A Study of Reinforcement Learning **Report SAC**

Prof. Pietro Michiardi - Eurecom Piero Macaluso
Prof. Elena Baralis - Politecnico di Torino
Version: 1.0 from April, 26 2019

Contents

Revision History

Revision	Date	Author(s)	Description
1.0	April, 26 2019	PM	- Created. - Added SAC description and implementation. - Added Environments Description - Added Uniform/Prioritized Replay implementation and simulation.

1 Introduction

After the analysis of **Deep Deterministic Policy Gradient (DDPG)** [1], I decided to explore other algorithms. **Soft Actor-Critic (SAC)** [2] [3] is one of the algorithms that intrigued me because of the promises of its discoverers: higher and more stable performance than DDPG, stochastic framework and less parameter to tune.

As will be clear later, the algorithm fully met the expectations.

The aim of this report is to show a background of the algorithm, the performances obtained, a comparison with the performance of DDPG and possible future developments.

2 Soft Actor-Critic

2.1 Application Field

SAC combines the off-policy actor-critic setup with a **stochastic policy (actor)**, devising a bridge between stochastic policy optimization and DDPG-style approaches.

As DDPG, SAC can be applied to situations characterized by the presence of a continuous action spaces and it is a **Model-Free**, **Off-Policy** and **Actor-Critic** algorithm.

SAC algorithm is able to overcome some of the problems of DDPG. The latter can achieve great performance, but the interaction between the deterministic actor network and the Q-function makes it difficult to stabilize and brittle with respect to hyper-parameters and other kinds of tuning. The learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.

For this reason, SAC exploits **Clipped Double-Q Learning** used also by **Twin Delayed DDPG (TD3)**. It learns two Q-functions instead of one, and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Another feature of SAC is **entropy regularization**. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This is strongly related to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on, but it can also prevent the policy from prematurely converging to a bad local optimum.

2.2 Key Points

2.2.1 Reinforcement Learning Notation

The Reinforcement Learning Setup is the standard one. The problem can be defined as policy search in a Markov decision process (MDP), defined by a tuple $(\mathcal{S}, \mathcal{A}, p, r)$. The state space $\mathcal{S}$ and action space $\mathcal{A}$ are continuous and the state transition probability $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, \infty)$ represents the probability density of the next state $s_{t+1} \in \mathcal{S}$ given the current state $s_t \in \mathcal{S}$ and action $a_t \in \mathcal{A}$. The environment emits a reward $r : \mathcal{S} \times \mathcal{A} \to [r_{\min}, r_{\max}]$ on each transition. $\rho_\pi(s_t)$ and $\rho_\pi(s_t, a_t)$ denote the state and state-action marginals of the trajectory (τ) distribution induced by a policy $\pi(a_t|s_t)$.

2.2.2 Entropy-Regularized Reinforcement Learning

Entropy is the average rate at which information is produced by a stochastic source of data. It is, in simple terms, a quantity which describes how random a random variable is. The motivation behind the use of entropy is that when the data source produces a low-probability value (rare), the event carries **more *information*** than when the source data produces a high-probability value.

Let x be a random variable with probability mass or density function P. The entropy $\mathcal{H}$ of x is computed from its distribution P according to

$$\mathcal{H}(P) = \mathbb{E}_{x\sim P}[-\log P(x)] \tag{1}$$

.

In **entropy-regularized reinforcement learning** the standard objective is generalized by augmenting it with entropy. The agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. Assuming an infinite-horizon discounted setting, this changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\Big)\right] \tag{2}$$

where $\alpha > 0$ is the temperature parameter that determines the relative importance of the entropy term controlling the stochasticity of the optimal policy. It is clear that the standard maximum expected return can be retrieved in the limit as $\alpha \to 0$.

From eq. (2) we can derive **state-value function** $V^\pi(s)$ and **action-value function** $Q^\pi(s,a)$:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\Big)\Big|s_0 = s\right] \tag{3}$$

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t\mathcal{H}(\pi(\cdot|s_t))\Big|s_0 = s, a_0 = a\right] \tag{4}$$

From these equations is possible to derive the connection between state-value and action-value function given by

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)] + \alpha\mathcal{H}(\pi(\cdot|s)) \tag{5}$$

and the **Bellman equation** given by

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_{s'\sim P, a'\sim\pi}[R(s,a,s') + \gamma(Q^\pi(s',a') + \alpha\mathcal{H}(\pi(\cdot|s')))] &(6)\\ &= \mathbb{E}_{s'\sim P}[R(s,a,s') + \gamma V^\pi(s')] &(7)\end{aligned}$$

2.2.3 Learning Equations

SAC algorithm learns a **policy** π_θ, two Q-functions Q_{ϕ_1}, Q_{ϕ_2}.

The state-value function is emplicitly parametrized through the soft Q-function parameters and the connection is given by:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)] + \alpha\mathcal{H}\left(\pi(\cdot|s)\right) &(8)\\ &= \mathbb{E}_{a\sim\pi}[Q^\pi(s,a) - \alpha\log\pi(a|s)] &(9)\\ &\approx Q^\pi(s,\tilde{a}) - \alpha\log\pi(\tilde{a}|s), \qquad \tilde{a}\sim\pi(\cdot|s). &(10)\end{aligned}$$

In [2] a function approximator for this function was introduced, but later [3] the authors found it to be unnecessary.

Learning Q The Q-functions are learned by Mean Squared Bellman Error (MSBE) minimization, using a target value network to form the Bellman backups. They both use the same target and have loss functions:

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}, d_t) \sim \mathcal{D}} \left[\left(Q_{\phi_i}(s_t, a_t) - \left(r_t + \gamma(1 - d_t) V_{\bar{\phi}_i}(s_{t+1}) \right) \right)^2 \right]. \tag{11}$$

where

$$V_{\bar{\phi}}(s_{t+1}) = \min_{i=1,2} Q_{\bar{\phi}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1}), \qquad a_{t+1} \sim \pi_\theta(\cdot|s_{t+1}) \tag{12}$$

The target value network, like the target networks in DDPG, is obtained by polyak averaging the value network parameters over the course of training.

Learning the Policy The policy should act to maximize the expected future return plus expected future entropy, for each state. That is, it should maximize $V^\pi(s)$, which can be expanded out (as before) into

$$E_{a \sim \pi}[Q^\pi(s, a) - \alpha \log \pi(a|s)] \tag{13}$$

The optimization of the policy makes use of the reparameterization trick, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

$$\tilde{a}_\theta(s, \xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I). \tag{14}$$

It is possible to rewrite the expectation over actions (pain point: the distribution depends on the policy parameters) into an expectation over noise (the distribution now has no dependence on parameters) thanks to the reparameterization trick:

$$\mathbb{E}_{a \sim \pi_\theta}[Q^{\pi_\theta}(s, a) - \alpha \log \pi_\theta(a|s)] = \mathbb{E}_{\xi \sim \mathcal{N}}[Q^{\pi_\theta}(s, \tilde{a}_\theta(s, \xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi)|s)] \tag{15}$$

To get the policy loss, the final step is to substitute Q^{π_θ} with one of our function approximators. As suggested from the authors of [3], $\min_{i=1,2} Q_{\phi_i}(s_t, \tilde{a}_\theta(s, \xi))$ is used. The policy is thus optimized according to

$$\max_\theta \mathbb{E}_{s \sim \mathcal{D}, \xi \sim \mathcal{N}}[\alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi)|s) - \min_{i=1,2} Q_{\phi_i}(s_t, \tilde{a}_\theta(s, \xi))], \tag{16}$$

which is almost the same as the DDPG and TD3 policy optimization, except for the stochasticity and entropy term.

The implementation of these updates can be found in algorithm 1 on the next page.

2.2.4 Replay Buffers

Most optimization algorithms assume that the samples are **independently and identically distributed (i.i.d)**, but data produced sequentially exploring the environment can not satisfy this assumption. To solve this problem, a Replay Buffer can be used: it is a set $\mathcal{D}$ of N recent experiences $(s_t, a_t, r_t, s_{t+1}, d_t)$ from which the algorithm will randomly sample a subset of $M \ll N$ experiences (mini-batch) at each iteration.

The replay buffer should be large enough to contain a wide range of experiences in order to have stable algorithm behavior, but it may not always be good to keep everything. Using only the very-most recent data leads to overfitting, while using too much experience may slow down the learning process.

Algorithm 1: Update Phase SAC

```
        with torch.no_grad():
            next_state_action, next_state_log_pi, _ =
                self.policy.sample(next_state_batch)
            qf1_next_target, qf2_next_target = self.critic_target(next_state_batch,
                next_state_action)
            min_qf_next_target = torch.min(qf1_next_target, qf2_next_target) -
                self.alpha * next_state_log_pi
            next_q_value = reward_batch + mask_batch * self.gamma * min_qf_next_target

        qf1, qf2 = self.critic(state_batch, action_batch)
        qf1_loss = F.mse_loss(qf1, next_q_value)
        qf2_loss = F.mse_loss(qf2, next_q_value)

        pi, log_pi, _ = self.policy.sample(state_batch)

        qf1_pi, qf2_pi = self.critic(state_batch, pi)
        min_qf_pi = torch.min(qf1_pi, qf2_pi)


        policy_loss = ((self.alpha * log_pi) - min_qf_pi).mean()

        self.critic_optim.zero_grad()
        qf1_loss.backward()
        self.critic_optim.step()

        self.critic_optim.zero_grad()
        qf2_loss.backward()
        self.critic_optim.step()

        self.policy_optim.zero_grad()
        policy_loss.backward()
        self.policy_optim.step()

        if self.autotune_entropy:
            alpha_loss = -(self.log_alpha * (log_pi +
                self.target_entropy).detach()).mean()

            self.alpha_optim.zero_grad()
            alpha_loss.backward()
            self.alpha_optim.step()

            self.alpha = self.log_alpha.exp()
            alpha_tlogs = self.alpha.clone() # For TensorboardX logs
        else:
            alpha_loss = torch.tensor(0.).to(self.device)
            alpha_tlogs = torch.tensor(self.alpha) # For TensorboardX logs

        if updates % self.target_update == 0:
            soft_update(self.critic_target, self.critic, self.tau)
```

2.2.5 Target Networks

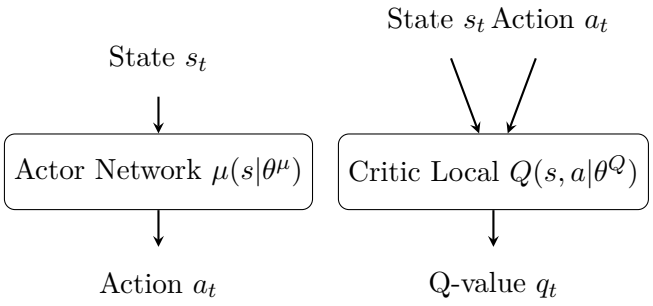

Figure 1: Actor and Critic Networks

In SAC we have 5 neural networks: the **local stochastic Policy Network**, the **2 local Q-Networks**, the **2 target Q-Network**.

Initially Policy (Actor) and Q-Networks (Critics) have randomly initialized weights. Then the local Actor (the current policy) starts to propose actions to the Agent, given the current state, starting to populate the Replay Buffer of experiences.

When the Replay Buffer is big enough, the algorithm starts to sample randomly a mini-batch of experiences for each timestep t. This mini-batch is used to update the local Critics (eq. (11) on page 5) and to update the actor policy (eq. (16) on page 5).

We can imagine the target networks as the *labels* of supervised learning.

Also the target networks are updated in this *learning step*. A mere copy of the local weights is not an efficient solution, because it is prone to divergence. For this reason, a "soft" target updates is used. It is given by

$$\theta' \leftarrow \tau\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

with $t \ll 1$.

The pseudo-code of this procedure is shown in algorithm 1 on page 9

2.2.6 Exploration vs. Exploitation

SAC algorithm trains a stochastic policy using **entropy regularization**, and explores in an **on-policy** way. α is the entropy regularization coefficient which is the one that explicitly controls the exploration-exploitation tradeoff. Higher α corresponds to more exploration, while lower α corresponds to more exploitation.

This is one of the most important parameter in the algorithm and it may vary from environment to environment. For this reason, it could require a careful tuning in order to find the one which leads to the stablest and highest-reward learning.

During the tests, the stochasticity is removed by using the mean action instead of a sample from the distribution. This tends to improve performance over the original stochastic policy, allowing us to see how well the policy exploits what it has learned.

2.3 Steps made

The initial step was trying to implement the algorithm in [3] using OpenAI Gym environment *Pendulum-v0*.

First, we used Neural Networks to compare the results with DDPG of the previous report and finally we decided to apply a Convolutional Neural Network (CNN) to states represented by a set of RGB images of the same environment.

2.3.1 Hyper Parameters

The aim of this section is to describe the Hyper Parameters of DDPG.

Alpha (`alpha`) it is the entropy regulation parameter.

Replay (`batch_size`, `replay_min_size`, `replay_max_size`) `batch_size` is the dimension of the mini-batch sampled by the memory. The learning process starts when the replay memory contains at least `replay_min_size` transitions and it starts to overwrite old transitions when it reaches `replay_max_size`.

Episode (`n_episode`, `episode_max_len`) the number of episode for each run is `n_episode`, while the maximum length of an episode is `episode_max_len`.

Convolutional Neural Networks (`weight_decay`, `update_method`, `lr`) set of parameter for each network. The first parameter is always set to 0 and never used. The second one is always set to Adaptive Moment Estimation (ADAM), while the third is the learning rate and it is usually set to `1e-3` or `1e-4`.

Update (`discount`, `soft_target_tau`, `n_updates_per_step`) `discount` is γ, `soft_target_tau` is τ, while `n_updates_per_step` is the number of times that the algorithm has to extract a mini-batch and perform the update of the networks for each timestep.

Test (`n_tests`, `every_n_episode`) `n_tests` is the number of episode to test in the testing phase, while `every_n_episode` indicates how often the testing phase starts.

Algorithm 1: Soft Actor-Critic

Input: Initialize policy parameter θ, Q-function parameters ϕ_1, ϕ_2

Initialize target network weights $\bar{\phi}_1 \leftarrow \phi_1, \bar{\phi}_2 \leftarrow \phi_2$

Initialize an empty replay buffer $\mathcal{D}$

repeat

  Observe state s_t and select action $a_t \sim \pi_\theta(\cdot|s_t)$

  Execute a_t in the environment

  Observe next state s_{t+1}, reward r_t and done signal d_t

  Store $(s_t, a_t, r_t, s_{t+1}, d_t)$ in replay buffer $\mathcal{D}$

  If s_{t+1} is terminal, reset the environment state.

  if *size of* $\mathcal{D}$ > *warm_up_threshold* **then**

    for *j in range(#updates_per_step)* **do**

      Randomly sample a batch of transitions, $B = (s_t, a_t, r_t, s_{t+1}, d_t)$ from $\mathcal{D}$

      Compute targets for Q functions:

$$y_q(r_t, s_{t+1}, d_t) = r_t + \gamma(1 - d_t)V_{\bar{\phi}}(s_{t+1})$$

      where

$$V_{\bar{\phi}}(s_{t+1}) = \min_{i=1,2} Q_{\bar{\phi}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1}), \qquad a_{t+1} \sim \pi_\theta(\cdot|s_{t+1})$$

      Update Q-functions by one step of gradient descent using

$$J_Q(\phi_i) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}, d_t) \sim B}\left[\frac{1}{2}(Q_{\phi_i}(s_t, a_t) - y_q(r_t, s_{t+1}, d_t))^2\right] \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$J_\pi(\theta) = \mathbb{E}_{s_t \sim B, \xi \sim \mathcal{N}(0,I)}\Big[\alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi)|s_t) - \min_{i=1,2} Q_{\phi_i}(s_t, \tilde{a}_\theta(s, \xi))\Big]$$

      where $a_t = \tilde{a}_\theta(s, \xi)$ which is differentiable wrt θ via the reparametrization trick.

      Update target Q-networks with:

$$\bar{\phi}_i \leftarrow \tau\phi_i + (1 - \tau)\bar{\phi}_i, \qquad \text{for } i = 1, 2$$

    end

  end

until *convergence*

Output: Optimized policy parameter θ and Q-function parameters ϕ_1, ϕ_2

3 OpenAI Gym Environments

3.1 Pendulum-v0

3.1.1 Description

The inverted pendulum swingup problem is a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright.

Observation Type: Box(3)

Index	Observation	Min	Max
0	$\cos(\theta)$	-1.0	$+1.0$
1	$\sin(\theta)$	-1.0	$+1.0$
2	$\dot{\theta}$	-8.0	$+8.0$

Actions Type: Box(1)

Index	Action	Min	Max
0	Joint effort	-2.0	$+2.0$

Reward The reward for each timestep t is given by

$$r_t = -(\theta_t^2 + 0.1\dot{\theta}^2 + 0.001a_t^2)$$

where theta is normalized between $-\pi$ and π. Therefore, the lowest cost is $-(\pi^2 + 0.1 * 8^2 + 0.001 * 2^2) = -16.2736044$, and the highest cost is 0. In essence, the goal is to remain at zero angle (vertical), with the least rotational velocity, and the least effort.

Starting State Random angle from $-\pi$ to π, and random velocity between -1 and 1

Episode Termination There is no specified termination. Adding a maximum number of steps might be a good idea. In this case 200.

Solved Requirements It is an unsolved environment, which means it does not have a specified reward threshold at which it is considered solved.

3.1.2 Hyper-Parameters Used

Type	Parameter	Value	Parameter	Value	Parameter	Value
Epsilon	`eps_start`	0.9	`eps_end`	0.2	`eps_decay`	300
Noise	`mu`	0.0	`sigma`	0.3	`theta`	0.15
Replay	`batch_size`	30	`replay_min_size`	2500	`replay_max_size`	10^6
Episode	`n_episode`	300	`episode_max_len`	200		
Networks	`weight_decay`	0.0	`update_method`	'adam'	`lr`	$1e^{-4}$
Update	`discount`	0.99	`soft_target_tau`	0.001	`n_updates_per_step`	1
Test	`n_tests`	100	`every_n_episode`	10		

4 Comparing Results

In order to better evaluate the performances of these algorithms, **TensorboardX** was used. The mean μ, min, max and standard deviation σ were calculated with a tool and the important areas they describe were plotted for better visualization.

Training Phase The training phase was repeated 20 times for `n_episode` episodes and the results were used to calculate aggregate values.

Test Phase After `every_n_episode` episodes, the test phase was triggered. In this part the current actor network was set in evaluation mode and tested on 100 random episodes. Also these results were used to calculate aggregate values.

In the first 10 episodes, the selection of the actions to take are sampled from a uniform random distribution over valid actions. This is a way to improve exploration in the first steps. After that, it returns to normal DDPG/SAC exploration.

4.1 Durations

Environment	Uniform Replay Memory		Prioritized Replay Memory	
	One	Total	One	Total
MountainCarContinuous-v0	15 min	5 h	13 min	4.5 h
Pendulum-v0	6 min	2 h	6 min	2 h

4.2 Pendulum-v0

4.2.1 DDPG with Uniform Replay Memory

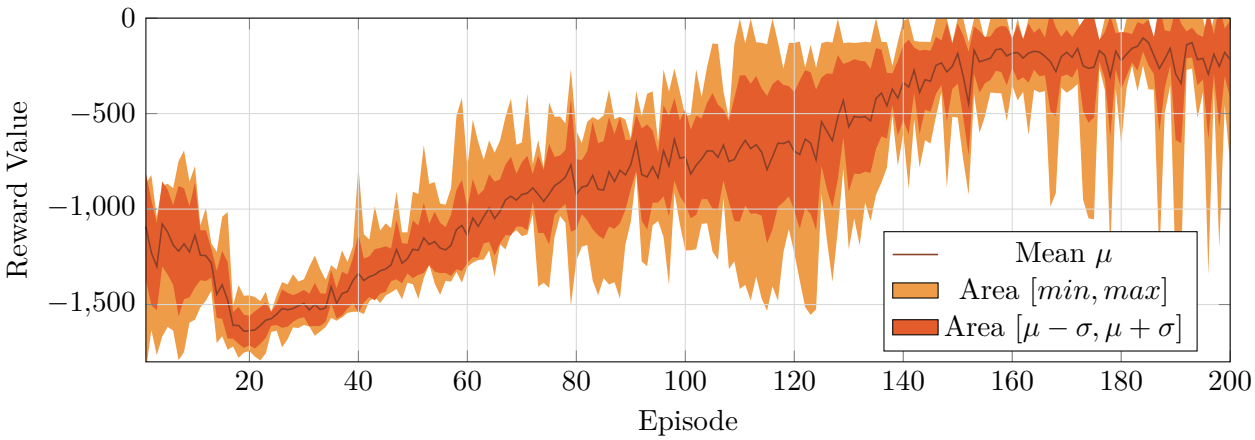

Figure 2: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

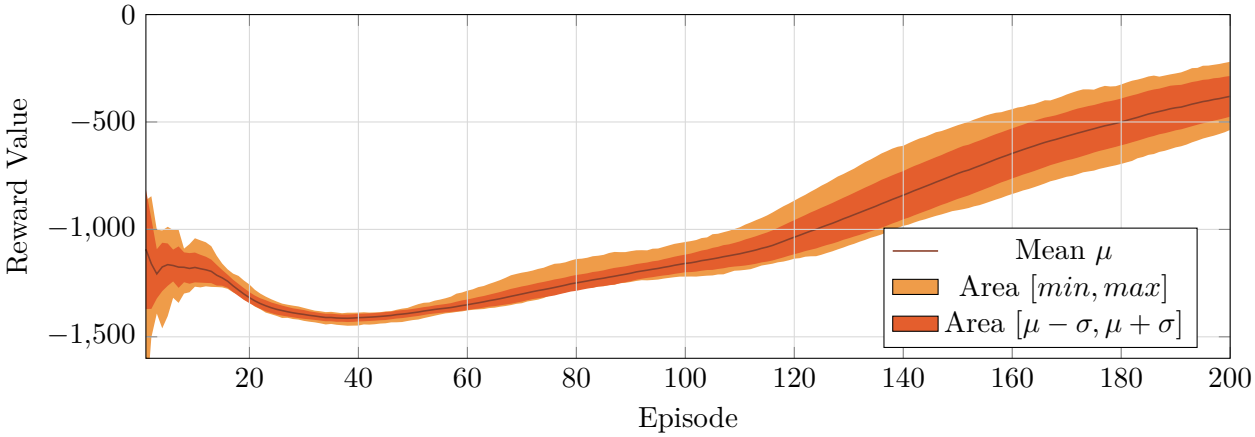

Figure 3: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

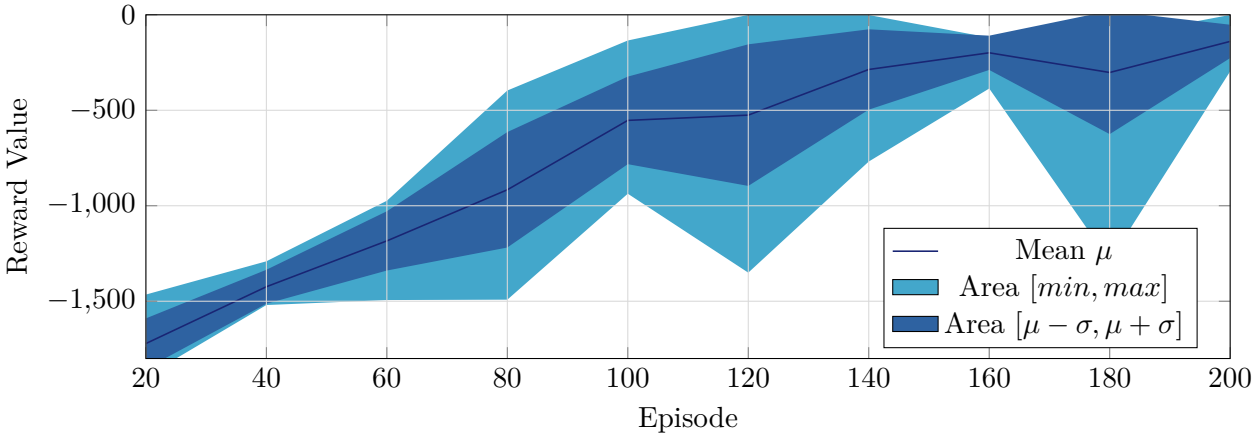

Figure 4: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 10 runs.

4.2.2 SAC

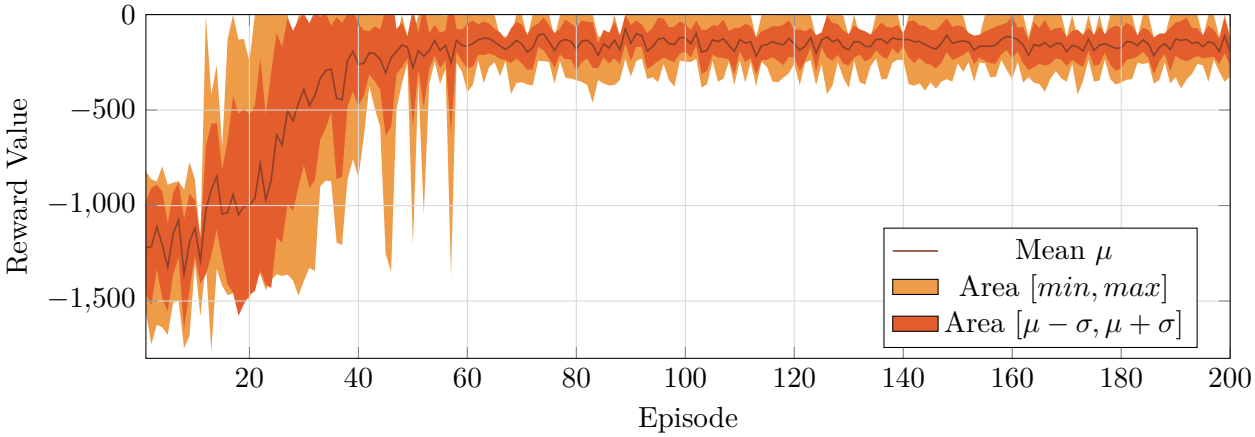

Figure 5: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

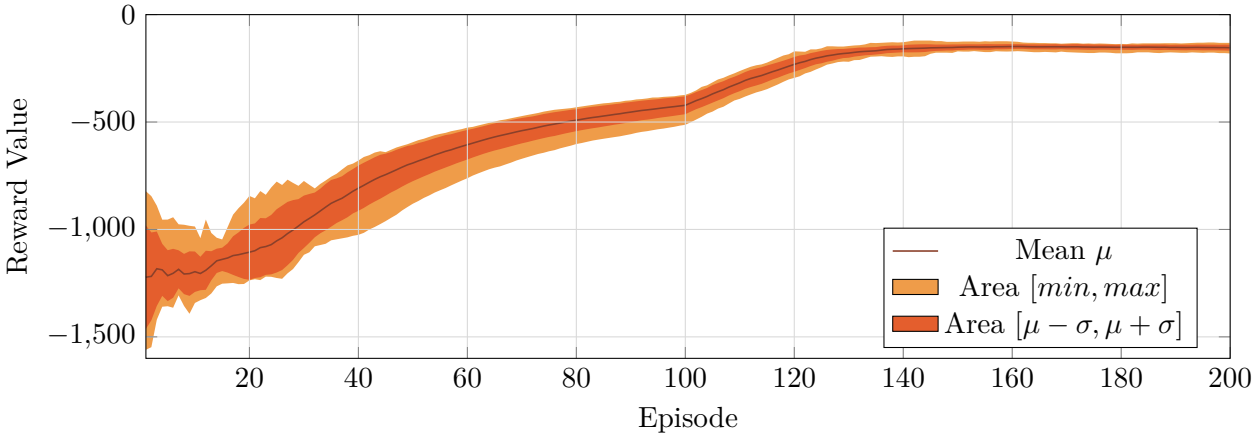

Figure 6: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

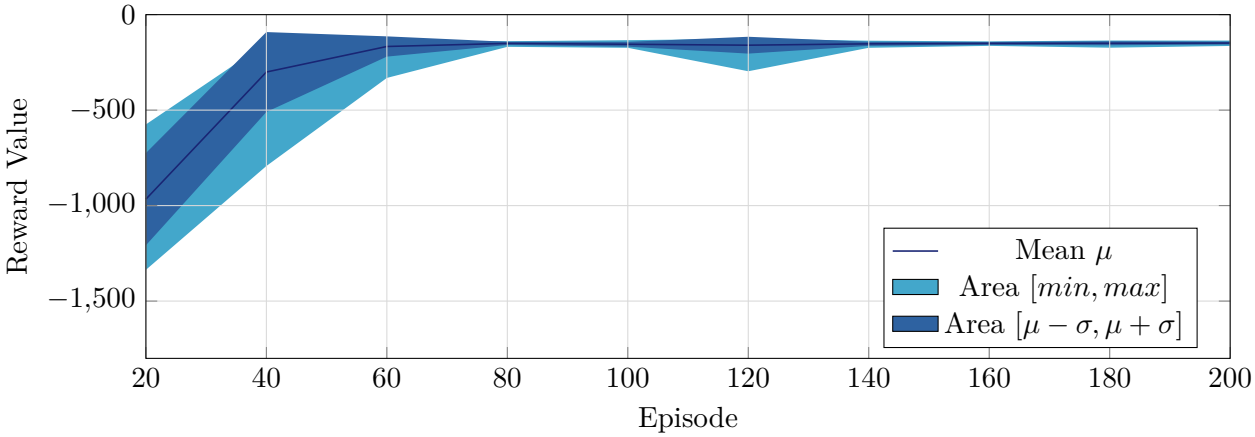

Figure 7: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 20 episodes) over 10 runs.

4.2.3 SAC with Autotune

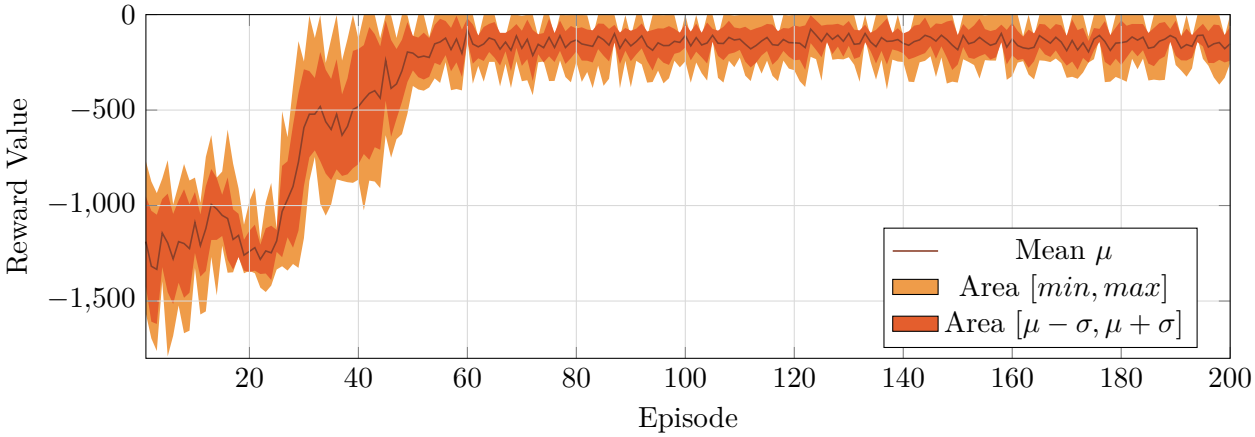

Figure 8: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

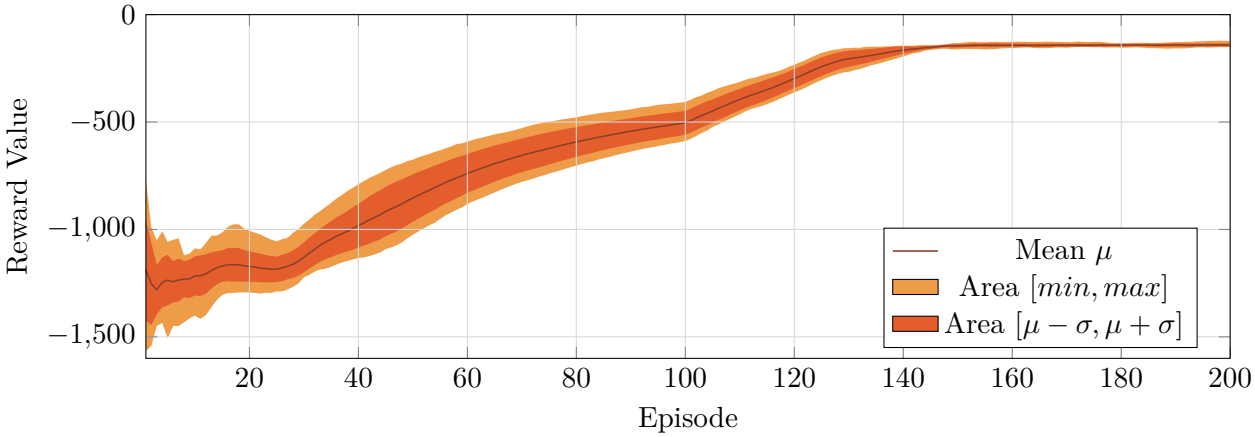

Figure 9: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

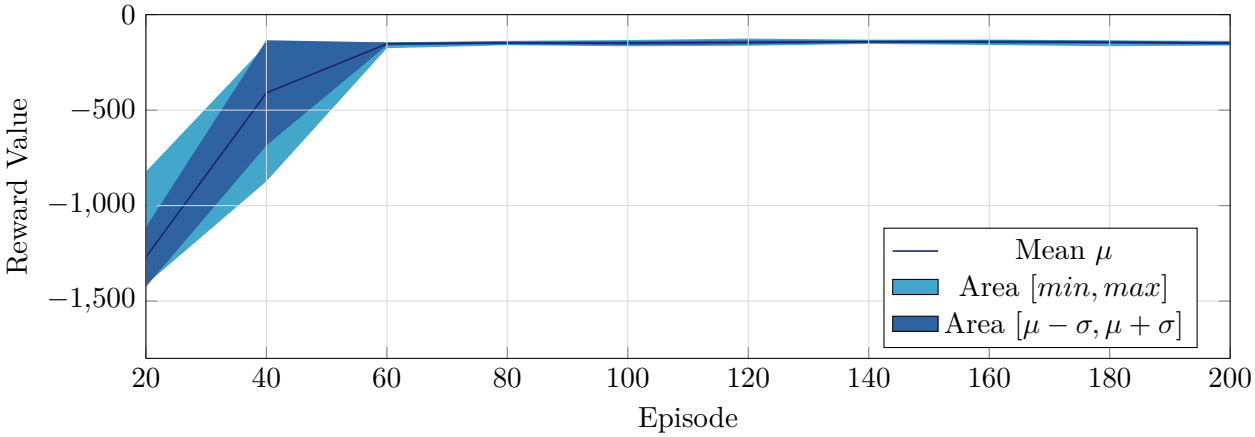

Figure 10: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 20 episodes) over 10 runs.

5 Comments

Analyzing the graphs, it is clear that the results in moving from DDPG to SAC are really impressive as anticipated in the paper [3].

The results obtained are better than the ones of DDPG, on the other hand the execution time increases: for this reason the number of element extracted from the mini-batch at each episode was decreased.

The aim of SAC auto-tuning is to learn the best value of α without hard-coded values. It reaches good results, but slightly worse than the plain SAC, but I think that this is an acceptable compromise

I think that the problem may be caused by some bugs in the implementation of this part.

6 Next Steps

The next steps that I am planning to analyze are:

Prioritized Problems I think that the problem behind not impressive results of Prioritized Buffer are related to some bug in the implementation of this part. I will try to find a way to fix this.

Convolutional Neural Network I had already started to test this part, but I have no relevant results, yet. I applied the code to *CarRacing-v0* which propose a continuous environment with a RGB vector as observation. I used a `StateBuffer` of size three to use the last three states as input for the Convolutional Neural Network (9 inputs $= 3 \cdot 3$ RGB channels).

I found difficulties to test whether an algorithm or a particular CNN architecture is working because of the length of the training in this particular environment. I will try to find a better way to test: an idea could be to apply this approach to *MountainCarConinuous-v0* or *Pendulum-v0*, trying to get the image as observation instead of raw data.

This is the **most important part** to develop because it is the base of the work with Anki Cozmo. I will focus mainly on that in the next days.

DDPG vs others I am thinking to invest a part of the work in searching alternative algorithms recently discovered and reading new papers about them. For instance, some of the latest algorithm used in Autonomous Driving Reinforcement Learning are **Deep Distributed Distributional Deterministic Policy Gradients (D4PG)**, **Twin Delayed DDPG (TD3)** and **Soft Actor Critic (SAC)**. The last one is a sort of bridge between stochastic policy optimization and DDPG-style approaches. They seems promising in the context of the project and maybe they could lead to better results. I will try to explore more deeply this part.

References

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[2] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[3] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[4] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.