A Study of Reinforcement Learning **Report SAC**

Prof. Pietro Michiardi - Eurecom
Prof. Elena Baralis - Politecnico di Torino

Piero Macaluso

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** $Q^\pi(s,a)$:

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

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

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} and a value function V_ψ.

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:

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}}\left[\bigg(Q_{\phi_i}(s,a) - \Big(r + \gamma(1-d) V_{\psi_{\text{targ}}}(s')\Big)\bigg)^2\right]. \tag{8}$$

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 V The value function is learned by exploiting a sample-based approximation of the connection given by eq. (5) on the previous page. Let's first rewrite the connection equation by using the definition of entropy to obtain:

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

The RHS is an expectation over actions, so we can approximate it by sampling from the policy:

$$V^\pi(s) \approx Q^\pi(s,\tilde{a}) - \alpha\log\pi(\tilde{a}|s), \qquad \tilde{a}\sim\pi(\cdot|s). \tag{11}$$

SAC sets up a mean-squared-error loss for V_ψ based on this approximation. SAC uses **Clipped double-Q Learning** like TD3 for learning the value function, and takes the minimum Q-value between the two approximators. So the SAC loss for value function parameters is:

$$L(\psi,\mathcal{D}) = E_{s\sim\mathcal{D},\tilde{a}\sim\pi_\theta}\left(V_\psi(s) - \left(\min_{i=1,2} Q_{\phi_i}(s,\tilde{a}) - \alpha\log\pi_\theta(\tilde{a}|s)\right)\right)^2. \tag{12}$$

Importantly, we do not use actions from the replay buffer here: these actions are sampled fresh from the current version of the policy.

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

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

The way we optimize 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

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

The implementation of these updates can be found in algorithm 1.

Algorithm 1: Updating Critic, Actor and Target Networks

```
        # UPDATE CRITIC #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.target_policy_net(next_states)
        q_targets_next = self.target_value_net(next_states, actions_next.detach())
        # Compute Q targets for current states (y_i)
        q_targets = rewards + (gamma * q_targets_next * (1.0 - done))
        # Compute critic loss
        q_expected = self.critic_net(states, actions)
        critic_loss = self.critic_loss(q_expected, q_targets)
        # Minimize the loss
        self.critic_opt.zero_grad()
        critic_loss.backward()
        self.critic_opt.step()

        # UPDATE ACTOR #
        # Compute actor loss
        actions_pred = self.actor_net(states)
        actor_loss = -self.critic_net(states, actions_pred).mean()
        # Maximize the expected return
        self.actor_opt.zero_grad()
        actor_loss.backward()
        self.actor_opt.step()

        # UPDATE TARGET NETWORK #
        self.soft_update(self.critic_net, self.target_value_net, self.soft_target_tau)
        self.soft_update(self.actor_net, self.target_policy_net, self.soft_target_tau)
```

2.2.4 Replay Buffers

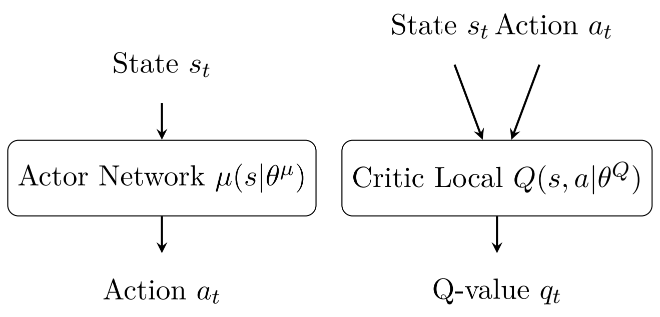
Most optimization algorithms assume that the samples are **independently and identically distributed (i.i.d)**, but data produced sequentially exploring the environment can not satisfy this assumption. To solve this problem, a Replay Buffer can be used: it is a set $\mathcal{D}$ of N recent experiences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

The first approach is to select the mini-batch sampling uniformly among all the entries in the Replay Buffer. A more complex approach is the one using a **Prioritized Buffer Replay** [4] where the samples with high expected learning progress are replayed more frequently. This prioritization can lead to a loss of diversity, which can be alleviated by **stochastic prioritization**, and a bias that can be corrected by **importance sampling**.

In section 4 on page 12 the results of the two approaches will be analyzed.

2.2.5 Target Networks



State s_t
State s_t Action a_t
Actor Network $\mu(s|\theta^\mu)$
Critic Local $Q(s,a|\theta^Q)$
Action a_t
Q-value q_t

Figure 1: Actor and Critic Networks

In DDPG we have 4 neural networks: the **local Actor**, the **local Critic**, the **target Actor** and the **target Critic**. The aim of Actor networks is to approximate the **Policy** while the Critic networks approximate the **Q-Value**.

Initially Actors and Critics have the same 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 Critic minimizing the Mean Squared Loss between the local Q-value and the target one (eq. (15)) and to update the actor policy using the sampled policy gradient (**??** on page ??).

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{15}$$

where y_i is given by **??** on page ??.

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 **??** on page ??

2.2.6 Exploration vs. Exploitation

In Reinforcement learning for discrete action spaces, exploration is done selecting a random action (e.g. epsilon-greedy). For continuous action spaces, exploration is done adding noise to the action itself. In [1], the authors use Ornstein-Uhlenbeck Process [5] to add noise to the action output $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}$. After that the action is clipped in the correct range.

2.3 Steps made

The initial step was trying to implement the algorithm in [1] using two simple environment provided by OpenAI Gym: *MountainCarContinuous-v0* and *Pendulum-v0.* In this report we will use only shallow Neural Networks implemented as shown in fig. 2 on the following page, because the state is represented directly by the data of the observation.

The next goal will be to apply a Convolutional Neural Network (CNN) to states represented by a set of RGB images of the same environments.

In this report the algorithm was implemented using a simple Replay Buffer and a Prioritized one in order to understand better the efficiency and the differences. In section 4 on page 12 is possible to observe the results and the comparison of these two approach.

2.3.1 Hyper Parameters

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

Epsilon (`eps_start`, `eps_end`, `eps_decay`) it is described by the function

$$\epsilon = \epsilon_{\text{start}} - (\epsilon_{\text{start}} - \epsilon_{\text{end}}) \min(1.0, \frac{e}{\epsilon_{\text{decay}}})$$

where e is the current episode number. It is used to decrease the impact of the noise on the actions in function of the number of episode. When it reaches the `eps_end`, it will become a constant.

Noise (`mu`, `sigma`, `theta`) these are the Ornstein-Uhlenbeck Process Noise parameters.

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`.

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.

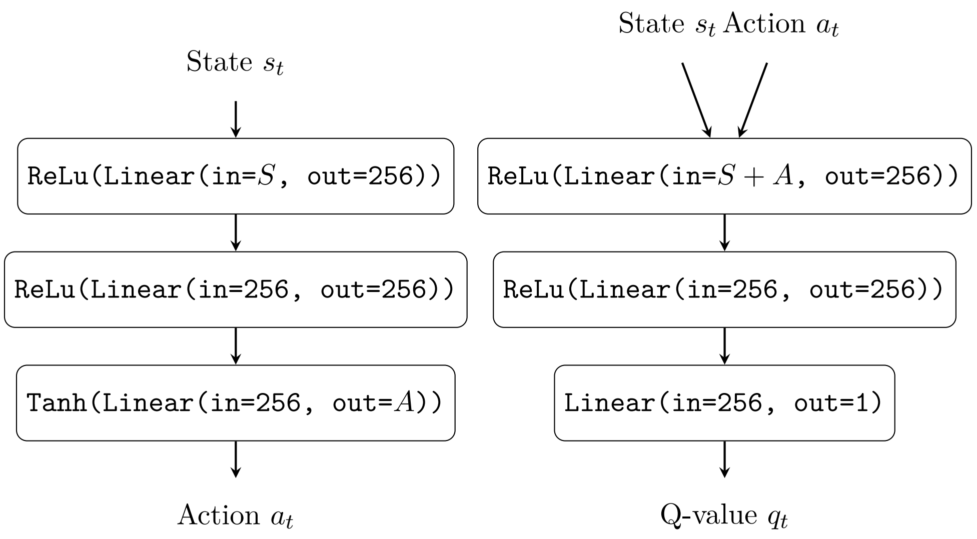

State s_t
ReLu(Linear(in=S, out=256))
ReLu(Linear(in=256, out=256))
Tanh(Linear(in=256, out=A))
Action a_t
State s_t Action a_t
ReLu(Linear(in=$S+A$, out=256))
ReLu(Linear(in=256, out=256))
Linear(in=256, out=1)
Q-value q_t

Figure 2: Actor and Critic Networks: S is the length of the array of states, while A is the length of the array of actions.

```
Algorithm 1: Soft Actor-Critic
   Input: Initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, V-function
          parameters $\psi$, empty replay buffer $\mathcal{D}$
 1 Set target parameters equal to main parameters $\psi_{targ} \leftarrow \psi$
 2 repeat
 3     Observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
 4     Execute $a$ in the environment
 5     Observe next state $s'$, reward $r$ and done signal $d$
 6     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
 7     If $s'$ is terminal, reset the environment state.
 8     if size of $\mathcal{D}$ > warm_up_threshold then
 9         for j in range(#updates_per_step) do
10             Ramdomly sample a batch of transitions, $B = (s, a, r, s', d)$ from $\mathcal{D}$
11             Compute targets for Q and V functions:
                   $y_q(r, s', d) = r + \gamma(1-d)V_{\psi_{targ}}(s')$   (16)
                   $y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \quad \tilde{a} \sim \pi_\theta(\cdot|s)$   (17)
12             Update Q-functions by one step of gradient descent using
                   $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_{\phi,i}(s,a) - y_q(r,s',d))^2 \qquad \text{for } i = 1, 2$
13             Update V-function by one step of gradient descent using
                   $\nabla_\psi \frac{1}{|B|} \sum_{s \in B} (V_\psi(s) - y_v(s))^2$
14             Update policy by one step of gradient ascent using
                   $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left( Q_{\phi,1}(s, \tilde{a_\theta(s)}) - \alpha \log \pi_\theta(\tilde{a_\theta(s)}|s) \right)$
                , where $a_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via the
                reparametrization trick.
15             Update target value network with:
                   $\psi_{targ} \leftarrow \tau\psi_{targ} + (1-\tau)\psi$
16         end
17     end
18 until convergence
```

3 OpenAI Gym Environments

3.1 MountainCarContinuous-v0

3.1.1 Description

An underpowered car must climb a one-dimensional hill to reach a target. The action (engine force applied) is a continuous value. The target is on top of a hill on the right-hand side of the car. If the car reaches it or goes beyond, the episode terminates. On the left-hand side, there is another hill. Climbing this hill can be used to gain potential energy and accelerate towards the target. On top of this second hill, the car cannot go further than a position equal to -1, as if there was a wall. Hitting this limit does not generate a penalty.

Observation Type: Box(2)

Index	Observation	Min	Max
0	Car Position	-1.2	$+0.6$
1	Car Velocity	-0.07	$+0.07$

Actions Type: Box(1)

Index	Action	Min	Max
0	Push car to the left (negative value) or to the right (positive value)	-1	$+1$

Reward The reward for each timestep t is given by $r_t = d_t * 100 - a_t^2 * 0.1$ where d_t is the done flag and a_t is the continuous value of the action taken.

This reward function raises an exploration challenge, because if the agent does not reach the target soon enough, it will figure out that it is better not to move, and won't find the target anymore.

Starting State Position between -0.6 and -0.4, null velocity.

Episode Termination Position equal to 0.5.

Solved Requirements Get a reward over 90.

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`	100|32	`replay_min_size`	10^4	`replay_max_size`	10^6
Episode	`n_episode`	300	`episode_max_len`	1000		
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		

3.2 Pendulum-v0

3.2.1 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.

3.2.2 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 MountainCarContinuous-v0

4.2.1 Uniform Replay Memory

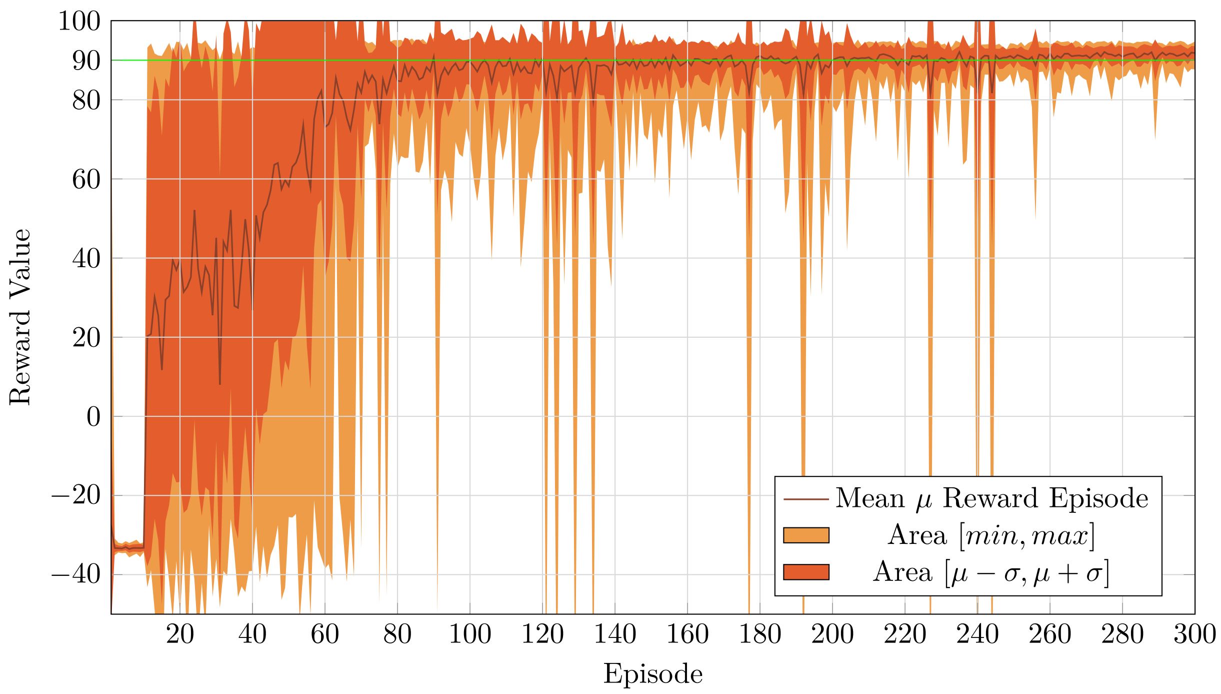


Figure 3: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

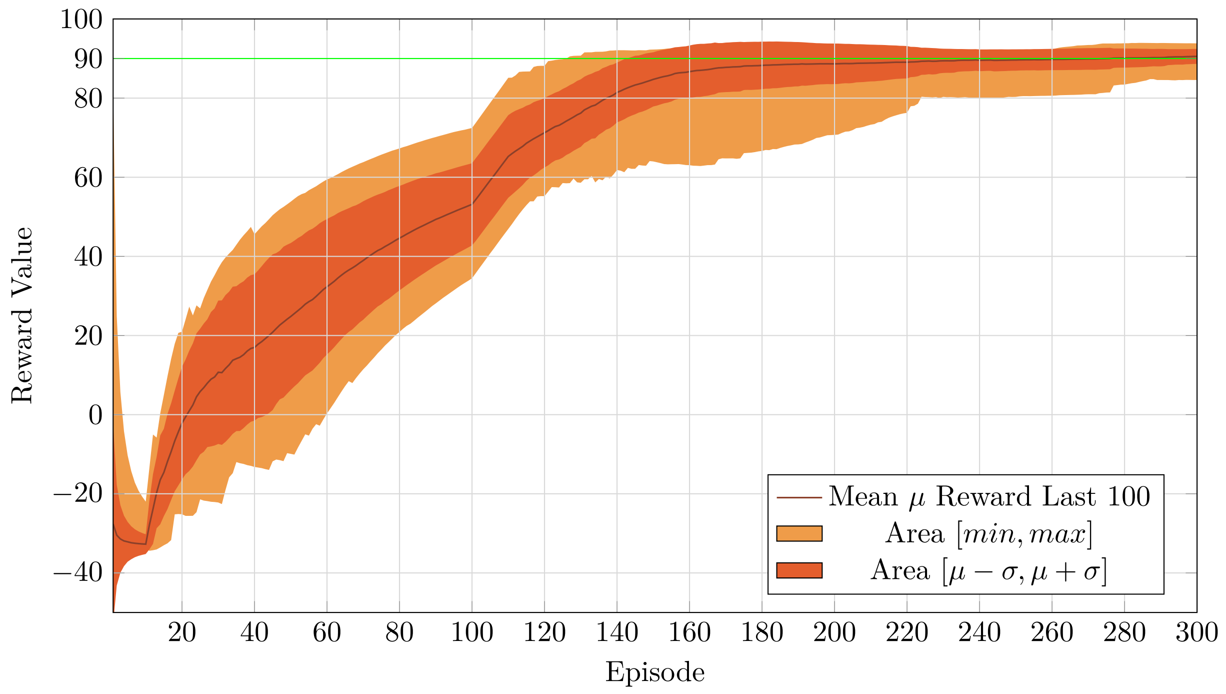


Figure 4: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

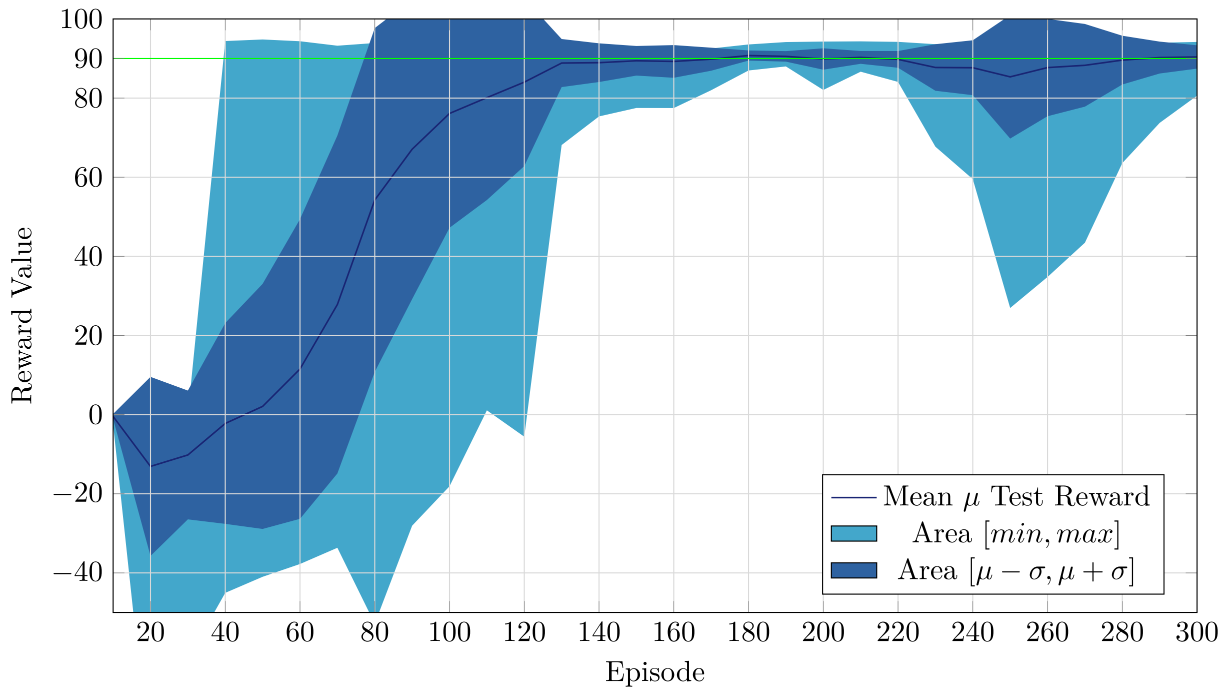


Figure 5: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

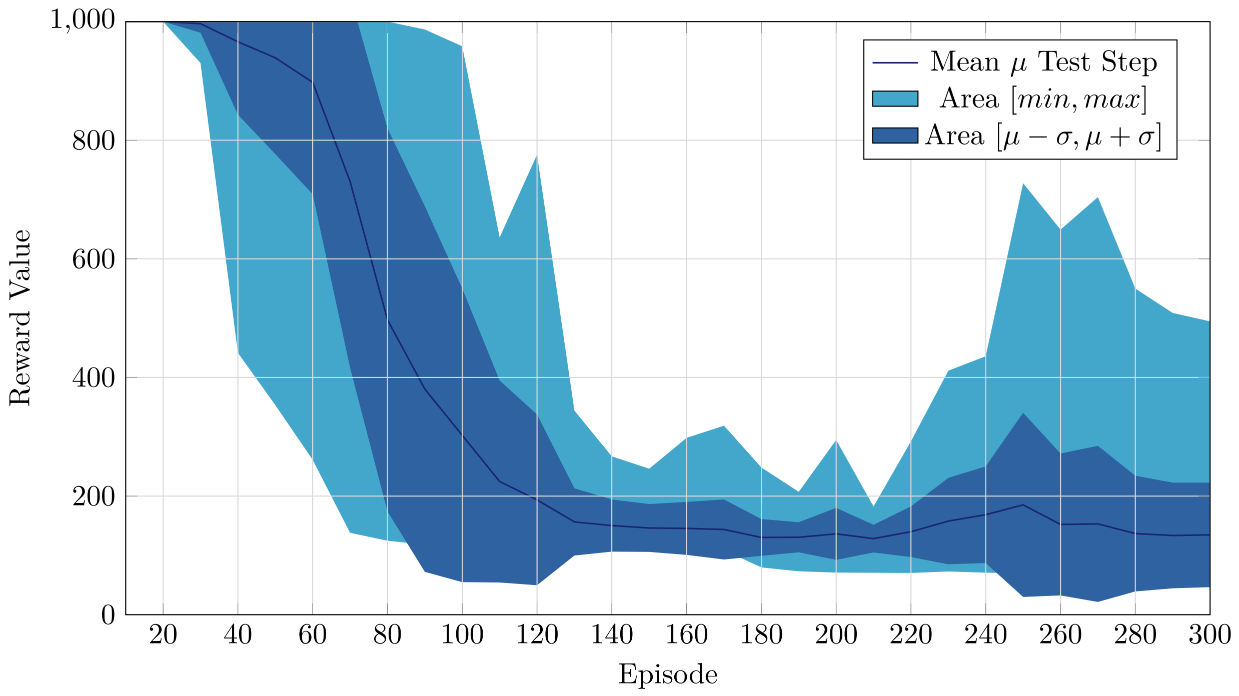


Figure 6: Mean, Standard Deviation Range and Min-Max range of the number of steps in the test phase (every 10 episodes) over 20 runs.

4.2.2 Prioritized Replay Memory

In this case the size of the mini-batch was reduced from 100 to 32 in order to make the learning faster.

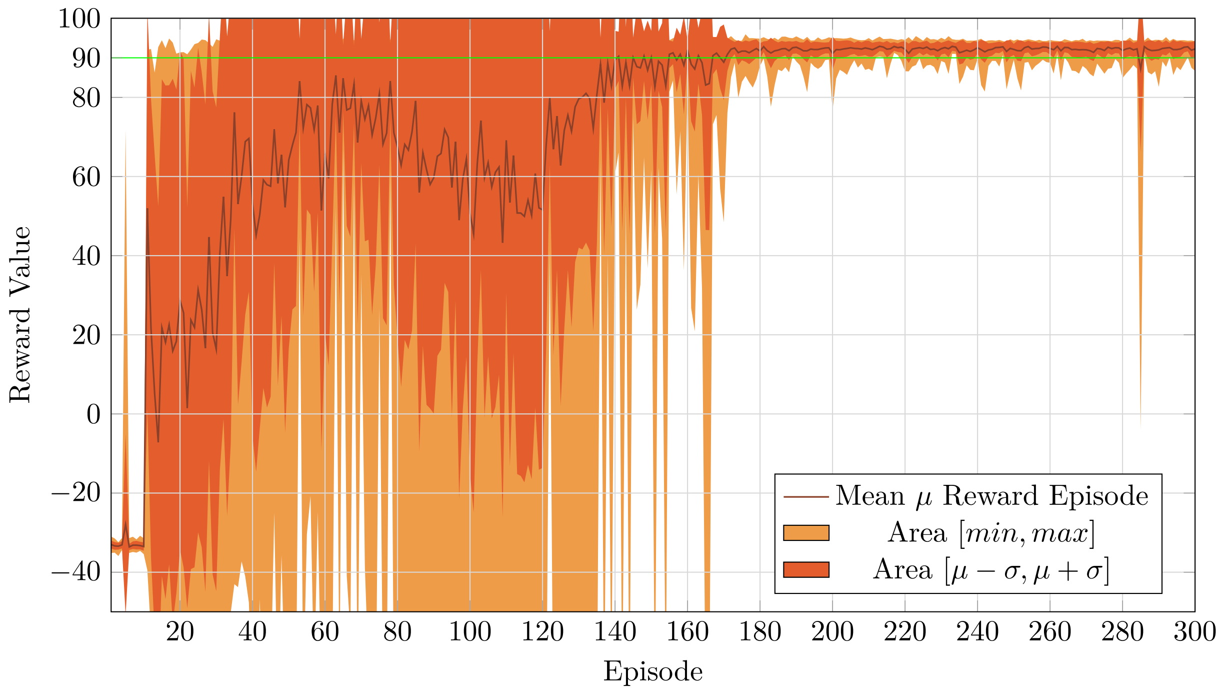


Figure 7: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

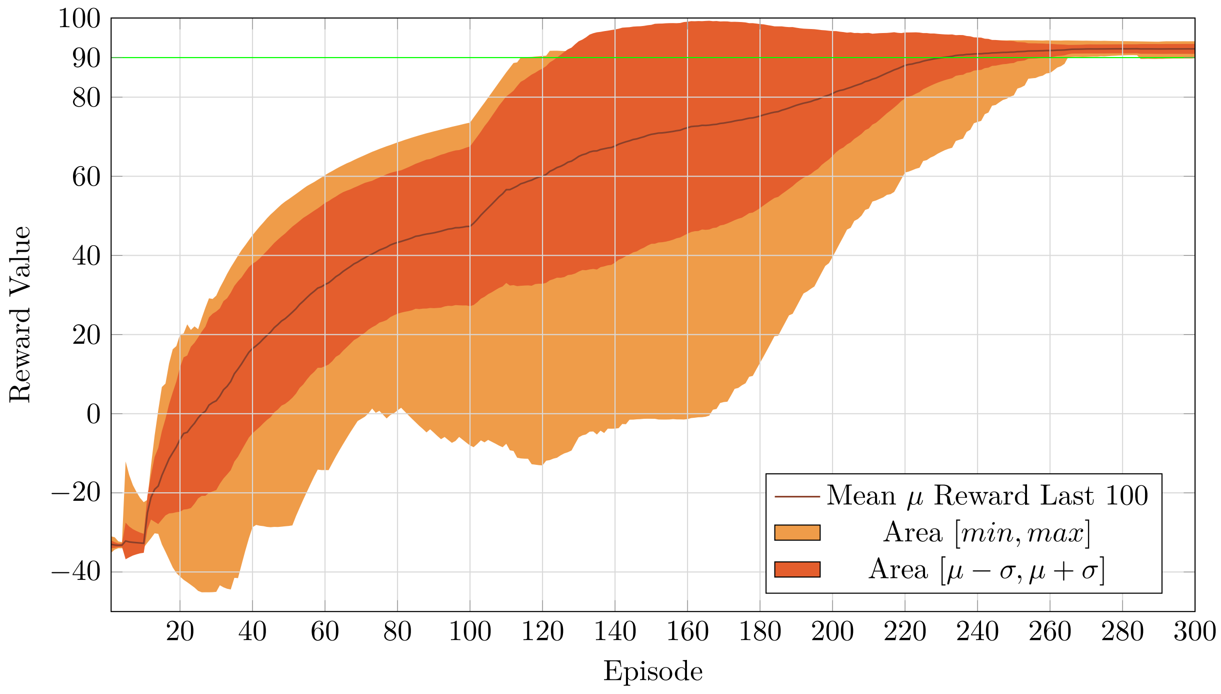


Figure 8: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

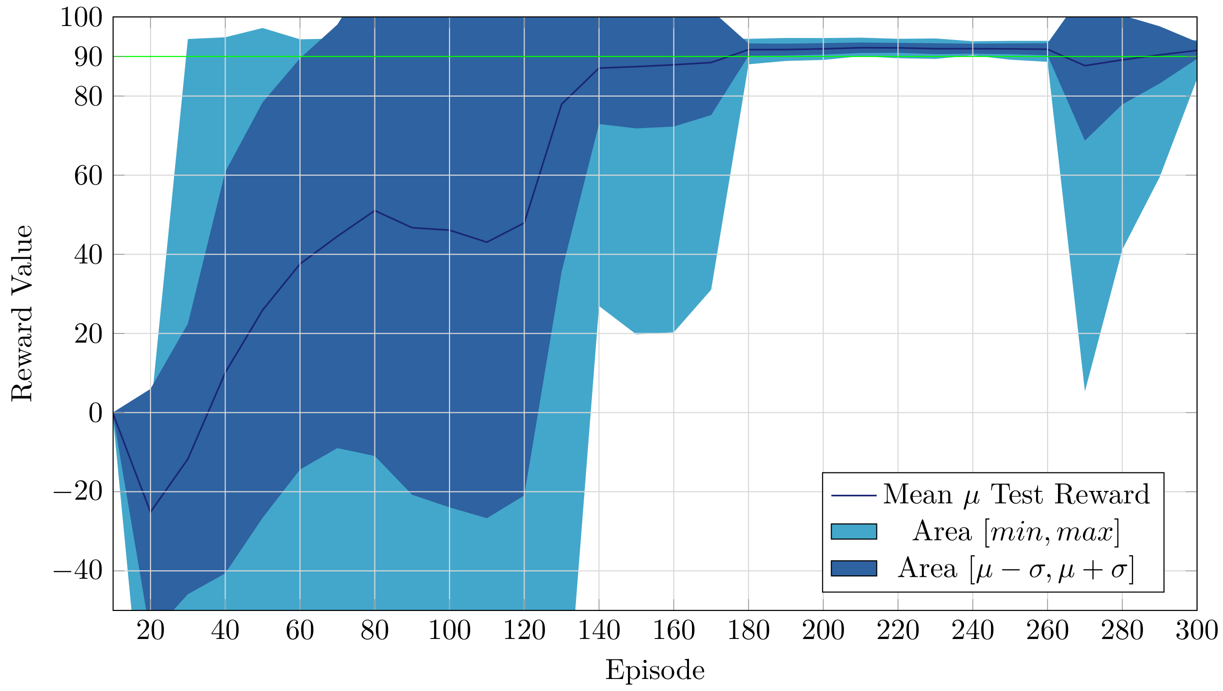


Figure 9: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

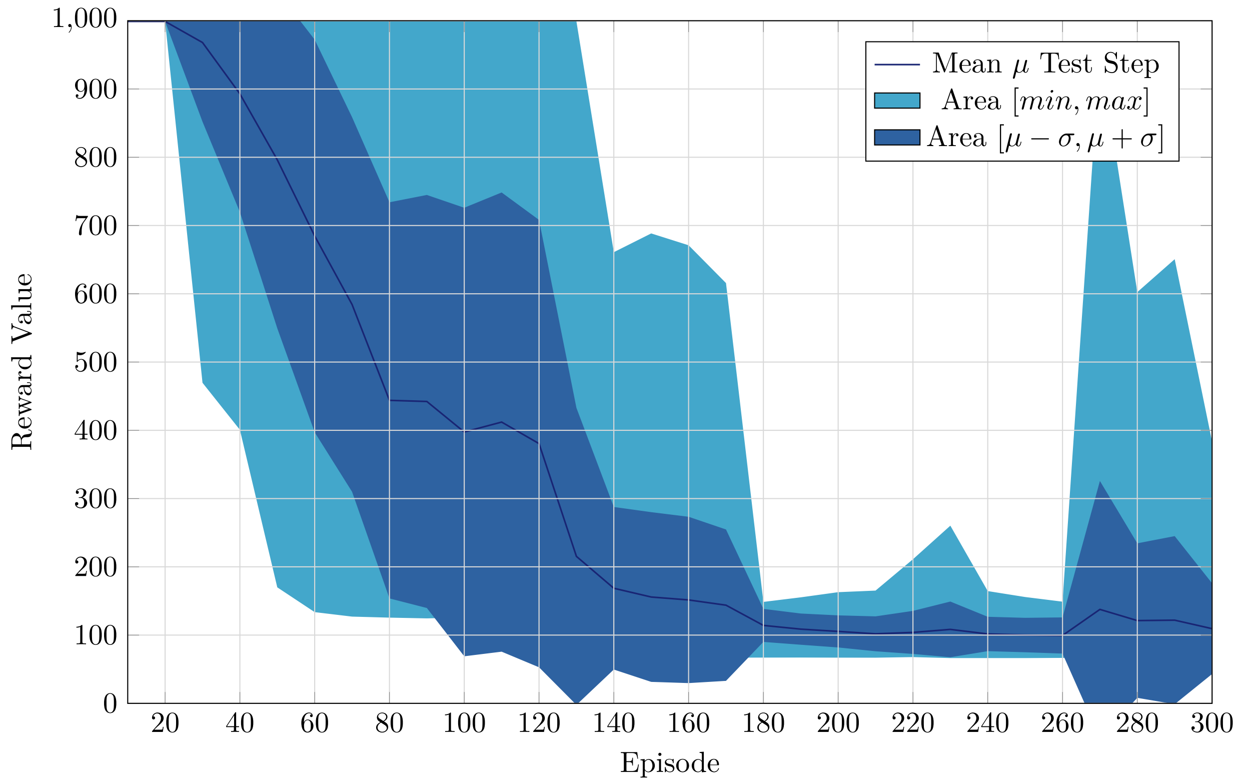


Figure 10: Mean, Standard Deviation Range and Min-Max range of the number of steps in the test phase (every 10 episodes) over 20 runs.

4.3 Pendulum-v0

In this case the graph about the steps taken in the test phase is not useful, because all episodes took all 200 steps.

4.3.1 DDPG with Uniform Replay Memory

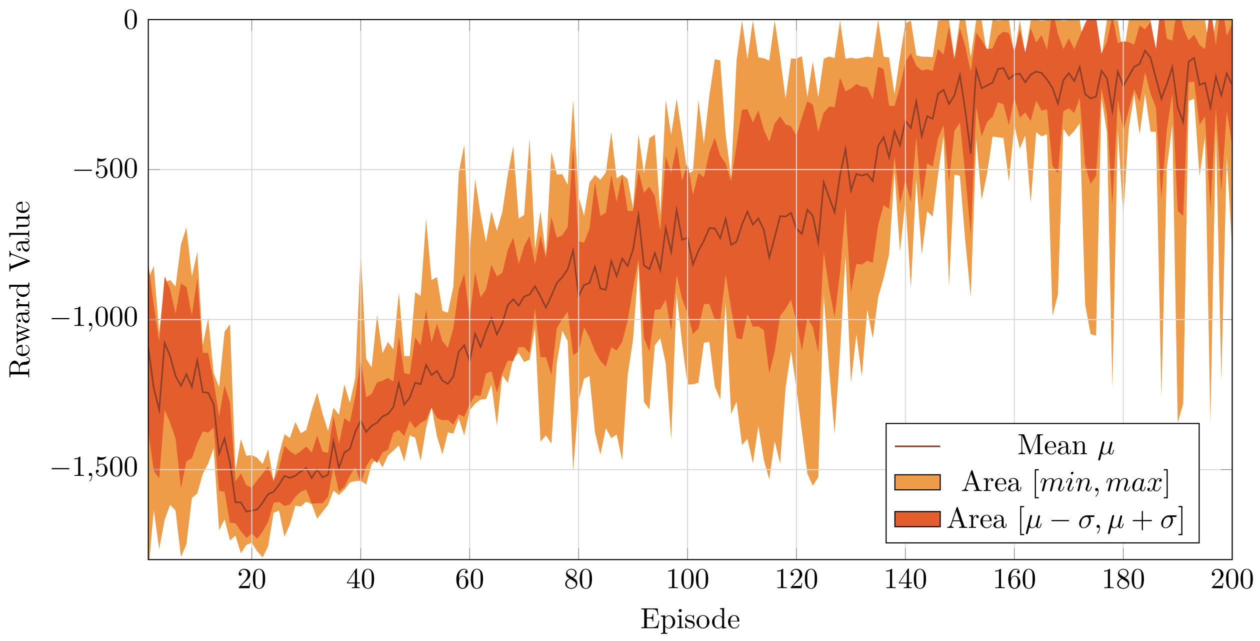


Figure 11: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

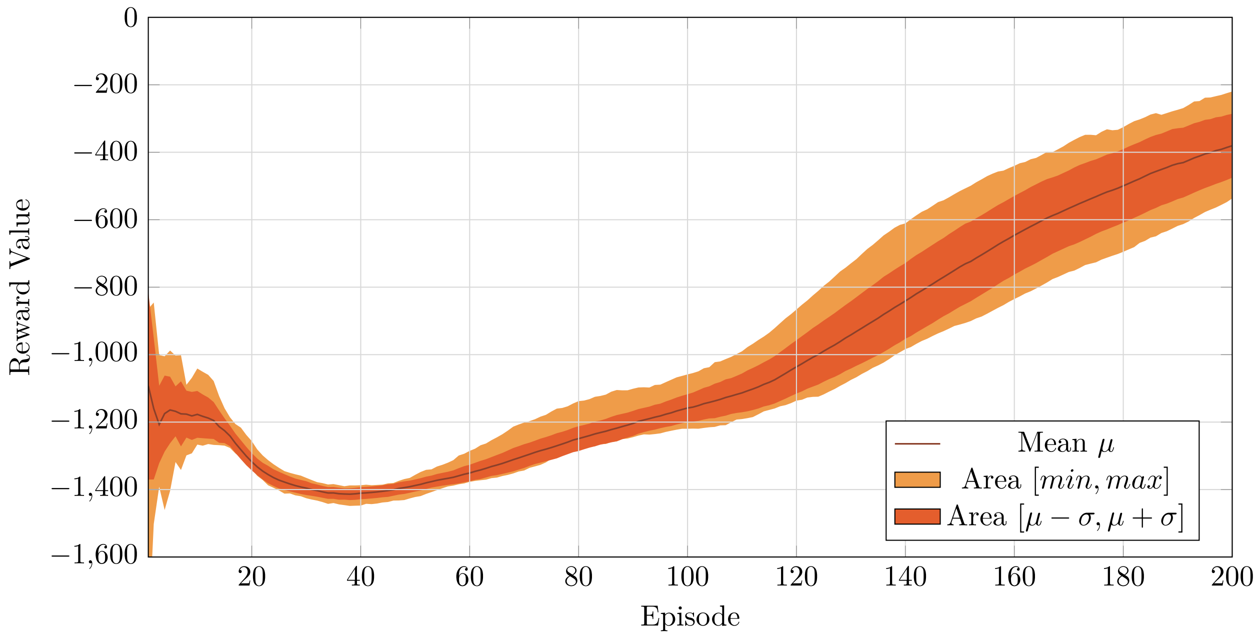


Figure 12: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

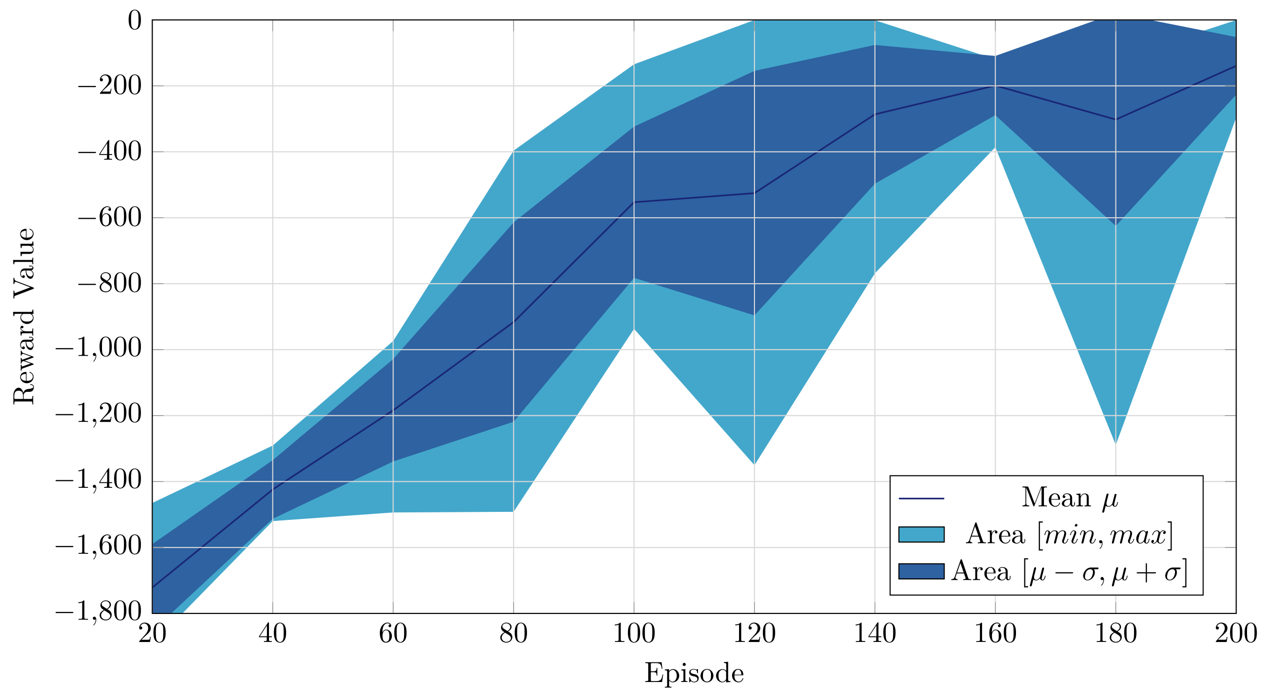


Figure 13: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 10 runs.

4.3.2 SAC

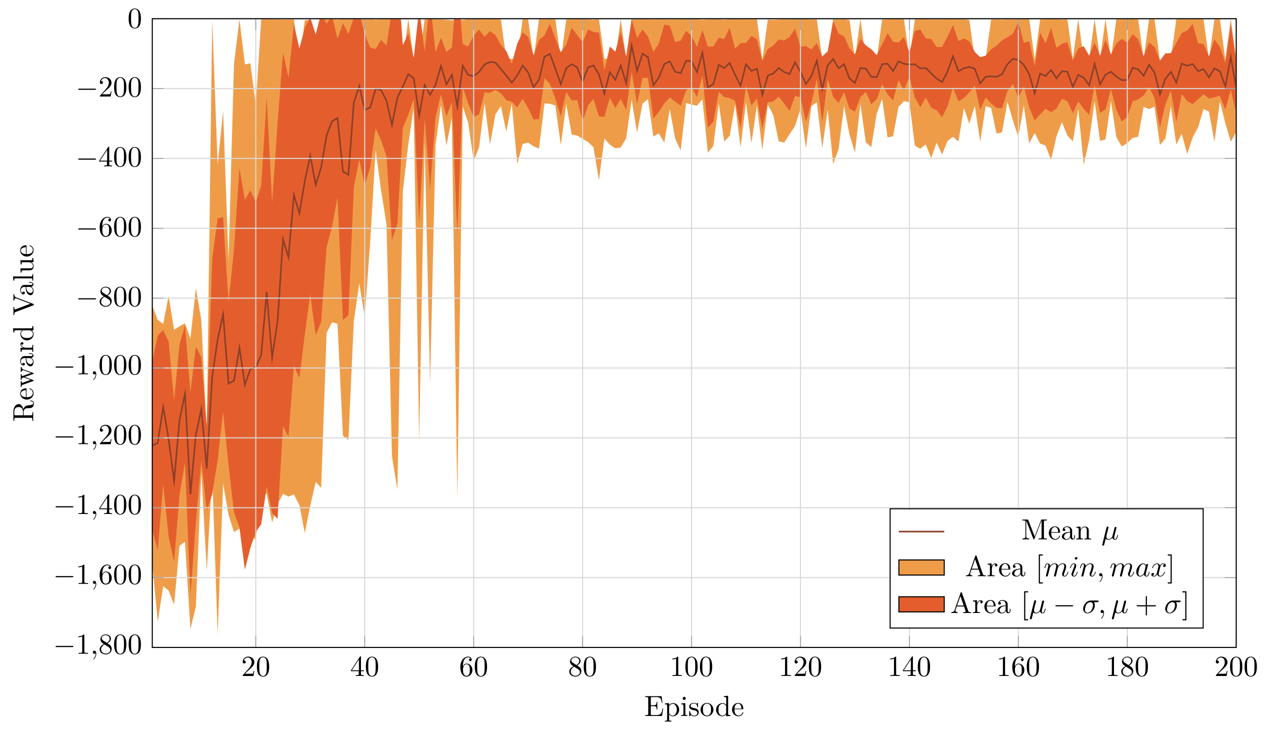


Figure 14: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

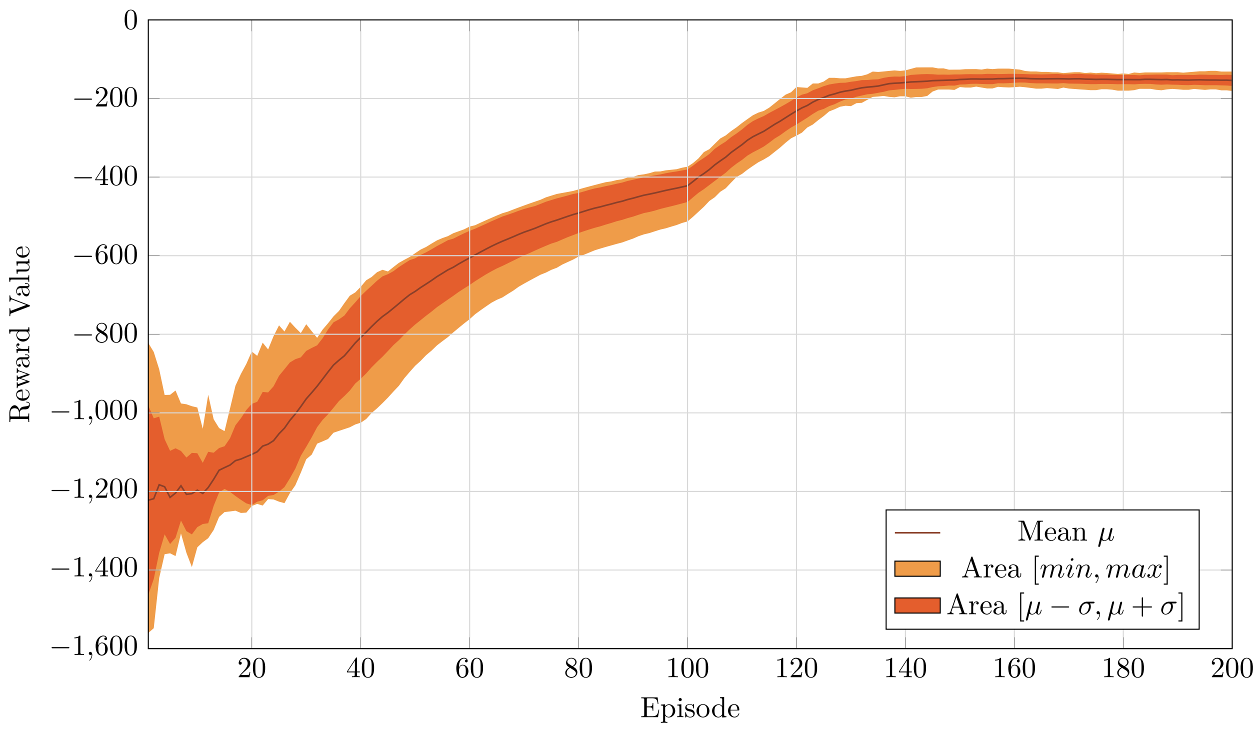


Figure 15: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

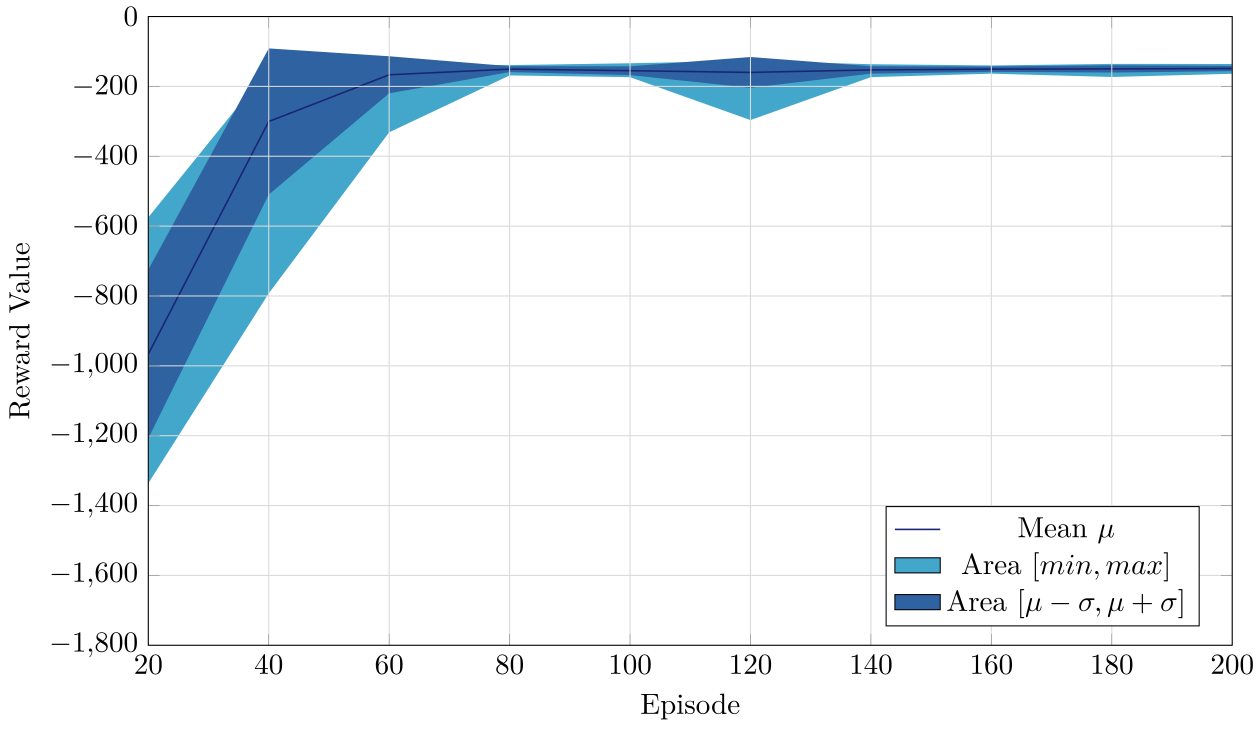


Figure 16: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 20 episodes) over 10 runs.

4.3.3 SAC with Autotune

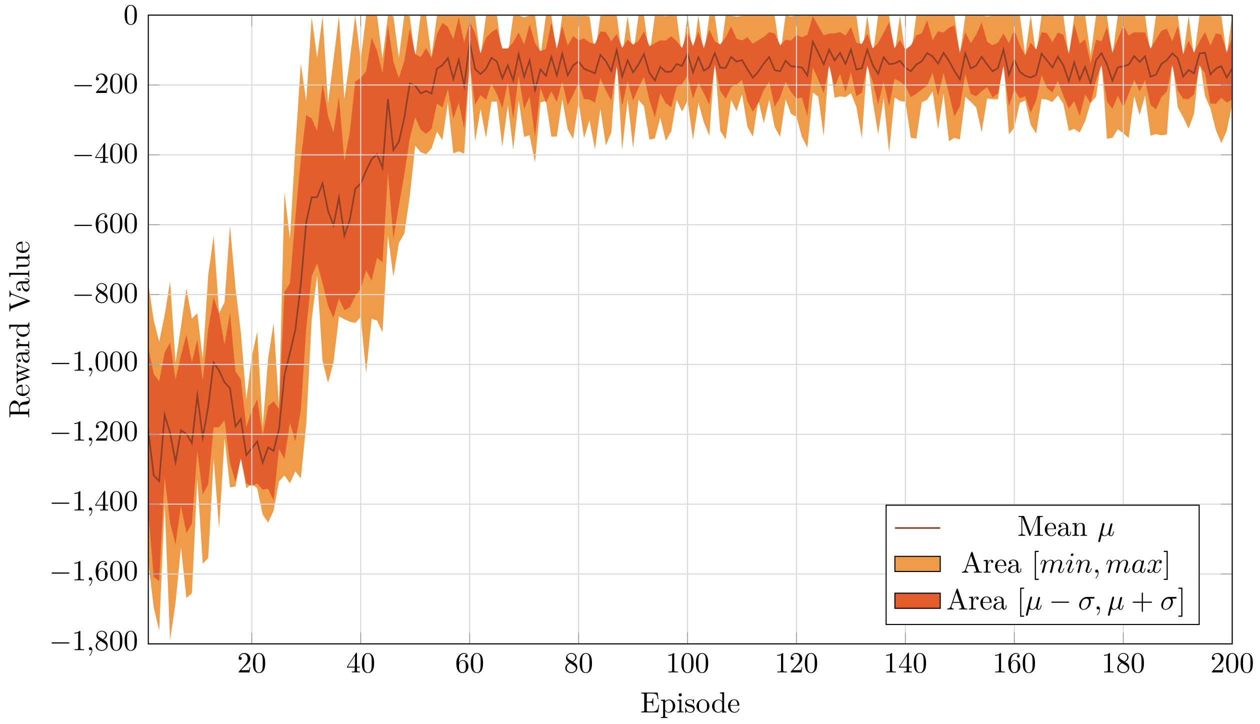


Figure 17: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 10 runs.

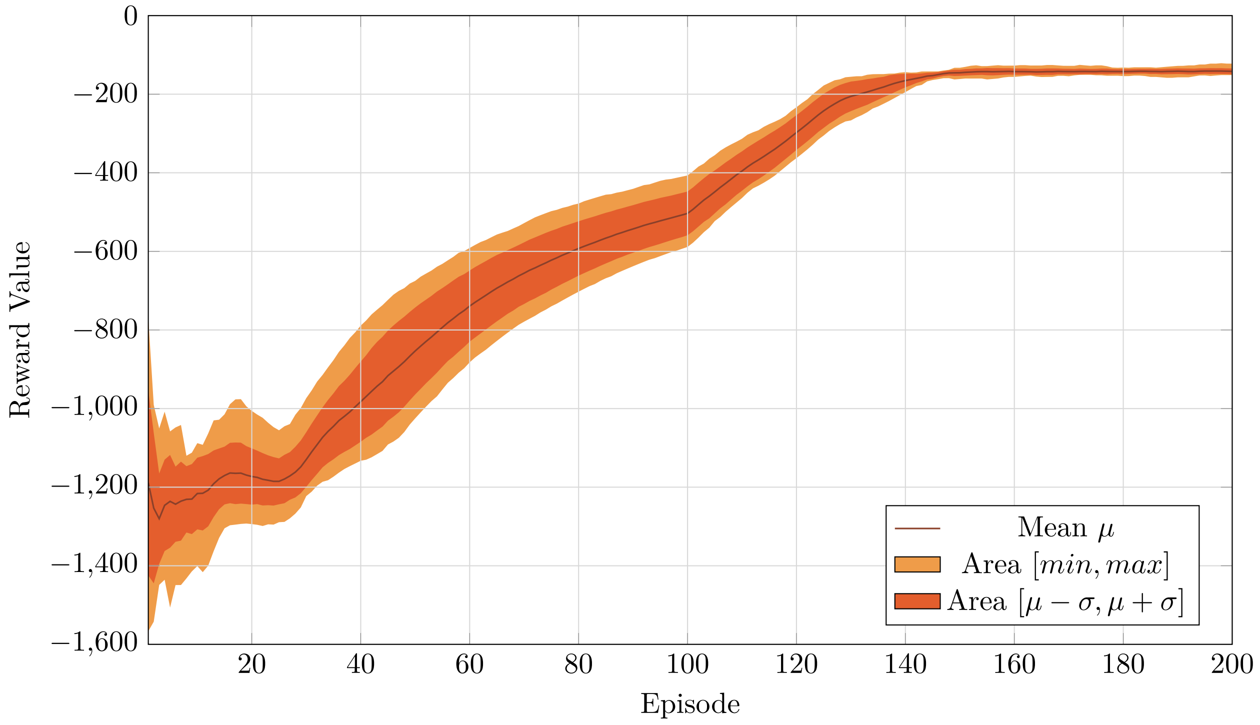


Figure 18: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 10 runs.

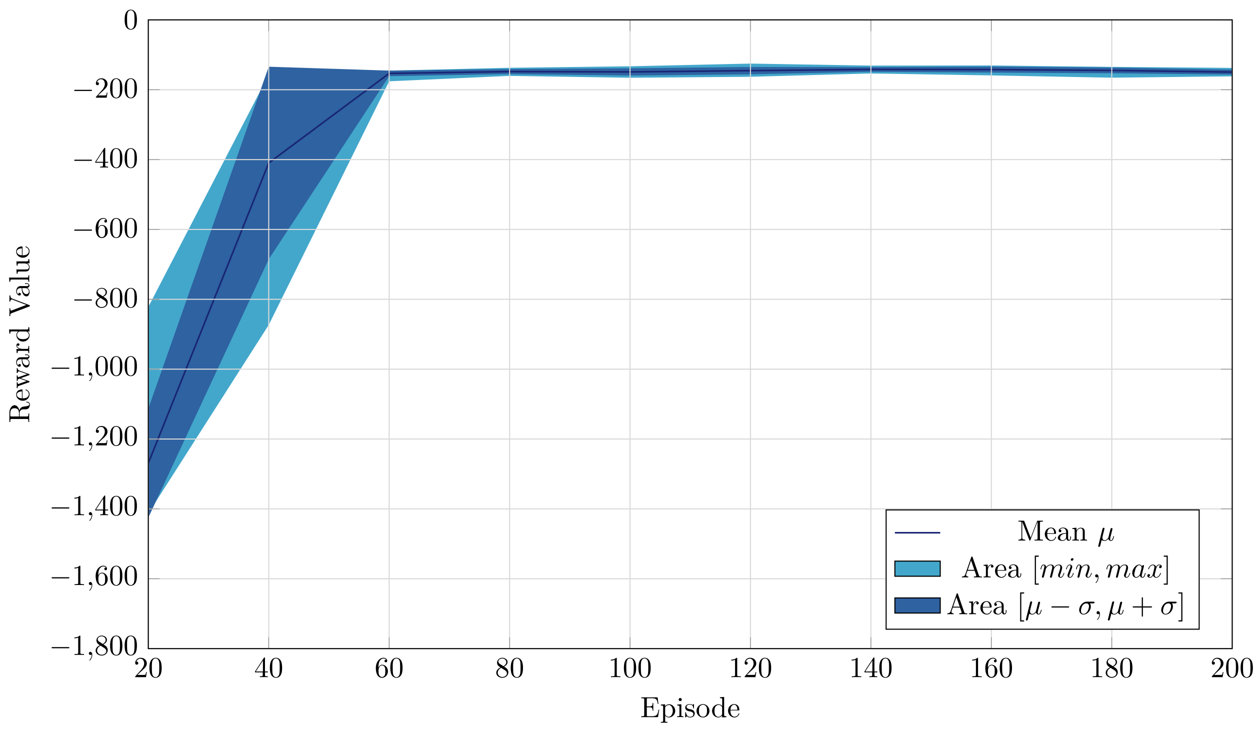


Figure 19: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 20 episodes) over 10 runs.

5 Comments

Analyzing the graphs, it is clear that the results in moving from the Replay Buffer to the Prioritized Buffer are not as impressive as it was expected.

The results obtained are almost the same or slightly better, 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.

In the *MountainCarContinuous-v0* environment, the mean on the last 100 episodes shown in fig. 8 on page 15 is not better than fig. 4 on page 13, especially around 150th episode. But above the 260th episode the results are more stable and efficient. Also comparing the graphs of test phase, the results do not strongly improve.

In the *Pendulum-v0* environment the situation is overall the same, without any important improvement. In some cases the Uniform Replay Buffer seems more efficient in the first part, while the Prioritized one seems more stable in the last episodes.

The aim of the Prioritized Buffer is to give more importance to surprising results. For this reason, I think that having worse results in the first part of the training (0-100 episodes) is normal and it is part of the algorithm. I expected impressive results in the last part, but unfortunately I obtained only slightly better results.

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] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

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