires more data and is less sample-efficient, SAC's off-policy nature allows it to reuse past experiences, speeding up learning. Additionally, SAC's entropy regularization helps maintain stable training.
- **SAC vs. TD3:** TD3 improves DDPG with techniques like target noise and delayed updates, but it still faces limitations in exploration and hyperparameter sensitivity. SAC, on the other hand, provides better exploration with entropy regularization, requiring less fine-tuning.

Why SAC Was Chosen

SAC was chosen for the BipedalWalker task due to the following advantages:

- **Sample Efficiency:** SAC's off-policy learning enables it to reuse past experiences, accelerating training.
- **Stable Training with Entropy Regularization:** SAC improves exploration and avoids overfitting, crucial for complex environments like BipedalWalker.
- **Scalability:** SAC effectively handles high-dimensional continuous state and action spaces, making it well-suited for the BipedalWalker task.

Project Description

This project applies the Soft Actor-Critic (SAC) algorithm to the BipedalWalker-v3 environment from OpenAI Gym, where a bipedal robot navigates a dynamic 2D terrain. The agent learns to walk without falling while optimizing movement. I used the hardcore mode, which adds randomized obstacles, making the task more challenging and improving the policy's robustness and generalization.

The environment features continuous state and action spaces, requiring the agent to balance and move efficiently across varying terrains. The hardcore mode introduces additional obstacles to further challenge the agent. Only minor modifications were made, such as adjusting the maximum episodes and evaluation frequency.

SAC Algorithm

Soft Actor-Critic (SAC) is an off-policy reinforcement learning algorithm that optimizes a stochastic policy. It blends traditional stochastic policy optimization with DDPG, incorporating the clipped double-Q trick to achieve target policy smoothing. A key feature of SAC is entropy regularization, which encourages exploration by balancing the trade-off between expected return and policy entropy. This helps prevent premature convergence to suboptimal solutions, enabling more efficient exploration while ensuring robust policy learning.

The SAC algorithm was folowed using the following psedocode from OpenAI Spinning Up:

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 1: Pseudocode for the Soft Actor-Critic algorithm.

Model Architecture and Key Equations

The Soft Actor-Critic (SAC) algorithm follows a sequential process, where different components interact and key equations are applied as the algorithm progresses. Here's a breakdown of the workflow:

Policy Network (Stochastic Policy): SAC uses a policy network $\pi_\theta(a|s)$ parameterized by θ, which outputs a Gaussian distribution for continuous actions. The policy is optimized by balancing the trade-off between expected reward and entropy to promote exploration. The objective function for the policy is:

$$J_\pi(\theta) = E_{s_t \sim \rho^\beta}\left[E_{a_t \sim \pi_\theta}\left[Q_{\bar{\theta}}(s_t, a_t) - \alpha \log \pi_\theta(a_t|s_t)\right]\right] \tag{1}$$

Where:

- $Q_{\bar{\theta}}(s_t, a_t)$ is the Q-value function,
- α is the temperature parameter that controls the trade-off between reward and entropy.

Q-Function Networks (State-Action Value Function): Two Q-functions $Q_{\phi_1}(s, a)$ and $Q_{\phi_2}(s, a)$ are used to estimate the value of state-action pairs. These are updated by minimizing the Bellman backup, which incorporates both the reward and the value of the next state:

$$L_{\phi_i} = E_{(s_t, a_t, r_t, s_{t+1})}\Bigg[\Bigg(Q_{\phi_i}(s_t, a_t) - \Big(r_t + \gamma E_{a_{t+1} \sim \pi_\theta}\Big[Q_{\bar{\phi}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1})\Big]\Big)\Bigg)^2\Bigg] \tag{2}$$

Where:

- γ is the discount factor, and
- α controls the entropy term.

The target Q-value is the sum of the immediate reward and the expected discounted future reward.

Value Network (State Value Function): The value network $V_\psi(s)$ estimates the expected return for a given state, optimized by the following loss function:

$$L_\psi = E_{s_t}\left[\left(V_\psi(s_t) - E_{a_t \sim \pi_\theta}\left[Q_{\bar{\phi}_1}(s_t, a_t) - \alpha \log \pi_\theta(a_t|s_t)\right]\right)^2\right] \tag{3}$$

This loss function ensures that the value function reflects the expected return from each state under the current policy.

Target Networks: To stabilize learning, SAC uses target networks for both the Q-functions and the value network. These are updated via soft updates using the Polyak averaging method:

$$\theta_{\text{target}} \leftarrow \tau\theta + (1 - \tau)\theta_{\text{target}} \tag{4}$$

Where τ is a small coefficient (typically 0.005) controlling the rate of updates.

Training Loop

The SAC algorithm follows these steps in each iteration:

- Sample a batch of transitions from the replay buffer.
- Update the Q-function networks using the Bellman backup equation.
- Update the value network using the value loss function.
- Update the policy network by maximizing the expected Q-value minus the entropy term.

Final Objective

The overall objective is to find the optimal policy π^*, which maximizes the trade-off between reward and entropy. The final objective for SAC is:

$$\pi^* = \arg\max_{\pi} E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1}) + \alpha H(\pi(\cdot|s_t))\right)\right] \tag{5}$$

Where:

- $R(s_t, a_t, s_{t+1})$ is the reward,
- α is the entropy trade-off coefficient,
- $H(\pi(\cdot \mid s_t))$ is the entropy of the policy.

This captures the goal of SAC: maximizing long-term return while encouraging exploration through entropy regularization.

The value function $V^\pi(s)$ includes the entropy bonus at each timestep:

$$V^\pi(s) = E_{a\sim\pi}\left[Q^\pi(s,a) + \alpha H(\pi(\cdot \mid s))\right] \tag{6}$$

The Q-function $Q^\pi(s,a)$ accounts for the expected rewards and entropy over time:

$$Q^\pi(s,a) = E_{s'\sim P, a'\sim\pi}\Big[R(s,a,s') + \gamma\Big(Q^\pi(s',a') + \alpha H(\pi(\cdot \mid s'))\Big)\Big] \tag{7}$$

SAC Training in Practice

SAC trains both the policy network and Q-functions simultaneously. The policy is updated by maximizing expected reward while minimizing the entropy loss. The Q-functions use the Bellman backup equation with entropy regularization, and the value network estimates the expected state value under the current policy.

The implementation includes replay buffer sampling, where transitions are stored and sampled for training. Policy updates maximize the expected reward while ensuring sufficient exploration through entropy regularization. The Q-functions and value networks are updated to ensure they reflect the optimal policy.

Experiment

Normal Mode: SAC vs. Baseline

In this section, we compare our implementation of the Soft Actor-Critic (SAC) algorithm with the Stable-Baselines3 SAC implementation in the normal mode, where the agent is required to walk on flat terrain for 2000 episodes.

Hyperparameters: The following hyperparameters are common for both my SAC implementation and the baseline SAC (Stable-Baselines3):

- Learning Rate: 3e-4
- Batch Size: 256
- Gamma: 0.99
- Tau: 0.005
- Max Buffer Size: 500,000
- Hidden Layer 1 size: 256
- Hidden Layer 2 size: 256
- Entropy Coeff: 0.2(fixed)

Different in Baseline SAC (Stable-Baselines3):

- Entropy Coeff: Auto mode

Results: Figure 2 and Figure 3 presents the aggregate rewards comparison between my SAC implementation and the baseline SAC in normal mode.

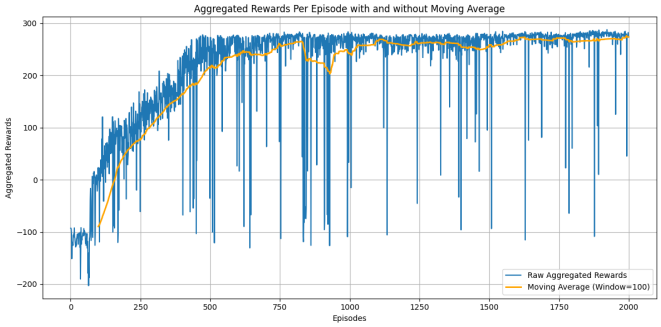

Figure 2: Absolute Rewards(blue) and Moving Average Rewards(yellow) of my SAC implementation in normal mode.

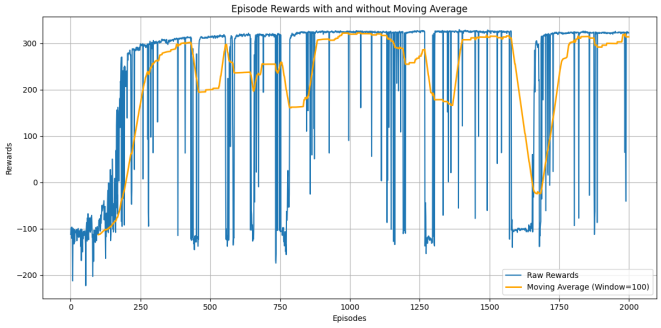

Figure 3: Absolute Rewards(blue) and Moving Average Rewards(yellow) of baseline SAC in normal mode.

Performance Analysis: My SAC implementation consistently averages around 270 rewards, while the baseline fluctuates with occasional dips, ultimately averaging around 320. These dips can be attributed to the baseline's auto-adjusting entropy coefficient, which encourages exploration and can cause temporary performance drops. However, this strategy results in a higher average reward over time, as it explores more effectively than my implementation, which uses

a fixed entropy coefficient for stability but at a lower reward average.

Experiment with Fixed Entropy in Baseline: To test this, I ran the baseline with a fixed entropy coefficient. This reduced reward fluctuations and led to a more stable average of around 310, confirming that the auto-adjusting entropy facilitates exploration, enhancing long-term rewards despite occasional dips.

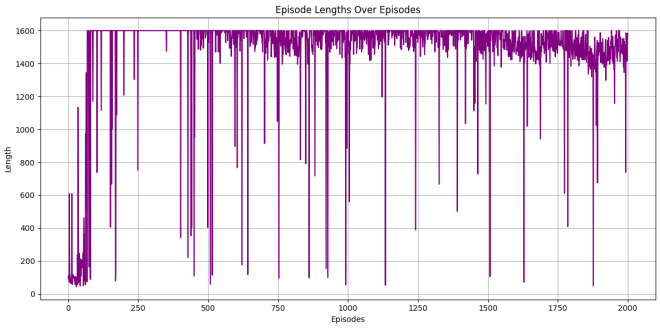

Figure 4: Episode Length of my SAC implementation in normal mode.

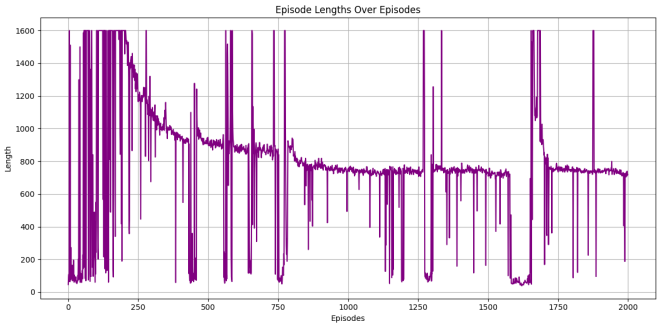

Figure 5: Episode Length of baseline SAC in normal mode.

Episode Length Comparison: In terms of episode length, my SAC implementation consistently runs for about 1600 timesteps, even when the task is completed, indicating slower movement. This suggests that while the agent makes progress, it does so more slowly. In contrast, the baseline SAC reduces its episode length to around 800 timesteps, reflecting faster walking and more efficient decision-making.

Evaluation Results: The speed difference was confirmed during evaluation, where my SAC took 30 seconds to complete the task, while the baseline finished in 14 seconds, consistent with episode length observations.

Training Performance: My implementation took about 9 hours to train for 2000 episodes, while the baseline completed training in 6 hours, highlighting the baseline's faster convergence due to optimizations.

Hardcore Mode: SAC vs. Baseline

The next experiment was conducted in hardcore mode for 500 episodes, where the agent faces randomized obstacles and complex terrain, providing a more challenging environment for both SAC implementations.

Hyperparameters: Most Hyperparameters were same, except

- Hidden Layer 1 Size: 400
- Hidden Layer 2 Size: 300
- My Implemetation Entropy Coeff: 0.1 (fixed)
- Stable Baseline Entropy Coeff: Auto mode

Results: Aggregate rewards comparison between my SAC implementation and the baseline SAC in hardcore mode.

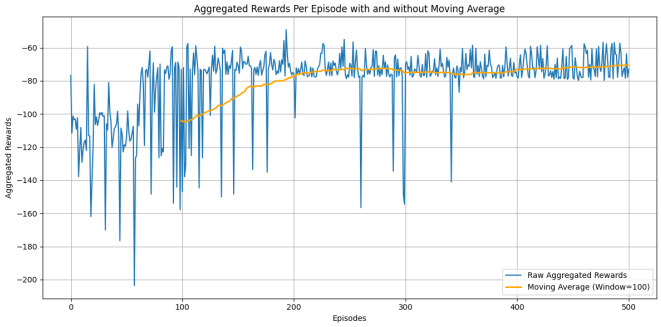

Figure 6: Absolute Rewards(blue) and Moving Average Rewards(yellow) of my SAC implementation in normal mode.

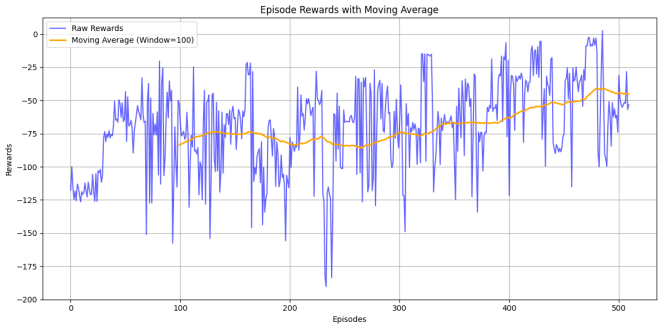

Figure 7: Absolute Rewards(blue) and Moving Average Rewards(yellow) of baseline SAC in normal mode.

Performance Analysis: In hardcore mode, my SAC implementation consistently averages around -80 rewards after 500 episodes, while the baseline fluctuates with occasional dips, ultimately averaging around -50. These dips in the baseline's performance can be attributed to the auto-adjusting entropy coefficient, which promotes exploration and leads to temporary performance drops. Despite these fluctuations, this strategy enables the baseline to achieve a higher average reward over time, as it explores more effectively than my implementation, which uses a fixed entropy coefficient for stability but at the cost of a lower average reward.

Episode Length in Hardcore Mode: In the hardcore mode, both my SAC implementation and the baseline typically reached the maximum timesteps of around 1600 in

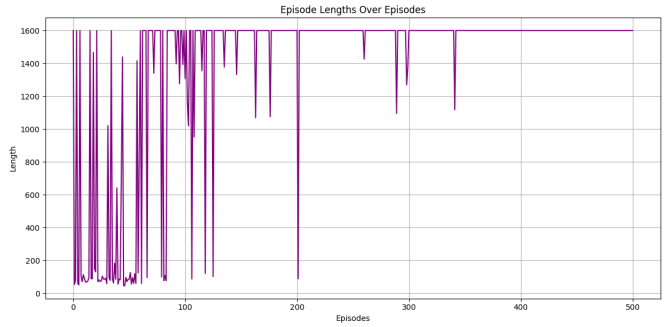

Figure 8: Episode Length of my SAC implementation in Hardcore mode.

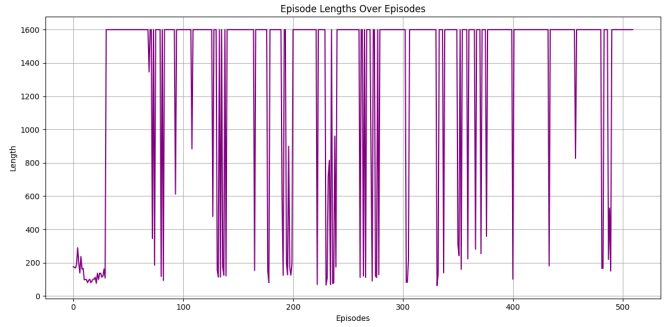

Figure 9: Episode Length of baseline SAC in Hardcore mode.

most episodes. This behavior is consistent with the complexity and challenges posed by the environment, where both agents struggle to complete tasks within the given time constraints.

Behavior in Hardcore Mode

In hardcore mode, the agent's performance was affected by its tendency to play too cautiously. Most of the time, it opted for a "safe" strategy, where it extended its legs flat on the ground to prevent falling and hitting its head. This cautious behavior, although reducing the risk of failure, resulted in slower movements, which ultimately hindered the agent's ability to accumulate rewards.

This behavior demonstrates the trade-off between safety and reward maximization, where the agent's efforts to avoid head collisions and maintain stability resulted in a suboptimal performance in terms of reward accumulation, as well as a longer time to complete episodes.

Conclusion

Key Findings

Our SAC implementation demonstrated stable performance in normal mode, averaging around 270 rewards, with occasional dips due to the fixed entropy coefficient. The baseline SAC, with a dynamic entropy coefficient, achieved higher average rewards (around 320), showing better exploration at the cost of some fluctuations.

In hardcore mode, the SAC implementation reached an average reward of -80, while the baseline outperformed it with an average reward of -50. This difference underscores the baseline's more effective exploration strategy, which allowed it to navigate the complex terrain and obstacles better.

Exploration vs. Exploitation

The key distinction between the two models lies in the exploration-exploitation trade-off. Our SAC's fixed entropy ensured consistent exploration but limited its ability to exploit the environment fully, while the baseline's dynamic entropy struck a better balance, leading to faster convergence and higher rewards, especially in challenging environments like hardcore mode.

Hyperparameter Sensitivity and Efficiency

The baseline SAC's superior performance and faster training times were largely due to optimized hyperparameters, particularly the dynamic entropy. In contrast, our SAC implementation, while more stable, required more training time to reach similar results. Moreover, hyperparameters such as learning rate and batch size played a crucial role in shaping the performance, with smaller learning rates offering greater stability but slower convergence, and larger batch sizes improving training speed at the cost of higher variance.

Overall Insights

This project reinforced the importance of balancing exploration and exploitation in continuous control tasks. SAC's performance is highly sensitive to hyperparameter choices, and while the baseline's dynamic entropy coefficient provided better exploration and adaptability, our fixed entropy implementation offered more stability. This work highlights the importance of fine-tuning hyperparameters to achieve optimal performance in both simple and complex environments.

Limitations and Future Work

The main challenges were SAC's sensitivity to hyperparameters and the high computational cost, particularly in hardcore mode. Future work could focus on improving exploration strategies, scaling SAC to more complex environments like humanoid walkers, and incorporating model-based RL for faster training. Additionally, exploring more adaptive entropy strategies could provide a better balance between exploration and exploitation.