INSTITUT POLYTECHNIQUE DE PARIS

Reinforcement Learning

Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor

Thibaut VALOUR, Augustin COMBES

18 février 2023

Figure 1: Dall-E impression : ”Hopper-v0 leg in the style of Edward Hopper”

Table of contents

1 Introduction

Reinforcement learning (RL) is a machine learning paradigm that involves training an agent to learn a behaviour by trial-and-error through interaction with an environment, instead of relying on labelled examples as we commonly do in supervised-learning. RL is commonly used in applications such as robotics, game-playing, and automated decision.

The idea of RL dates back to the 1950s with the work of Richard Bellman, who introduced the concept of dynamic programming as a way to solve optimal control problems. In the 1970s, the RL paradigm was further developed with the introduction of the temporal difference learning algorithm, which allowed agents to learn from experience in real-time.
In the following decades, RL was further advanced with the development of various algorithms such as Q-learning, SARSA, and policy gradient methods. These algorithms enabled agents to learn optimal policies for a wide range of tasks.
In recent years, RL has been advanced with the introduction of deep neural networks, resulting in what is known as deep reinforcement learning. This has enabled agents to learn policies for tasks that were previously considered too complex.
One of the challenges in RL is finding a trade-off between exploration and exploitation. This involves balancing the need to explore the environment to discover new information and the need to exploit the information already learned to maximize reward.

To adress this trade-off, several researchers from the Berkeley Artificial Intelligence Research (BAIR) lab presented in August 2018 the Soft-Actor-Critic (SAC) method [2], which extends the concept of maximum entropy RL to continuous action spaces. By adding an entropy term to the policy objective, SAC encourages exploration while ensuring that the policy remains close to a prior distribution over actions. Since its publication, the paper has been cited 4,615 times according to Google Scholar, and is considered an important advance in reinforcement learning theory.

This report will present the SAC method, first through a summary, then an overview of the mathematical theory behind it. After that, we will present the results of our reimplementation of this method in the context of learning a specific task on a simulated gym (OpenAI) environment.

2 Conceptual overview of the paper

2.1 The rise of Off-policy

As mentioned before, this work introduces a new off-policy actor-critic model, SAC (as in Soft-Actor-Critic). The authors claim the SAC results non-only outperformed previous off-policy works, it was also the first to outperform on-policy RL frameworks, challenging (when not beating) them in terms of learning speed for the easier benchmark continuous tasks (Hopper, Walker2d, HalfCheetah) and in terms of performance for the more difficult ones (Ant, Humanoid).

Even though the SAC framework is independent of the chosen model for the Actor and the Critics, we will consider in our review that each function is approached using a MLP network. This also implies that the framework still holds when using more advanced models. Thus, the efficiency of this method can benefit from any breakthrough in deep learning.

The gap of performance relatively to prior works is due to a conceptually different model. Indeed, the SAC model aims to learn the Q-function simultaneously to the actor training in the maximum entropy framework, which incentives the model to find a both elegant and natural balance between exploration and performance.

Actually, it was not obvious to us that SAC is a truly off-policy algorithm, as the paper is mentions that the model's optimization happens as a *"soft actor-critic algorithm through a policy iteration formulation, where we instead evaluate the Q-function of the current policy and update the policy through an off policy gradient update"*. In reality, according to the OpeanAI Spinning Up page[1] and what we have read in online Q/A, SAC addresses a stochastic, entropy-regulated, on-policy process when exploring. Nevertheless, since the policy is updated with data from a buffer obtained with a different policy than the current one, SAC is actually an off-policy algorithm.

Prior approaches directly solved for optimal policy using the optimal Q-function, addressing the problem in what is referred to as *"Q-learning"* in the literature. Q-learning is a model-free, off-policy RL framework in which one first approaches the optimal Q-function using the Bellman equation. This optimal Q-function gives, for each step, the cumulative reward we can expect when following the optimal policy. That is, once this function is learned, we can retrieve the optimal policy by selecting the action that induces the highest Q-value for each time step.

SAC differs from Q-learning by the order in that it trains jointly the actor and the critics. The learning stability of the Q-function is ensured with soft target updates, which prevent the learning process from getting stuck in local optima.

2.2 An educational and pictorial analogy: GAN

While SAC addresses a completely different problem than Generative Adversarial Network (GAN), its process implies several bricks, that can somewhat look like the GAN foundational Generator / Discriminator framework. Indeed, the conceptual novelty in GAN models when they first were introduced in 2014 was having two separate networks training in an adversarial way : the Generator maps a stochastic, un-parametrized gaussian noise to synthetic data, while the Discriminator's goal of identifying real from synthetic samples provide a feedback signal that allows the Generator's output to approach the real data's target distribution.

Likewise, in SAC, the actor and critic networks are updated iteratively, and the critic's goal is to provide an accurate estimate of the expected cumulative reward to guide the actor optimization.

Another common point in optimization is that, as mentioned earlier, the policy learning happens as the Q-function is approaching the optimal one, increasing training stability and eventually yielding better performance for the policy learned by the Actor. In similar fashion, it is a key point in GAN optimization for the Discriminator's discriminative power not to overpower the Generator's generative one, since it leads to learning a suboptimal Generator solution. Also, this balance issue can cause mode collapse, in which the Generator only produces a limited range of inputs because the Discriminator have early on got too performant at identifying whether one sample was coming from the real distribution or from the synthetic one, that could be seen as an over-exploitation consequence.

The paper insists on the importance of approaching the value function in a stable way, using soft-updates that blend the current value network with the target one. This importance of stability also reminds GAN training, where Generator/Discriminator balance is the main issue one have to deal with. Likewise, this work uses stability tricks to ensure stability during learning :

[1] https://spinningup.openai.com/en/latest/algorithms/sac.html

- despite the theoretical redundancy it induces, the state value function and the Q-function are approached using separate networks
- the Q-function is actually approached itself using two separate networks, and selecting the minimum of the two Q-function estimators before eventually performing gradient update over the policy and value models

3 Scientific review of the paper

3.1 Reinforcement Learning Framework

Let S denote the state space and A the action space. At each time step t, the agent observes the current state $s_t \in S$, selects an action $a_t \in A$ according to a policy $\pi(a_t|s_t)$, and receives a scalar bounded reward $r_t \in [r_{min}, r_{max}]$ from the environment. The sequence of states, actions, and rewards (s_t, a_t, r_t, s_{t+1}) forms a trajectory or rollout.

The probability density of the next state s_{t+1} given the current state s_t and action a_t is represented by the transition probability density $p(s_{t+1}|s_t, a_t)$. The expected total reward from time step t onward under policy π and starting from state s_t is given by the state-value function $V^\pi(s_t) = \mathbb{E}_\pi \left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t\right]$, where $\gamma \in [0, 1)$ is a discount factor that trades off immediate rewards against long-term rewards.

The goal of the agent is to learn a policy π that maximizes the expected discounted sum of rewards, also known as the return, over all possible trajectories. One common method for doing so is to use a Q-function, which estimates the expected discounted sum of rewards starting from a state-action pair (s, a) and following policy π thereafter. Specifically, we define the Q-function for a given policy π as $Q^\pi(s, a) = \mathbb{E}_{a_t \sim \pi} \left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t = s, a_t = a\right]$. The optimal Q-function, denoted $Q^{(s,a)}$, is the maximum Q-value achievable from state-action pair (s, a) under any policy. The optimal policy, denoted π, chooses actions that maximize the expected return at each state.

In many RL problems, the state and action spaces are too large to be represented exactly. To address this, we often use function approximation, such as neural networks, to approximate the Q-function or policy. This introduces approximation error, which can lead to suboptimal behaviour. One way to mitigate this issue is to use off-policy algorithms, which estimate the Q-function using a behaviour policy that explores widely, while learning a target policy that exploits the current best estimate of the Q-function.

A key concept in the SAC method is the entropy. For a policy π at a state s_t, it is defined as the expected value of the negative logarithm of the probability distribution of the policy over the action space A, denoted by $\pi(a_t|s_t)$:

$$H(\pi(s_t)) = \mathbb{E}_{a_t \sim \pi}[-\log \pi(a_t|s_t)]$$

The entropy can be interpreted as a measure of the amount of uncertainty or randomness in the policy. A high entropy policy is more exploratory and can help to discover new and potentially better policies, while a low entropy policy is more exploitative and tends to stick to a known good policy.

3.2 Scientific description of the model and its optimisation

We will describe the soft-policy iterative algorithm, which objective is to learn the optimal maximum entropy policy, by consequently evaluating and improving the policy in the maximum entropy framework described earlier. The soft policy iteration converges to the optimal policy inside a set of policies Π (e.g. the ones parametrized by a two-layers perceptron model). The convergence proof of the soft-policy algorithm towards the optimal policy relies on two lemmas: the first one ensures the critic's progress (*Soft Policy Evaluation*), the second one the actor's progress (*Soft Policy Improvement*).

Our global objective in the maximum entropy framework is to maximize the expected entropy of the policy π over $\rho_\pi(.)$, the state marginal of the trajectory following this policy :

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

3.2.1 From Critics...

Let π be any policy. We wish to find a better Q-function to criticize this policy.

Tabular setting: Tackling first the tabular case where $|\mathcal{A}| < \infty$, the center component of this process will be the Bellman operator $\mathcal{T}^\pi$. Denoting $V(s_{t+1}) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - log\pi(a_t|s_t)]$, the action of this operator transforms a given Q-function Q into $\mathcal{T}^\pi Q(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V(s_{t+1})]$.

Let us explain the expression of $V(s_{t+1})$. The expression involves the negative log-probability $-log\pi(a_t|s_t)$, which can be understood as how surprising it is that the policy π takes the action a_t while being in state s_t. Actually, the lower the action's probability when following the policy π, the greater this negative-log-probability term: this term can therefore be seen as a reward for taking actions with low probabilities according to our current policy π.

Next, the additional term is incorporated into the existing action-value function $Q(s_t, a_t)$, which estimates the anticipated long-term reward of taking action a_t from state s_t. This addition involves an exploration term that rewards actions selected from an unexpectedly stochastic sample, as well as an evaluation term that assesses the suitability of selecting action a_t in the current state. Ultimately, $V(s_{t+1})$ is the predicted value of this sum over the action distribution chosen by our fixed policy π when in state s_t. The term $\mathbb{E}_{a_t \sim \pi}[-log\pi(a_t|s_t)]$ can be viewed as an incentive to maintain a broad distribution with multiple modes, instead of having only one area of actions with high probabilities.

Once these intuitions are exposed for the value term, it is easier to understand the expression of the operator. It mixes the current reward $r(s_t, a_t)$ with the soft state value function evaluation, using a factor γ that tunes the preference for the present, and that is an hyperparameter of the model when implementing the algorithm.

Using the Bellman operator, we can start from any function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and iterate the operator until we converge to the optimal soft-Q value of π, as ensured by the **Soft Policy Evaluation** lemma proven in the paper.

Continuous setting: A few things change in the continuous case, that is the context we will use in practice. Given an (approximate) of the continuous policy $\pi_\Phi(a_t|s_t)$, the SAC approximates the soft-Q function with a family of θ-parametrized functions that we will denote as $Q_\theta(s_t, a_t)$. While the state-value is fully defined from the Q-function and the policy, as seen in the equation:

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

we could then define a natural continuous counterpart of the state-value as:

$$\tilde{V}(s_t) = \hat{\mathbb{E}}_{a_t \sim \pi_\Phi}[Q_\theta(s_t, a_t) - log\pi_\Phi(a_t|s_t)]$$

but the paper claims approaching this quantity using an independent ψ-parametrized function $V_\psi(s_t)$ yields to stabler training.

Critics' optimization in the continuous setting thus means addressing two optimizations programs:

- First, the *Soft-value objective*, in which we minimize the squared residual error between the value of our separate model and the values we would expect from the $V(s_t)$ formula:

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

where $\mathcal{D}$ is equivalently the distribution of the previously sampled action and state space or a replay buffer. We can estimate the gradient of this objective in an unbiased way by:

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\theta(s_t, a_t) + log\pi_\Phi(a_t|s_t))$$

where we sample the actions according to the current policy. The existence of such an unbiased estimator is key for SAC's training for the difficult tasks such as the Humanoid benchmark, since it provides justification for sampling the actions using the current policy instead of having to use a replay buffer: this is a very strong quality of SAC's framework.

- Then, the *Soft-Q objective*, in which we minimize a soft Bellman residual:

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

where we fit Q_θ to a continuous counterpart of the discrete operator we used in the tabular case: $\hat{Q}(s_t,a_t) = r(s_t,a_t)+\gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_{t+1})]$. Actually, this is a continuous and random counterpart, since we did not substitute the discrete value function $V(.)$ to its immediate continuous equivalent $V_\psi(.)$, but to a perturbed version of it $V_{\bar{\psi}}(.)$, where the parameters $\bar{\psi}$ are an exponentially moving average of ψ, that the paper claims stabilizes training.

We can then estimate the gradient of this objective by:

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(a_t,s_t)(Q_\theta(s_t,a_t) - r(s_t,a_t) - \gamma V_{\bar{\psi}}(s_{t+1}))$$

3.2.2 ... to Actor

Let Q be any Q-function. We wish to find a better policy π that will lead to better actions, in the sense of Q.

Tabular setting: In this setting, we will update the policy towards the exponential of the Q-function Q. We will do so while constraining the policy to remain in a predefined policy family of functions Π, that is, we will project the updated policy to this space of considered (and tractable) ones using the KL-divergence (see Appendix for clear introduction of the pseudo-metric).

Indeed, we will determine the next policy π_{new} according to the following equation:

$$\pi_{new} = \arg\min_{\pi'\in\Pi} D_{KL}\left(\pi'(.|s_t)||\frac{exp(Q^\pi(s_t,.))}{Z^\pi(s_t)}\right)$$

First, it is blatantly obvious that this equation is interesting if, and only if, the new policy π_{new} performs better than the current policy in terms of expected cumulative reward. The cost function of this optimization problem is:

$$D_{KL}(\pi'(.|s_t)||\frac{exp(Q^\pi(s_t,.))}{Z^\pi(s_t)})$$

which can be understood as a strategy where we find a new policy π' that is close to the current policy π in terms of the actions it selects, while also taking into account the expected value of each action according to the current action-value function Q.

$Z^\pi(s_t)$ is a partition function, that is, a normalization constant that ensures that the probability distribution over actions sums to one. While we could worry about this term being generally impossible to compute, the paper highlights this difficulty is avoidable due to the null contribution of the term to the gradient.

Looking at the equation, we can observe that the right-hand term of the KL divergence is proportional to $\exp(\frac{1}{\alpha}Q^{\pi_{old}}(s_t,\cdot))$, where the temperature parameter α is not explicitly mentioned. This distribution resembles a Boltzmann distribution (i.e., $p_i \propto \exp\frac{-\epsilon_i}{kT}$) commonly used in statistical physics. Interestingly, the paper uses several concepts from statistical physics, yet does not refer to the Boltzmann distribution or physics.

As for the Critics, this optimization step comes with a mathematical guarantee that we are approaching the optimum policy. The **Soft Policy Improvement** lemma proven in the paper ensures that $Q^{\pi_{new}}(.,.) \geq Q^{\pi_{old}}(.,.)$ on the product space $\mathcal{S}\times\mathcal{A}$, that is, we made progress and approached the optimal Q-function.

Continuous setting: Tackling the continuous case, we keep the same notations as the previous subsection on Critics' continuous optimization, and we wish to update the parameters Φ that define the policy function $\pi_\Phi : (a_t,s_t) \to \pi_\Phi(a_t|s_t)$ given fixed state-value function $V_\psi(s_t)$ and soft-Q function $Q_\theta(s_t,a_t)$.

The policy parameters are learned directly by minimizing the expected KL-divergence, and according to the following objective:

$$J_\pi(\Phi) = \mathbb{E}_{s_t\sim\mathcal{D}}[D_{KL}(\pi_\Phi(.|s_t)||\frac{exp(Q_\theta(s_t,.))}{Z_\theta(s_t)})]$$

which is the natural continuous counterpart of the objective presented earlier in the tabular case. The intuitions we gave about the tabular case's objective also hold for this continuous version.

Minimizing this objective function can seem intractable when written under this form. The paper mentions using a "re-parametrization trick" to enhance the "likelihood ratio gradient estimator". We understood this trick was possible due to the fact our target transformation was a continuous transformation of the Q function, which is a differentiable function.

We give in this paragraph an explanation of the re-parametrization trick. First, we can replace the KL-divergence raw expression:

$$\mathbb{E}_{s_t\sim\mathcal{D}}[D_{KL}(\pi_\Phi(.|s_t)||\frac{exp(Q_\theta(s_t,.))}{Z_\theta(s_t)})] = \mathbb{E}_{s_t\sim\mathcal{D}}[\mathbb{E}_{a_t\sim d}[log(\frac{\pi_\Phi(a_t|s_t)}{exp(Q_\theta(s_t,a_t))})]]$$

$$= \mathbb{E}_{s_t\sim\mathcal{D}}[\mathbb{E}_{a_t\sim d}[log(\pi_\Phi(a_t|s_t)) - Q_\theta(s_t,a_t))]]$$

Then, re-formulating the distribution of a_t as $a_t = f_\Phi(\epsilon_t, s_t)$, this becomes

$$\mathbb{E}_{s_t\sim\mathcal{D},\epsilon_t\sim\mathcal{N}(0,1)}[log(\pi_\Phi(f_\Phi(\epsilon_t,s_t)|s_t)) - Q_\theta(s_t, f_\Phi(\epsilon_t,s_t)))]$$

Using this expression, we get the following unbiased objective gradient estimator:

$$\hat{\nabla}_\Phi J_\pi(\Phi) = \nabla_\Phi log\pi_\Phi(a_t,s_t) + (\nabla_{a_t} log\pi_\Phi(a_t|s_t) - \nabla_{a_t}Q(s_t,a_t))\nabla_\Phi f_\Phi(\epsilon,s_t)$$

3.3 Illustrative schema of the framework

We will perform optimization iterating between Actor and Critics optimization steps, and according to the following process.

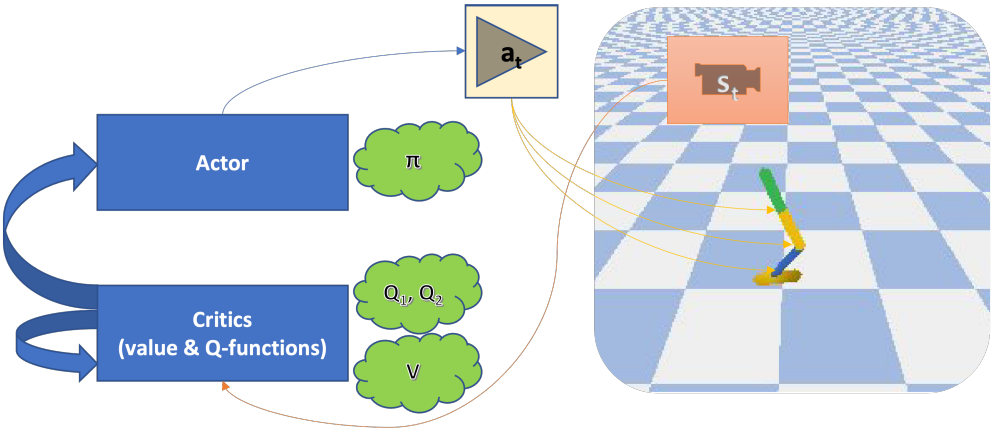

Figure 2: SAC Framework

4 Implementation

We will now present our implementation of the SAC method. Our code, as well as some illustrations and videos of our results, are available on this GitHub repository : `https://github.com/thibautvalour/RL_Soft-Actor-Critic`

4.1 Practical details

For our implementation, we used the Gym toolkit, which is a popular open-source module for developing and comparing reinforcement learning algorithms. Gym provides a wide range of environments, including classic control tasks, Atari games, and robotics simulations, such as those based on the MuJoCo physics engine used by the Authors of the SAC paper [2]. Created by OpenAI, Gym is widely used by researchers and practitioners in the field of machine learning, and it offers a user-friendly API that abstracts away many of the low-level details of environment management. Unfortunately, we had trouble installing the MuJoCo dependencies, so we opted for the PyBullet physics simulator, which is also supported by Gym and provides a diverse set of environments.

4.2 Presentation of our environment

To test our implementation, we wanted an environment that would be complex enough, so our SAC agent has to struggle a bit to master the game, but in the same time it has to learn quickly enough to enable us to run many simulations with different parameters on our device. We chose to focus on the environment 'HopperBulletEnv-v0' which met both these conditions.

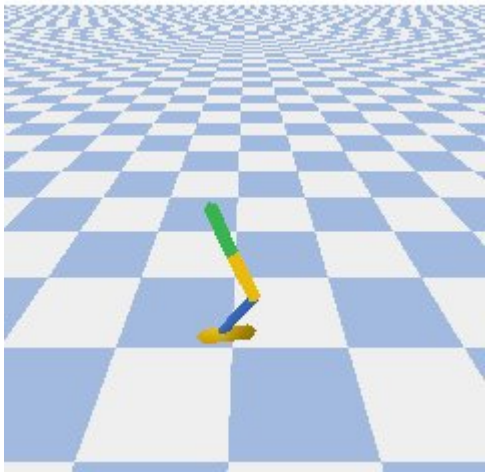

Figure 3: Hopper environment

The hopper is a one-legged, two-dimensional creature that has four primary body parts: a torso at the top, a thigh in the centre, a leg at the bottom, and a single foot on which the body is supported. By adding tension to the three hinges that link the four body components, it is possible to create hops that travel forward (to the right). The objective of the policy is to make the leg walk away without falling.

The leg can freely move in the 2D space, both the observation and action space are continuous. The observation space has 15 dimensions, which are the positions and velocities of each coordinate of the upper hopper point, as well as the joint angles and angular velocities of the different parts of the leg. They evolve in the interval $(-\infty, \infty)$.
The action space has 3 dimensions: the torques applied on the thigh rotor, the leg rot and the foot rotor. They can be chosen in the interval $[-1, 1]$.

The reward is made from several components which aim different objectives. The first is a so-called "healthy reward", which gives a reward for each step the hopper hasn't fallen. This reward incites the hopper to live as long as possible. $r_h = 1$
We also add a forward reward that depends on the distance travelled during the step, so the hopper is incited to walk (or even run) as fast as possible. $r_f = w_f * (x_t - x_{t+1})$
Finally, to make the march more harmonious and to prevent large movements in all directions that would offset each other, we add a penalty term on action norm: $r_a = -w_a * ||a_t||_2^2$

There are two ways to end an episode, or a game. The first possibility is that the leg has fallen off, which means that the z-coordinate of its upper point is below a limit. The second is if the time step limit per episode is reached. In practice, a usual value would be 1000 steps per episode.

4.3 Baseline results

We used the same hyperparameters as in [2] for our first simulation. These are recalled in the appendix. However, it's worth mentioning that our environment wasn't exactly the same since we didn't use the MuJoCo engine, but the PyBullet one.

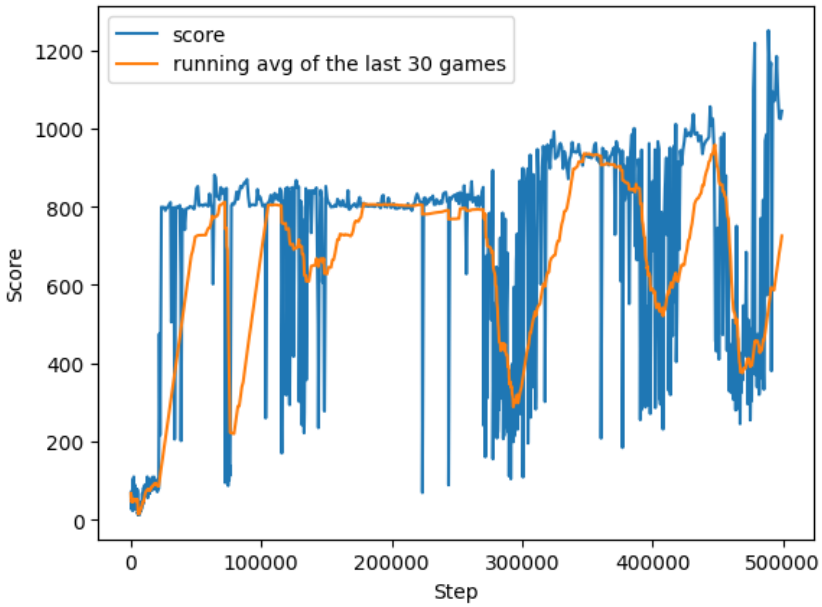

Figure 4: Baseline results

As shown in this plot, the learning process was marked by considerable instability, often yielding near-zero episode scores even after convergence. To enhance the interpretability of the plot, we present the running average of the last 30 games. Our model was capable of learning to walk without falling until almost reaching the maximum step, thus yielding satisfactory results. While we cannot directly compare our curve with the one from the original SAC article [2], as mentioned, the similarity in shape is striking.
These results will serve as our baseline for further analysis.

4.4 Sensitivity analysis

We will perform a sensitivity analysis on some hyperparameters used in the SAC algorithm to understand how they impact the performance of the agent in the Hopper environment. This will help us determine which hyperparameters are most critical and which can be tuned for better performance.

Due to computational complexity, we chose to run our simulations over 500000 steps, instead of the 1000000 steps as in the paper for the *Hopper-v0* task.

4.4.1 Alpha parameter

First, we will examine what happens when we tune the temperature parameter α, which weights the importance between the entropy term and the reward one, inducing more or less stochasticity to find the optimal policy.

It seems that having a smaller temperature means learning faster, but we suspect it also means exploring less during learning, as the stochasticity is tuned down. This confirms the intuition we provided for the alpha parameter being an important hyperparameter to address the exploration/performance dilemma.

Actually, when consulting the numerous implementations of the SAC paper in paperwithcode, we saw posterior papers have been proposing an auto-tuning process for this alpha parameter as for [3], making it an adaptive quantity during the problem instead of a fixed hyperparameter

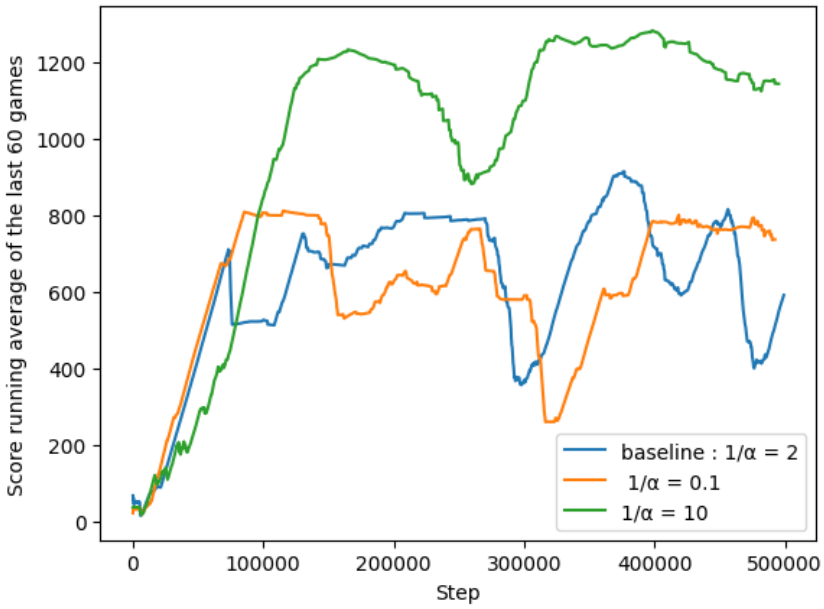

Figure 5: Greater and smaller alphas

4.4.2 Gamma parameter

The parameter γ represents the discount factor in the state-value function $V^{\pi}(s_t) = \mathbb{E}\pi\left[\sum i = t^{\infty}\gamma^{i-t}r_i|s_t\right]$. In the original simulation, this parameter was set to a relatively high value. To assess the impact of γ on the model, we experimented with setting it equal to 1 (although this doesn't make theoretical sense in the equation above) and with lowering it.

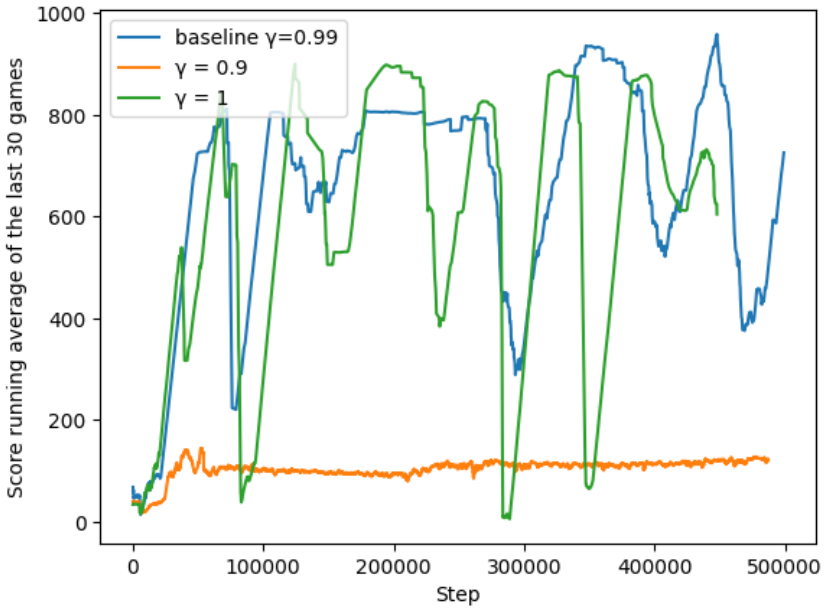

Figure 6: Greater and smaller gammas

The results show that the model's performance is largely unaffected by a gamma value of 1, but when gamma is set to 0.9, the model fails to learn. This is because, with a lower discount factor, the model becomes more inclined to prioritize short-term rewards over long-term benefits, such as choosing to take a risky action that may yield immediate gains but ultimately lead to failure.

4.4.3 Smaller buffer size

The size of the buffer memory is an essential parameter in the SAC algorithm, as it determines the amount of past information that is retained in memory. In an experiment, we decided to reduce this size.

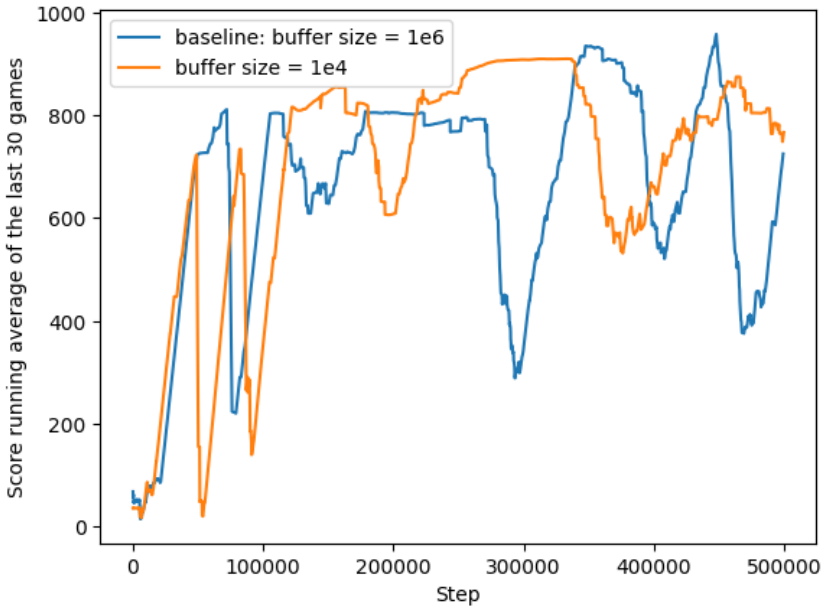

Figure 7: Buffer size from 1 million (baseline), to 10,000

Our hypothesis was that decreasing the buffer size would improve the final performance, since some of the states encountered during the early stages of training when the hopper did not know how to walk at all would not be necessary to retain for the entire learning process, as we will never encounter them again.

To our surprise, we did not observe any significant impact on the final performance or learning speed with the smaller buffer. Although the SAC paper does not provide detailed information on the behaviour of the algorithm for each benchmark task, we conclude that the buffer size may be more critical in learning more challenging problems than the Hopper task.

4.4.4 Tau parameter

The target smoothing coefficient represents the amount of smoothing applied to the target value function (estimate of the expected cumulative reward of a policy) during the training process, and is claimed to reduce the variance of the estimated cumulative reward. A smaller target smoothing coefficient should result in a slower update of the target network, resulting in a smoother target value function.

Again, the target smoothing coefficient can be interpreted as a trade-off between exploration and exploitation. A larger target smoothing coefficient encourages exploration by allowing the agent to more easily deviate from its current policy, while a smaller target smoothing coefficient encourages exploitation by keeping the agent closer to its current policy. Unlike the temperature coefficient that can be adaptively tuned during training in posterior works, the target smoothing coefficient is, according to our researches, typically tuned through trial and error to find a value that balances exploration and exploitation for a particular environment and task and remains an hyperparameter even in more recent works.

In addition to the baseline value of 5.10^{-3}, we tried a much lower value of 10^{-5}. We also attempted $\tau = 1$ in order to remove the soft update. This is referred to in [2] as ”hard target update”.

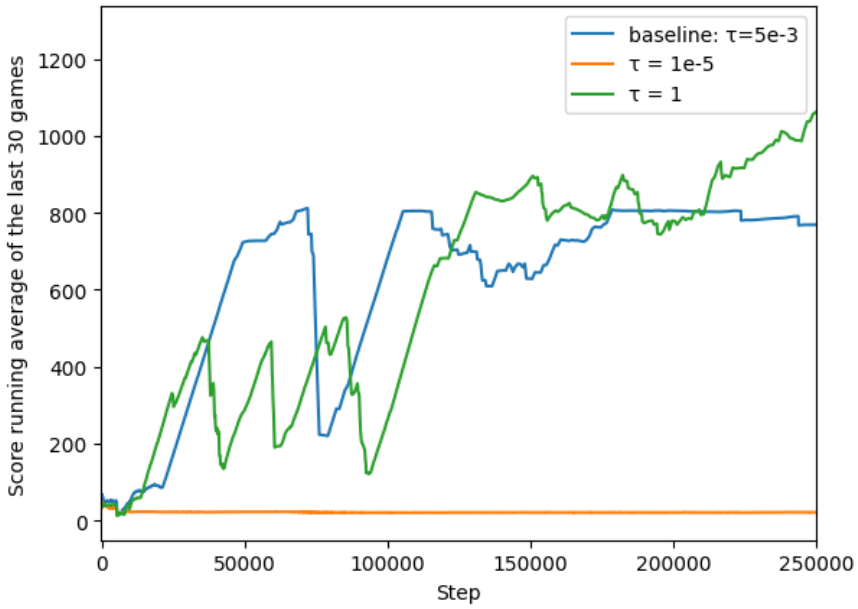

Figure 8: Greater and smaller taus

While we were expecting the performance to drop with a smaller value of τ, we did not expect to observe that the SAC completely fails to learn when encouraging exploration. Indeed, using the smaller value for τ, we can see in the result graph that the score stays at almost null value during the full training process.

On the other hand, we would have expected to see a quicker performance for the SAC training under the hyperparameter $\tau = 1$, as encouraging exploitation should result in better scores at the beginning of training, before the RL model get stuck under the glass ceiling because of the drop in exploration it mechanically induces. In reality, we do not observe such differences, as the over-exploiting model learns slower than the baseline one and end up having a superior reward when stopping at $500,000$ steps.

4.4.5 Conclusion on hyperparameters

To briefly conclude about this sensibility study over hyperparameters, while we did not have complete confidence with our interpretation of each consequence of our experiences compared to our baseline, it allowed us to understand better the practical implications of deviating from the canonical set of hyperparameters. When the observed results did not match our expectations, it was also a reminder that in Deep Learning, we should lead several full training experiments to truly stress-test our intuitions about these parameters, even though it would require having more computational resources than we currently have.

4.5 Ablation study

In addition to the sensitivity analysis, an ablation study was conducted in order to analyse the utility and impact of the key elements of the Soft-Actor-Critic algorithm. As a combination of several Reinforcement Learning ingredients, such as double-Q and soft update of the value function, it was important to understand the individual contribution of each element to the overall success of the algorithm. The study involved systematically removing a given element to observe the effect on the outcome.

4.5.1 Single Q-critic

One of the simplest elements to remove in the implementation of the Soft-Actor-Critic algorithm is the double Q-learning (Silver et al. 2015 [1]). This trick was introduced to reduce the risk of overestimating the value function, which can occur due to bias in the critic function. By having two critic functions that learn independently, the overestimation of the value function can be minimized, and taking the minimum between them can be seen as a form of regularization technique.

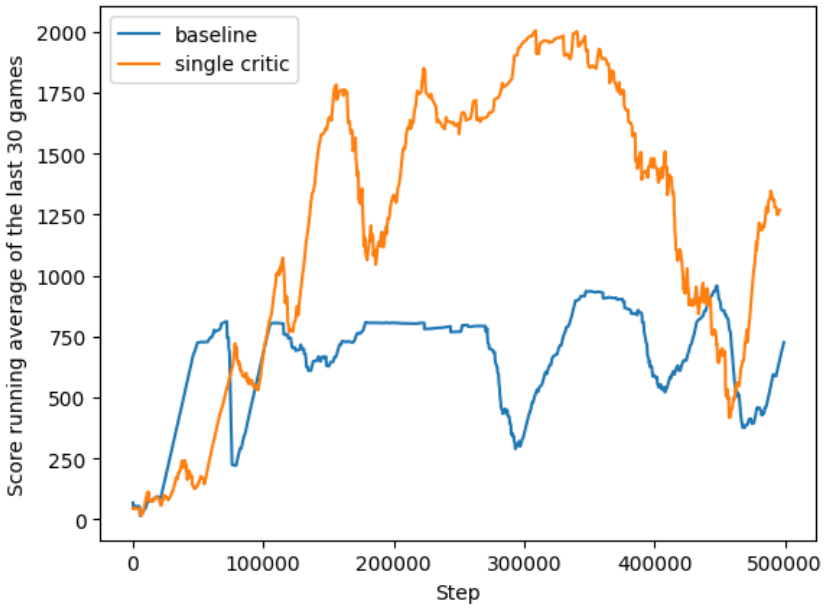

Figure 9: Single Q-learning

However, removing the double-Q learning resulted in unexpected results. The training process seemed to be much easier for the model, and it outperformed the baseline. A possible explanation is that double-Q learning makes the learning smoother but less likely to adapt quickly to a new discovery from only one network. By being cautious about change, the model misses opportunities.

4.5.2 Non-stochastic actions

In the SAC algorithm, actions are produced by sampling from a normal distribution with mean and standard deviation obtained from the policy. This introduces some stochasticity in the process and promotes exploration. To investigate the impact of removing this stochasticity, we tested using the mean of the distribution directly as the action.

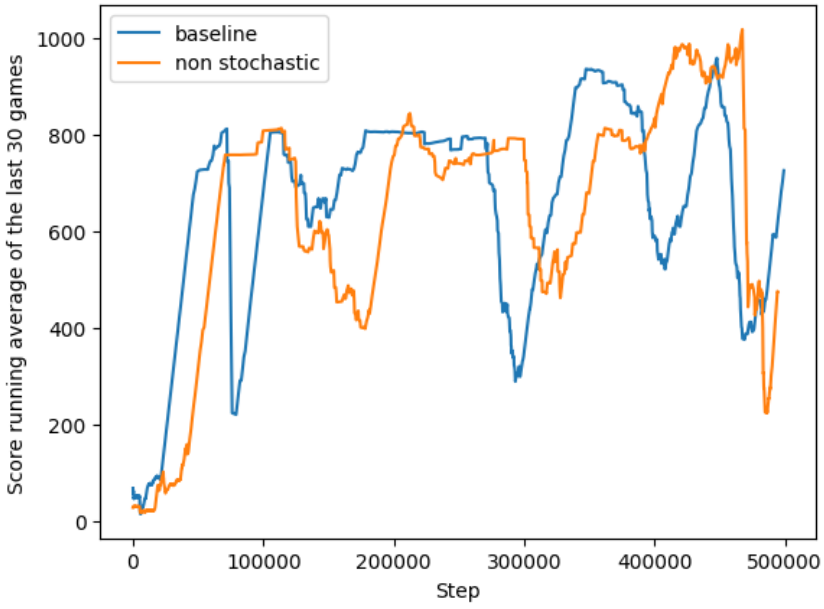

Figure 10: Non-stochastic actions

Our results suggest that removing the stochasticity affects exploration, as our model took slightly longer to converge. However, the scores achieved after the first learning session were similar to the baseline.

Again, the purpose of this study was to test our earlier qualitative intuitions about each component of the SAC. These tests would need to be carried out on other benchmarks and with several simulations for each combination of parameters before any conclusions could be written in stone.

5 Conclusion

We overall liked working on this Reinforcement Learning project. It was satisfying working on such a fundamental RL paper, that introduced a breakthrough in the way we currently address learning. Even though the SAC model was introduced in 2018, it is still a competitive approach to tackle continuous problem.

We understood as we were advancing in the surrounding SAC literature how important the SAC model was. This can be seen with the load of articles trying to refine its result, adding a small novelty to the SAC framework, or, as mentioned before, trying to better approach the hyperparameters that would lead to better performances, either with other values adapted to different tasks or with dynamic tuning of the parameters.

As for other prior works and projects, we regretted being constrained both with time and computational power. We have been only able to train enough for the Hopper and Walker2d benchmark tasks, and when realizing our experiments on Hopper, we had to only use half the recommended steps. In addition to that, we would have loved to dive into the training of Humanoid-like challenging tasks, that we could not perform for this very reason.

Finally, we hope we will be able to dive in the other SOTA works around RL (PPO, SAC-X, DRL...) the same way we got interested in this paper.

6 Appendix

6.1 Kullback-Leibler Divergence

Let P and Q be two discrete probability distributions defined on a finite probability space **X**, the Kullback-Leibler divergence between these distributions is calculated from the following formula:

$$KL(P||Q) = \sum_{x \in \mathbf{X}} P(x) \log \frac{P(x)}{Q(x)}$$

6.2 Baseline Hyperparameters

Table 1: SAC Hyperparameters

Parameter	**Value**
Shared	
Optimizer	Adam
Learning rate	$3 \cdot 10^{-4}$
Discount (γ)	0.99
Replay buffer size	10^6
Number of hidden layers (all networks)	2
Number of hidden units per layer	256
Number of samples per minibatch	256
Nonlinearity	ReLU
Target smoothing coefficient (τ)	0.005
Target update interval	1
Gradient steps	1

References

[1] David Silver Hado van Hasselt, Arthur Guez. Deep reinforcement learning with double q-learning. 2015.

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

[3] Tianwei Ni Yufei Wang. Meta-sac: Auto-tune the entropy temperature of soft actor-critic via metagradient. 2019.