HalfCheetah Trained to Run with Soft Actor-Critic Algorithm

Venkata Tej Kiran Reddy Polamreddy, Ji Liu

May 2023

1 Abstract

Reinforcement learning is a powerful technique for designing controllers of complex robots and it offers an alternative to the traditional control schemes that hinge on the accurate modeling of the physics of the robot. Combined with neural networks and deep learning, reinforcement learning has been proven to be capable of training bipedal and quadruped robots to not only walk but run. In this project, we aim to apply one of the most recently developed reinforcement techniques, i.e., Soft Actor-Critic (SAC), to the training of a virtual dog-like robot, i.e., HalfCheetah, to run. The objective function optimized by the SAC algorithm includes not only the traditional reward but also the entropy of the policy, scaled by the temperature term. By tuning the magnitude of the temperature term, the algorithm balances between exploration and exploitation to find the policy that maximizes the soft value function. We implemented two variants of SAC, where one has a fixed temperature value and the other one can tune the temperature value automatically. We also explored different training schemes and we demonstrate that with SAC HalfCheetah can be trained to run reliably in the virtual environment.

2 Introduction

For real-world robotic applications, designing stable and precise control strategies is one of the biggest challenges facing engineers. One approach is model-based control, where the robot's dynamics need to be faithfully modeled. However, this can be quite challenging especially when the robot has many degrees of freedom and accurately measuring the details of the physics of the robot, e.g., friction within joints, the location of the center of mass, etc., can be no trivial matter. Alternatively, model-free reinforcement learning (RL) offers a control strategy that alleviates the need for accurately capturing these parameters, which can be difficult to measure, and opens up new ways to design controllers for complex robots.

RL generally utilizes the Markov Decision Process (MDP) as the framework for solving these control problems. It assumes an agent that interacts with the environment through its actions, which are chosen based on the observation of the current agent and environmental state. After the action is performed, the agent and the environment transition into potentially new states, and feedback is provided to the agent by the environment in the form of a reward, which is dependent on the previous state and the action chosen by the agent. After receiving the reward, the agent could reevaluate its strategy for choosing actions, i.e., policy, and the process of changing its behavior to maximize the future returns of reward is then referred to as RL. Unlike general supervised learning where the correct answer is provided to the learning agent, RL works by informing the agent whether its action is ”good” or ”bad” and the agent modifies its policy accordingly.

Quite a few new RL algorithms have been proposed over the past 10 years that demonstrated powerful capabilities for solving complex control problems. These algorithms include Deep Q-learning (DQL) [1], Trusted Region Policy Optimization (TRPO) [2], Proximal Policy Optimization (PPO) [3]. These methods often use neural networks for function approximations and can be trained using standard gradient techniques. The algorithm implemented in the current project is Soft Actor-Critic (SAC) [4, 5], and it has several advantages over the previous methods. First, SAC is an off-policy method that can be trained using mini-batches of previously collected data and thus it is more sample-efficient that on-policy methods such as TRPO and PPO. Second, SAC is a maximum entropy RL method that naturally achieves the balance between exploration and exploitation, which is a central theme in RL. Third, the latest version of SAC further offers the ability to tune such balance automatically as the learning progresses such that the agent could settle down for optimal actions in some states while keep exploring in other states where the optimal actions are still unclear [4]. SAC's performance has been shown to be superior to other methods [4, 5], and thus we chose to implement SAC for the current project, where we aim to train a dog-like robot, i.e., HalfCheetah, to run. We show that our implementation works as intended through the proof-of-principle demonstration of the classic inverted pendulum control problem and then we show that HalfCheetah can also be trained reliably to run through SAC.

3 Method

3.1 Environment

For the current project, two virtual environments based in Mujoco were used [6], i.e., "InvertedPendulumBulletEnv-v0" and "HalfCheetah-v2" [7]. The inverted pendulum environment was used only for testing purposes and main goal is to train HalfCheetah, which is a robot constrained in the XZ plane (thus the name "Half"). It consists of 9 links and 8 joints, while torque can only be applied to 6 of the joints and thus the action space is 6-dimensional. The observational space is 17-dimensional, consisting of information regarding the pose and velocity of the robot and its joints. The rewards returned by the environment consist of a positive term for moving forward and a negative term for the control effort. Therefore, the environment encourages the robot to move forward but penalizes large control efforts.

3.2 SAC algorithm

The SAC algorithm aims to solve the following problem [4]:

$$\pi^* = \arg\max_{\pi} \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\,\mathcal{H}(\pi(\cdot|s_t))] \tag{1}$$

where $r(s_t, a_t)$ is the reward for taking action a_t under state s_t and $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the policy $\pi(\cdot|s_t)$. In other words, SAC searches for the optimal policy that maximizes not only the expected reward but also the randomness of the policy such that the policy will keep exploring. The factor α is referred to as the temperature parameter that controls the contribution of the entropy term to the objective function and thus controls the randomness of the optimal policy.

To solve the above problem, a variant of the policy iteration method is used, which alternates between soft policy evaluation and policy iteration. The soft policy evaluation step will converge to the true soft Q-value by repeatedly applying the modified Bellman operator $\mathcal{T}^\pi$ defined as follows:

$$\mathcal{T}^{\pi} Q(s_t, a_t) \triangleq r(s_t, a_t) + \gamma \, \mathbb{E}_{s_{t+1}}[V(s_{t+1})] \tag{2}$$

$V(s_{t+1})$ is the soft value function defined as follows:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] \tag{3}$$

where the second term in the expectation corresponds to the entropy of the policy. In the policy improvement step, information projection is used such that the algorithm searches within a family of Gaussian policies for the one that has the least Kullback-Leibler (KL) divergence with the exponentiated old soft Q-function:

$$\pi_{new} = \arg\max_{\pi' \in \Pi} D_{KL}\left(\pi'(\cdot|s_t) \middle\| \frac{\exp\left(\frac{1}{\alpha} Q^{\pi_{old}}(s_t, \cdot)\right)}{Z^{\pi_{old}}(s_t)}\right) \tag{4}$$

where $Z^{\pi_{old}}(s_t)$ is the normalization factor. The exponentiation of the old soft Q-function is guaranteed to improve the policy.

For practical implementation of the SAC algorithm, neural networks are used for functional approximations of the soft value function, soft Q-function (critic), and the policy (actor). The latest version of SAC [4] only has function approximation for the soft Q-function and the policy while the earlier version of SAC [5] has the additional functional approximation for the soft value function. We chose to implement the version with the soft value function as it seemed to be more stable. Thus, we have the soft value function, soft Q-function, and actor expressed by $V_\psi(s_t)$, $Q_\theta(s_t, a_t)$ and $\pi_\phi(a_t|s_t)$ respectively, where ψ, θ and ϕ are the parameters of the neural networks. The gradient of each of the networks can be calculated from the loss functions defined below.

First, the loss function defined for the value function is as follows:

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}}\left[\frac{1}{2}\left(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}\left[Q_\theta(s_t, a_t) - \alpha \log \pi_\phi(a_t|s_t)\right]\right)^2\right] \tag{5}$$

where $\mathcal{D}$ is the replay buffer and the action a_t is sampled according to the current policy. According to this loss function definition, the target of the soft value function update is the expected value of the Q-function plus the entropy term.

Second, the loss function for the Q-function is defined as follows:

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}}\left[\frac{1}{2}\left(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t)\right)^2\right] \tag{6}$$

where

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \, \mathbb{E}_{s_{t+1}}\left[V_{\hat{\psi}}(s_{t+1})\right] \tag{7}$$

$V_{\hat{\psi}}$ is the target value function with exponentially moving parameter $\hat{\psi}$ such that the training can be stabilized [8]. The above two loss functions can be readily minimized through stochastic gradient descent.

Finally, the loss function for the policy is the following:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}}\left[D_{KL}\left(\pi_{phi}(\cdot|s_t) \middle\| \frac{exp(Q_\theta(s_t, \cdot))}{Z_\theta(s_t)}\right)\right] \tag{8}$$

To perform gradient descent on the above loss function, the original paper [5] used the reparameterization trick such that the gradient with respect to ϕ can be readily calculated.

To reduce the bias in the update of the value function in equation (5), the original paper [5] also implemented two separately parameterized and updated Q-networks, and use the minimum of the two as the update target in equation (5).

The value, critic, and actor networks are approximated using neural networks. All of the networks have roughly the same architecture, where they consist of 2 fully connected layers, and each layer has 256 nodes. For the value and critic network, the output is a single scalar given the state or the state-action pair. The action network's output has double the dimension as the action space, as it outputs a mean and variance from which the Gaussian policy samples the action for each action dimension.

The temperature variable α can be a fixed value, but the more modern implementation of SAC can auto-tune α as the training progresses. The gradient of α can be computed from the following objective function:

$$J(\alpha) = \mathbb{E}_{\alpha_t \sim \pi_t}[-\alpha \log \pi_t(a_t|s_t) - \alpha\,\mathcal{H}] \tag{9}$$

For specific implementations, we utilized the TensorFlow package and coded the neural networks in Python. We also utilized OpenAI's Gym to interface with the Mujoco physics simulation engine.

3.3 Training Setup

The hardware system used for training the SAC algorithm consists of a single Nvidia GTX1080ti with 48GB DDR4 RAM on a AMD 1950X(32 core) CPU. Training has been performed over single running instance of the simulation environment. We have used SAC replay buffer size of 1M samples with 256 samples per training batch.

4 Result

4.1 Validation of SAC implementation

To show that our implementation of the SAC algorithm works as intended, we trained the algorithm on the classic control problem of the inverted pendulum in the virtual environment (with a fixed $\alpha = 1$). The inverted pendulum problem aims to balance the pole in the upright position for as long as possible by pushing the cart horizontally to which the pole is attached. This problem is a small one in that the action space is one-dimensional. In this environment, the agent collects a unit reward for each time step it can keep the pole in the upright position. Figure 1 shows the learning curve of the SAC training with a fixed α value of 1. Over 250 episodes of training, the collected reward in each episode steadily increased, and it achieved 1000 score points by the end of the training, which is the maximum possible score under this environment. Therefore, the agent has learned to balance the pole through the SAC algorithm, and we have shown here that our algorithm is performing as intended.

4.2 Training of HalfCheetah with fixed temperature parameter α

Next, we proceeded to train the HalfCheetah to run with SAC with a fixed α value (Figure 2). Again, the α is the temperature parameter that controls how random the optimal policy is under the SAC objective function formulation. A fixed α value also helps with tuning other hyperparameters such as learning rate and it also serves as a control condition for comparison with later training that has auto-tuned α value.

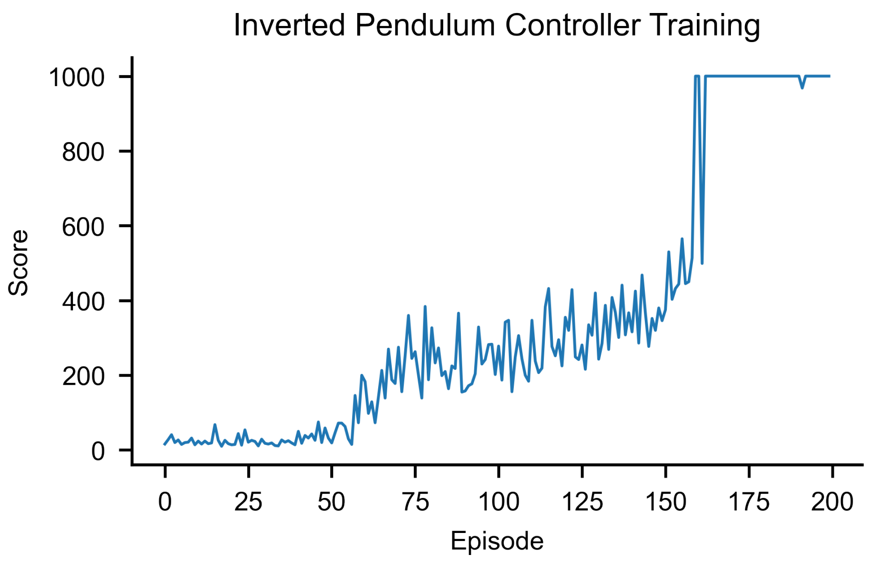

Figure 1: Inverted pendulum problem trained with SAC with fixed $\alpha = 1$.

Applying the exact same algorithm tested on the inverted pendulum, we did not immediately obtain the desired result. After training for 500 episodes, HalfCheetah would either flip over or it would develop some less-than-optimal strategy such as relying only on the front leg to slide across the surface (back flip video, sliding video). We sought to address these issues through the following. First, we more carefully calculated the log probability of choosing a specific action, which is a value calculated repeatedly throughout the learning algorithm. In SAC, the actions are sampled from Gaussian policies, meaning that the action values are not bounded, while the environment usually requires bounded control input within some range. Thus, the SAC algorithm applies a tanh function to the value sampled from the Gaussian policy such that the output is bounded between -1 and 1. Suppose the $\vec{u}$ is the vector of values sampled from Gaussian policy, and the action values $\vec{a}$ are computed by applying the tanh function: $\vec{a} = \tanh \vec{u}$. In such a case, the log probability of outputting the action value $\vec{a}$ given the state s is:

$$\log \pi(\vec{a}|s) = \log p(\vec{u}|s) - \sum_{i=1}^{D} \log\left(1 - \tanh^2(u_i)\right) \tag{10}$$

where D is the action space dimension and u_i is the ith entry of $\vec{u}$. $p(\vec{u}|s)$ is the probability of sampling $\vec{u}$ given the Gaussian policy. The above equation is calculated from the following relationship, which relates the probability density of $\vec{u}$ and $\vec{a}$:

$$\pi(\vec{a}|s) = p(\vec{u}|s) \left|\det\left(\frac{\mathrm{d}\vec{a}}{\mathrm{d}\vec{u}}\right)\right|^{-1} \tag{11}$$

In SAC, each action dimension is sampled from a separate Gaussian distribution, and thus the Jacobian $\frac{\mathrm{d}\vec{a}}{\mathrm{d}\vec{u}}$ is diagonal. To squash the action value between some arbitrary boundary b_l and b_u, the new action value is now calculated as $\vec{a} = \frac{1+\tanh \vec{u}}{2}(b_u - b_l) + b_l$. Again applying equation (11), the log probability of the squashed action is now:

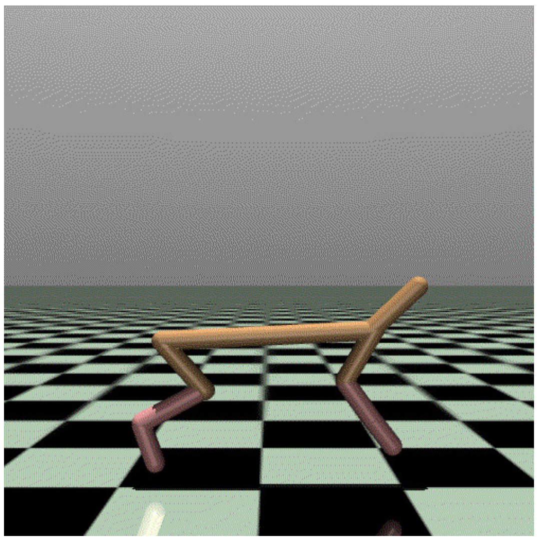
Figure 2: HalfCheetah environment.

$$\log \pi(\vec{a}|s) = \log p(\vec{u}|s) - \sum_{i=1}^{D} \left[\log \frac{b_u^{(i)} - b_l^{(i)}}{2} + \log\left(1 - \tanh^2(u_i)\right) \right] \tag{12}$$

where $b_u^{(i)}$ and $b_l^{(i)}$ are the upper and lower bound of the action of the ith dimension. This concludes the first modification we made to our implementation of SAC, which would adapt the algorithm to arbitrary action range mapping.

The second modification pertains to the training paradigm. One key issue with RL is that the quality of the data collected depends on the quality of the policy, and thus bad policy can only generate bad data, from which policy improvement is unlikely. Therefore, one way to collect data without the influence of a particular policy is to simply apply a uniformly random policy and collect state transition and reward data. As SAC is an off-policy RL algorithm, it can utilize these data for subsequent training. We thus added a data-collecting session to fill the replay buffer before the actual training started. The session consisted of 50 episodes, each with 500 simulation steps.

Combined with these two modifications to the training scheme, we were able to obtain a more reasonable learning curve (Figure 3). The reward collected per episode becomes more positive over time, suggesting that the HalfCheetah is learning to move forward (positive rewards are received for moving forward).

Despite the improvement of the reward collected, since the variability of the reward for each episode remains large, the training is not converged after 500 episodes. In addition, since training HalfCheetah is a rather complex problem (6-dimensional and the observation space is 17-dimensional), training the full-length episode, i.e., with 1000 steps such as shown in Figure 3, took a significant amount of time ($\sim$5 hrs). To explore ways to improve the efficiency of the training, we decided to use shortened episodes (only 30 simulation steps) which in turn allowed us to train more episodes. The rationale behind this choice is that full-length episodes might not always provide useful data throughout the entire episode, since the HalfCheetah could get stuck in a pose from which it cannot recover forward speed, especially since the policy is still improving and less than optimal. Allowing the HalfCheetah to reset more frequently could provide a way to prevent this scenario from happening, which wastes simulation time. A second reason is that since a fixed α is used, the level of exploration is kept constant throughout the training. Thus, in the full-length episodes, the HalfCheetah

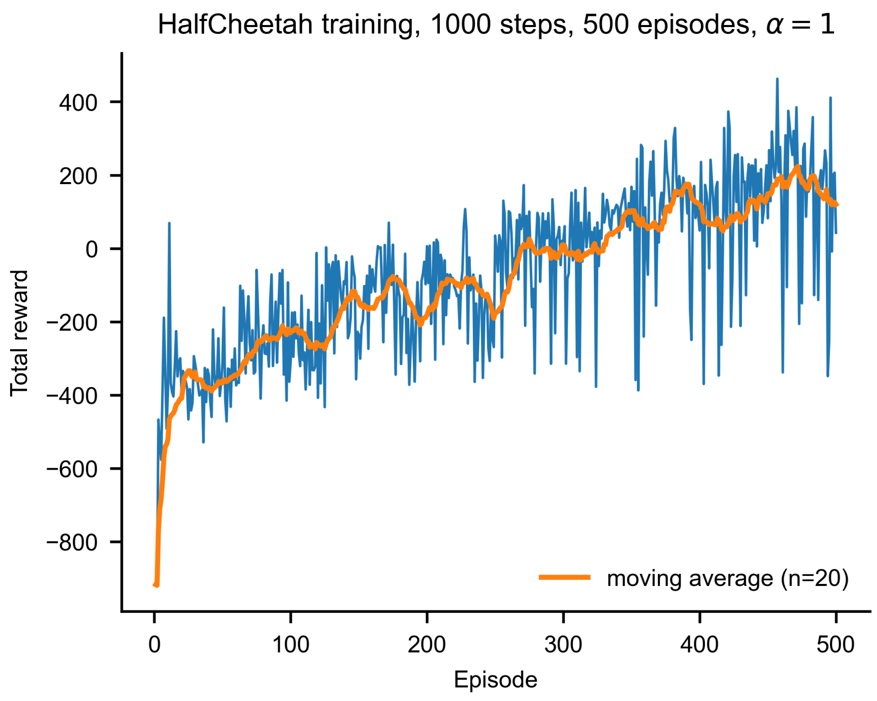

Figure 3: HalfCheetah trained with SAC with fixed $\alpha = 1$. Each episode consisted of 1000 simulation steps.

might have too many exploratory steps, increasing the chance of unrecoverable poses. A third reason is that the running behavior is repetitive and our virtual environment consists only of the ground plane. Thus, the most important aspect for moving forward is to get into the gait cycle, which happens at the beginning of the simulation, and maintain the gait. Therefore, once HalfCheetah manages to get into and remain in the gait cycle for some time, further increasing the episode length might not produce more valuable data. Given these reasons, we decided to continue training the network but with shortened episodes.

The result of this approach is shown in Figure 4. Here we used 2 sessions of shortened episode training, with each having 1000 episodes. The reward per simulation step increased significantly over the first 2000 episodes despite the shortened episode (Figure 4, orange curve), and the performance of HalfCheetah was considerably better than the end of the full-length episode training. In the second session (Figure 4, green curve), the reward kept increasing but only marginally. Nevertheless, these results suggest that the shortened episode could provide a more efficient alternative training scheme for HalfCheetah. The demonstration for the trained HalfCheetah can be found here.

4.3 Training of HalfCheetah auto-tuned temperature parameter α

The key of the SAC algorithm is the balancing of exploration and exploitation through the control of the temperature parameter α. In the previous section, we demonstrate that with a fixed alpha, the RL algorithm achieved reasonable results. However, the more recent version of SAC [4] offers the ability to auto-tune the α value throughout training such that the entropy of the policy is greater than a minimum value. By constraining the entropy level, the algorithm ensures that the agent keeps exploring. In addition, it also allows a variable level of randomness in the policy across different states. This is due to the entropy definition, which takes

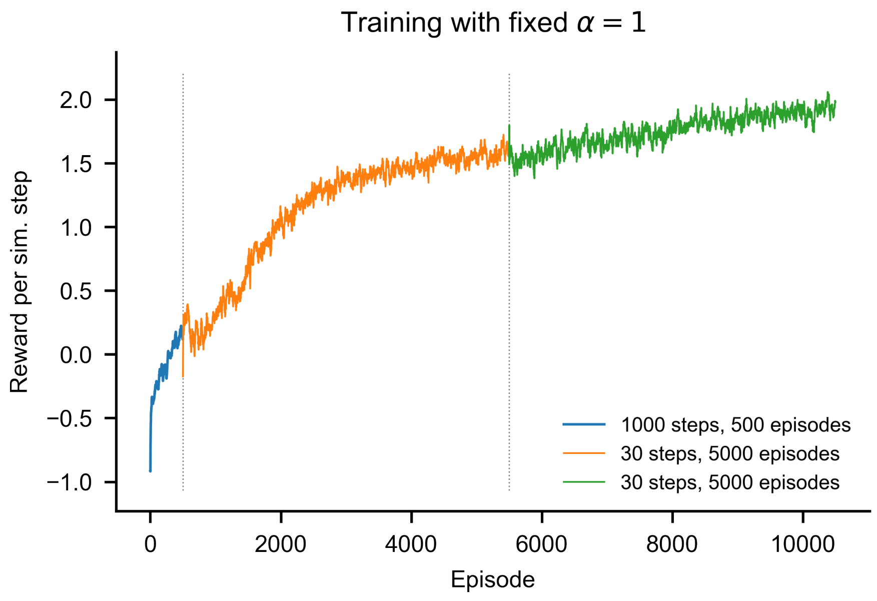

Figure 4: HalfCheetah trained with SAC with fixed $\alpha = 1$. The blue curve represents the first 500 episodes, each with 1000 simulation steps (same data from Figure 3). The orange and blue curves represent two additional training sessions. In each session, 5000 episodes were used, and each episode only has 30 simulation steps. All curves were smoothed with a window size of 20. The reward was normalized by the number of simulation steps for comparison purposes.

expectation over all states. In other words, for a specific state for which an optimal action has been found, the randomness for choosing all possible actions can be low such that with a high probability the optimal action is chosen, while for other states where the optimal actions are not clear, the randomness for choosing different actions is still high to encourage exploration. Thus, this formulation provides more flexibility by assigning non-uniform randomness within the policy and could provide an advantage in training the agent.

Figure 5 shows the result with auto-tuned α training with full-length (1000 steps) episodes. $alpha$ had an initial value of 1 and is allowed to change automatically. The total reward collected per episode increased significantly over time. Most importantly, the reward collected was much greater compared with fixed α training (compare with Figure 3). This suggests that an auto-tuned α indeed facilitates training as previously hypothesized.

Next, we tried the same approach as outlined in the previous section with shortened episodes to see if we could further improve the training efficiency. The result is shown in Figure 6, which consisted of 3 training sessions with different episode lengths. At the beginning of each session, the α value is reset to 1 such that the exploration level is increased at the beginning of each of these sessions. In the first session of full-length episode training, auto-tuning α significantly outperformed fixed-α training, as noted previously (Figure 6, blue curve). In the second session, we switched to using 100-step episodes, as the auto-tuning of α reduced the need to restrict the excessive exploration as seen previously with a fixed α. In the second session, the reward steadily increased but plateaued after about 1000 episodes. As the second session with shorter episodes did not drastically increase the performance of the training algorithm, we tried increasing the episode length to 500 steps in the third session, where we saw another boost in the performance (6, green curve). By the end of the third session, the reward per simulation step is close to 6, which is roughly 4 times greater

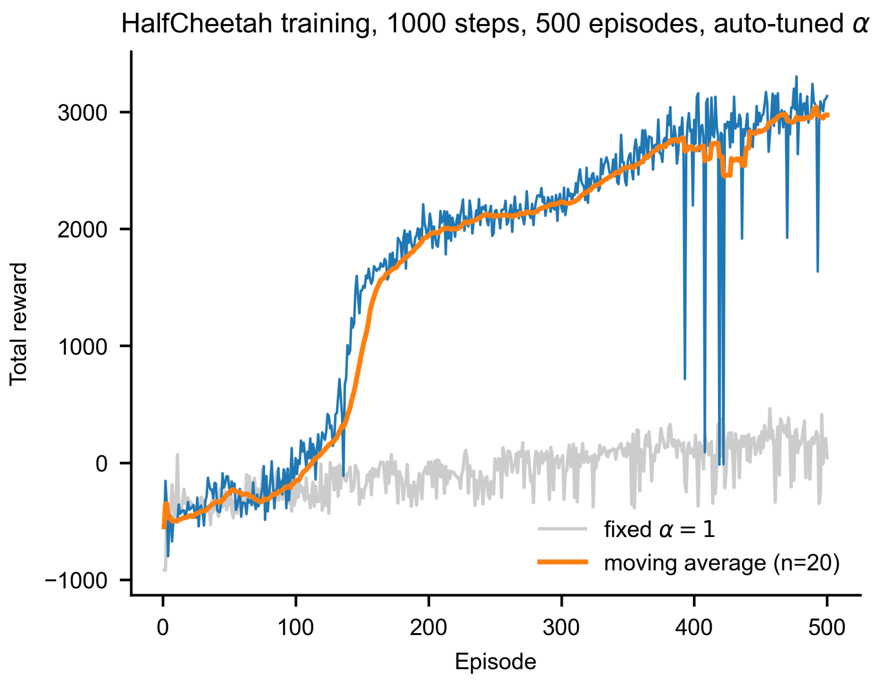

Figure 5: HalfCheetah trained with SAC with auto-tuned α. All episodes are full-length, i.e., 1000 simulation steps.

than that achieved with fixed α training. Although it is not quite conclusive whether changing the episode length indeed made the training more efficient with auto-tuned α, our current data show that longer episodes might be more beneficial for auto-tuned α training, as the first and third session (1000 steps and 500 steps respectively) had the most reward-per-step increase. Therefore, our previous strategy of restricting episode length might be most applicable to just the fixed α case. On the other hand, resetting α at the beginning of the training session might have increased the chance the agent finds a better policy and prevented the agent from being stuck in a sub-optimal policy. A video demonstration of the agent trained with auto-tuned alpha can be found here. Together, we achieved the best training result with HalfCheetah using auto-tuned α.

4.4 α value during auto-tuning sessions

Finally, we looked into how the α value changes during the auto-tuning sessions (Figure 7). During the 3 sessions presented in Figure 6, the α values generally decreased over time, which accompanies the increase in performance in general. These results show that as α decreases, the randomness of the policy decreased as it converges to the optimal policy. When α approaches 0, the algorithm essentially becomes the conventional policy iteration. Therefore, the initial non-zero α value helps the exploration and helps to achieve faster training.

5 Discussion

In the current project, we implemented the RL algorithm Soft Actor-Critic (SAC) [4, 5]. This algorithm extends the conventional Actor-Critic algorithm through the introduction of the entropy term in the objective

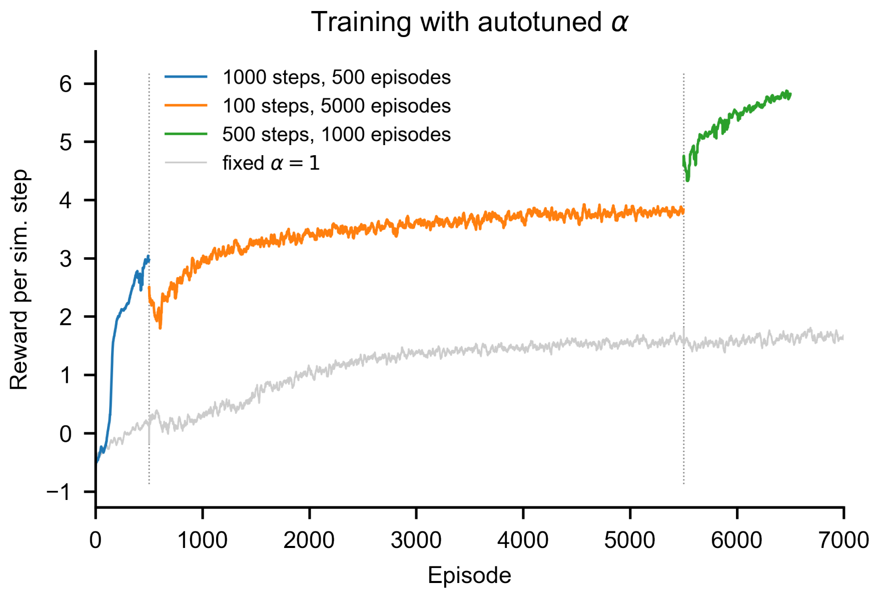

Figure 6: HalfCheetah trained with SAC with auto-tuned α. The blue curve represents the first 500 episodes, each with 1000 simulation steps (same data from Figure 5). The orange and blue curves represent two additional training sessions. In the first session (orange curve), 5000 episodes were used, and each episode only has 100 simulation steps. In the second session (green curve), 1000 episodes were used, with each having 500 simulation steps. All curves were smoothed with a window size of 20. The reward was normalized by the number of simulation steps for comparison purposes. The previous result with fixed α is also plotted in gray (same data as in Figure 4).

function, and at the same time, the value function is also extended to have the entropy of the policy. Therefore, the optimal policy for SAC maximizes not only the expected cumulative rewards into the future but also the randomness of the policy. Therefore, the algorithm tries to achieve the balance between exploitation and exploration, which is a central theme of RL, and it is a concept introduced in the very first problem we encountered in this course, i.e., the bandit problem [9]. In the bandit problem, continuous exploration can be achieved through the ϵ-greedy algorithm, and the magnitude of ϵ determines the probability that an exploratory action is chosen. Here, SAC's policy is intrinsically probabilistic, and the randomness of the final optical policy is determined by the temperature variable α. A large α means a greater weight put on the entropy term in the objective function and thus a more random optimal policy. Thus, by tuning the value of α, one can tune the randomness of the policy. In our implementation of the SAC algorithm with auto-tuned α, we observed that α decreased as the training progressed, which was accompanied by the increase in the reward collected. This suggests that the agent is transitioning into a more deterministic policy as the training progressed, which corresponds to less exploration but more exploitation. In general, we found that the auto-tuned α achieved a much better result than having a fixed α, where the level of exploration is fixed and the agent could get stuck in sub-optimal policy such as sliding. Therefore, auto-tuning α offers an elegant and efficient way to balance exploration and exploitation, which could speed up training, as shown here where the agent's performance increased much faster with auto-tuned α.

To improve the efficiency of our training paradigm, we experimented with the hyperparameters such as the maximum simulation step for each episode. This might be the most relevant for fixed α training, as it is used to largely restrict excessive exploration. With shortened episodes, we achieved a modest increase in the

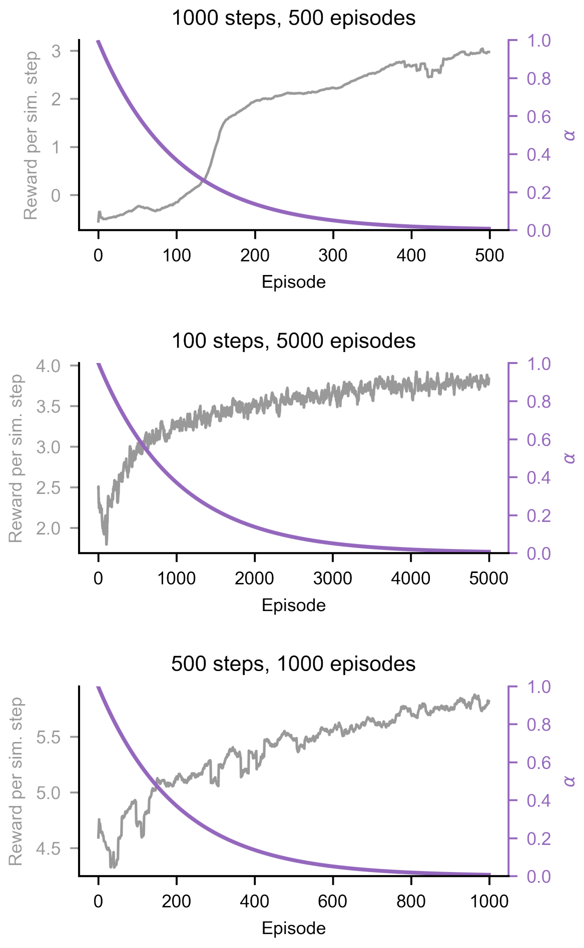

Figure 7: α value changes during the 3 training sessions as presented in Figure 6.

performance with fixed α. In contrast, auto-tuned α does not suffer from the same issue, and it seems that the performance received the most boost when longer episodes are used (6. It provides another justification for simply sticking with the auto-tuned α for training policy for complex robots.

As SAC is an off-policy algorithm, it can reuse the data stored in the replay buffer, and thus increase sample efficiency. To generate data that is not affected by the policy we are training, we added a data collection session before the actual training of the agent networks, i.e., the agent behaves completely randomly. As these state transition and reward data are not dependent on our specific policy, we are less likely to run into the issue where bad policy generates bad data that cannot further increase the agent's performance. Indeed, with the addition of this data collection session, we can achieve a better result even with the fixed α training. Therefore, it seems to be an essential step that can augment the performance of the SAC algorithm.

6 Conclusion

In this project, we successfully implemented the Soft Actor-Critic algorithm and applied it to the training of a 2D dog-like robot to run in a virtual environment. We explored 2 versions of the SAC algorithm and we show that a better performance is achieved with an auto-tuned temperature variable that adjusts the randomness of the policy as the training progressed. We also added a data collection session using a uniformly random policy before the onset of training, which improved the performance of SAC. Together, SAC is a very powerful tool to achieve a model-free controller for complex robots and we intend to apply it to the full 3D quadruped robot to extend our project.

7 Acknowledgements

We would like to thank Prof. Jerry Wu and the TAs, i.e., Akash Ravindra and Akhilrajan for a great semester. For this project, Venkata Tej Kiran Reddy Polamreddy is responsible for implementing the SAC algorithm, testing and training the agent, generating data and video for the report, and the preparation for the presentation and package submission. Ji Liu is responsible for implementing, troubleshooting and tuning the SAC algorithm, generating the figures, preparing for the presentation, and writing the report.

References

[1] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[2] John Schulman et al. “Trust region policy optimization”. In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[3] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

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

[5] 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.

[6] Emanuel Todorov, Tom Erez, and Yuval Tassa. “Mujoco: A physics engine for model-based control”. In: *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2012, pp. 5026–5033.

[7] URL: `https://www.gymlibrary.dev/environments/mujoco/half_cheetah/`.

[8] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[9] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.