Soft Actor-Critic with Auto-tuned Temperature for Bipedal Walking

Anonymous author

Abstract

This paper implements Soft Actor-Critic(SAC) in both standard and hardcore versions of BipedalWalker-v3 by adjusting the learning rate and reward scale. Both versions achieved over 300 points at episodes 117 and 1296 respectively. While the standard version converged within 200 episodes, it was difficult to determine if the hardcore version converged due to frequent crashes of the NVIDIA CUDA Center (NCC) and limited time. Future work should focus on running the agent in a more stable environment and exploring adaptive learning rate methods to optimize performance in complex environments.

1 Methodology

Open AI Gym's Bipedal Walker Environment is a simulation environment for reinforcement learning in a continuous action space designed to operate a two-legged walker traversing challenging terrain [3]. The agent's objective is to learn to control the walker's actions so that it walks as efficiently as possible without falling over [3]. The agent will receive points for going forward, over 300 points for reaching the end, and -100 points for falling; applying motor torque costs a small number of rewards, and the most optimum agent will accomplish a higher score [3]. The action space of the environment has been shown in table 1.

	Name	**Min**	**Max**
0	Hip 1 (Torque/velocity)	-1	$+1$
1	Knee 1 (Torque/velocity)	-1	$+1$
2	Hip 2 (Torque/velocity)	-1	$+1$
3	Knee 2 (Torque/velocity)	-1	$+1$

Table 1: The action space of BipedalWalker-v3 [3]

The paper proposes using the Soft Actor-Critic (SAC) algorithm, an off-policy maximum entropy deep reinforcement learning method with a stochastic actor, to train a bipedal robot to learn to walk in the BipedalWalker-v3 environment [1]. SAC differs from standard reinforcement learning by incorporating an additional objective of maximizing the entropy of the policy, as shown in Equation 1 [2]. While standard RL aims to maximize the expected sum of rewards $\sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)]$, SAC has additional objective that encourages more explorations, and it can lead to a more diverse and robust policy [2]. In equation 1, $H(\pi(\cdot|s_t))$ refers to the entropy of the policy and is defined as in equation 2, where it represents the expected value of the negative logarithm of the policy [2]. By maximizing entropy, SAC can avoid getting stuck in suboptimal policies and explore a broader range of actions, which is particularly important for continuous control tasks with high-dimensional and continuous action spaces [2]. α in equation 1 controls the trade-off between maximizing the sum of the expected rewards, and the policy's entropy [2]. This is to ensure that the agent not only doesn't fail to learn an effective policy because of an excessive emphasis on exploration but also doesn't become trapped in a poor policy due to an insufficient emphasis on exploration.

$$\jmath(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[r(s_t,a_t) + \alpha H(\pi(\cdot|s_t))\right] \tag{1}$$

$$H(\pi(\cdot|s_t)) = \mathbb{E}_{a\sim\pi(\cdot|s)}[-log(\pi(a \cdot s))] \tag{2}$$

However, it can be difficult to calculate the optimal α, so we implemented auto-tuning α methodology from [1], that predicts the α by using Equation 3, where α_t and a_t is the entropy coefficient and action at time step t, π^* is the current policy and $\bar{H}$ is the target entropy that the policy should aim to achieve, defined as in Equation 4, where $dim(A)$ is the number of possible actions in the environment [1]. The goal of using auto-tuning α is to find the optimal trade-off between exploration and exploitation.

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{a_t\sim\pi^*} \left[-\alpha_t \log \pi_t^*(a_t|s_t;\alpha_t) - \alpha_t\bar{H}\right] \tag{3}$$

$$\bar{H} = -dim(A) \tag{4}$$

SAC uses a combination of a policy network, and a double action-value function (Q-function) to optimize the policy and estimate the target value during training. As shown in Equation 5, the loss function for the critic (Q-function). It tries to estimate the quality of the agent's actions in a given state [2]. The function is defined as the mean squared error between predicted Q-value and the target Q-value in given action s_t and state a_t from replay buffer D. Then we can update the critic parameters by using Equation 6. During the implementation, [1] mentioned they also involved the double Q-learning tricks in improving the model performance by avoiding problems such as overestimating Q-values [4]. We first update two Q function parameters and predict two Q-values, then take the minimum into the policy updating process; in the meantime, we calculate the Mean Squared Bellman Error (MSBE) loss in both Q-values, and the total result of the value will be used to update the policy network. This approach can help the model achieve more accurate Q-value estimates, and better performance [4].

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\frac{1}{2}(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t))^2\right] \tag{5}$$

$$\hat{\bigtriangledown} J_Q(\theta) = \bigtriangledown_\theta Q_\theta(a_t,s_t)(Q_\theta(s_t,a_t) - r(s_t,a_t) - \gamma V_{\bar{\psi}(s_{t+1})}) \tag{6}$$

The network training was done by following Equation 7, which is used to optimize the policy network by minimizing the Kullback-Leibler (KL) divergence between the current policy and the Boltzmann distribution of the Q-function [2]. The goal is to ensure the updated policy can follow the action with the highest probability and Q-value. To make this simpler, [2] used a reparameterization trick to ensure the policy is differentiable so that the policy will be parameterized as $a_t = f_\psi(\epsilon_t; s_t)$ and lead to the final objective function shown in Equation 8.

$$J_\pi(\psi) = \mathbb{E}_{s_t\sim D}\left[D_{KL}(\pi_\psi(\cdot|(s_t)||\frac{\exp(Q_\theta(s_t,\cdot))}{Z_\theta(s_t)}))\right] \tag{7}$$

$$J_\pi(\psi) = \mathbb{E}_{s_t\sim D,\epsilon_t\sim N}\left[\log \pi_\psi(f_\psi(\epsilon_t;s_t)|s_t) - Q_\theta(s_t, f_\psi(\epsilon_t;s_t))\right] \tag{8}$$

In [2], the author discusses the use of reward scaling in the SAC algorithm, where the reward is multiplied by a scaling factor before being used to update the policy and Q-function. This scaling aims to balance the exploration-exploitation trade-off and make the algorithm less sensitive to the reward signal's scale. However, it is essential to note that the reward scale acts as a temperature parameter for the energy-based optimal policy and can significantly affect its stochasticity [2]. Therefore, most of the experiments were based on tuning the reward scaling and getting model performance improvements.

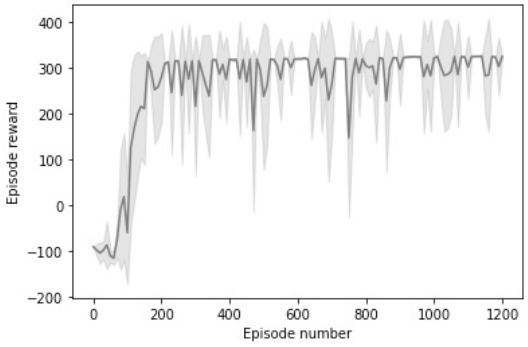

Figure 1: Best result for BipedalWalker-v3

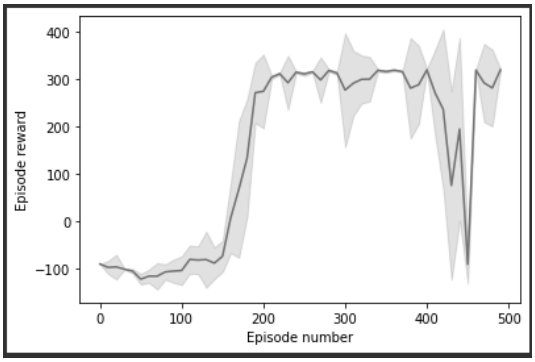

Figure 2: Bad example for BipedalWalker-v3

2 Convergence Results

Figure 1 shows the best result for the BipedalWalker-v3; it first hits over 300 rewards at episode 117 and shows convergence at the beginning of the 200 episodes. However, many experiments and time have been made to reach this point. As [2] suggested, all the experiments were based on tuning the reward scale to make the balance between exploration and exploitation. Most of the experiments show that, even though SAC shows a high level of convergence speed and easy to make good progress, it is hard to maintain the convergence result as shown in Figure 2. The assumption of the reason would be due to the high learning rate and reward scale, making it imbalanced between exploration and exploitation. So in the following few experiments, carefully tuning the value and then maintaining it at convergence score.

Figure 3 shows the best result for the hardcore version of BipedalWalker-v3. The result was achieved when the agent first hit over 300s rewards in episode 1296, and the agent is able to hit over 300 rewards after 1300 episodes. During the experiments, we found that the hyperparameters worked well in the standard version but not suitable in the hardcore version. This is because the hardcore version has more complex situations that have higher requirements for the agent than the standard version to balance exploration and exploitation. We decreased the learning rate and reward scale while increasing the buffer size to facilitate the agent's learning process and ensure that it was able to better comprehend the environment's complexities. In addition, the higher buffer capacity enabled the agent to retain long-term memory of earlier actions, so enhancing its ability to learn from previous behaviour. However, after 2250 episodes, the reward dropped, possibly because the agent continued to explore and reached the memory capacity limit.

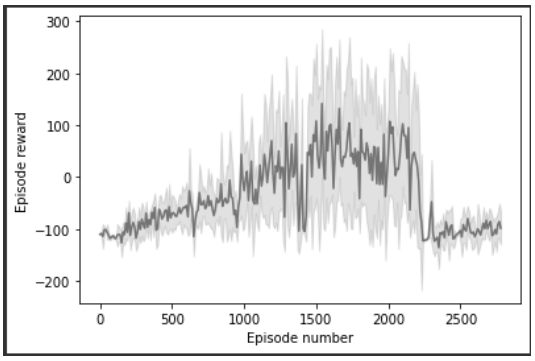

Figure 3: Best result for BipedalWalker-v3 hardcore

3 Limitations

There are several limitations to this approach. Firstly, although the paper [1] mentioned that SAC is relatively stable, it can be time-consuming to tune the model based on the environment. Additionally, the auto-updating α system can be difficult to control in further episodes, even with relatively good hyperparameters. As a result, the agent may still try to explore even after consistently achieving rewards over 300, which can lead to unstable behavior in the system and result in lower rewards. In some experiments, we found that appropriate hyperparameters could mitigate instability in the SAC system, but could not completely eliminate it.

Secondly, the NVIDIA CUDA Centre (NCC) used often crashes, which means that many good experiments cannot be saved and it is difficult to see the final result. This makes it particularly challenging to train the agent in more complex environments, such as the hardcore versions of BipedalWalker-v3. As a result, the complexity and time-consuming nature of tuning hyperparameters and the training process reduced the efficiency and quality of the work.

Future Work

In the future, work should focus on balancing the learning rate and reward scale, particularly in more complex environments like the hardcore version of BipedalWalker-v3. One potential solution could be to use an adaptive learning rate to improve system performance and stability, rather than using a fixed learning rate. Additionally, implementing a auto-save model function during training could help avoid result loss. Finally, investing in a more stable hardware environment that allows for more episodes to be run with each pair of hyperparameters could also lead to better results in future experiments.

References

[1] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[2] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[3] Costa Huang. *Bipedalwalker v2*. Dec. 2020. URL: `https://github.com/openai/gym/wiki/BipedalWalker-v2`.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.