A Study of Reinforcement Learning **Report DDPG**

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

Version: 1.0 from April, 16 2019

Contents

Revision History

Revision	Date	Author(s)	Description
1.0	April, 16 2019	PM	- Created. - Added DDPG description and implementation. - Added Environments Description - Added Uniform/Prioritized Replay implementation and simulation.

1 Introduction

The path of my master thesis began with a study of the foundations of Reinforcement Learning through a series of video lectures and slides of the course [1] held by prof. David Silver and reading some chapters from [1].

Time was spent on the implementation of some algorithms presented in the video lectures to better understand the theory and to get started with the Reinforcement Learning framework **OpenAI Gym** [2]. This is the framework that will be used to implement the project of my thesis.

In the last weeks, the main focus was on a more careful and accurate analysis of the paper [3] from which the development of the thesis starts, and the references it contains, including the paper about **Deep Deterministic Policy Gradient algorithm (DDPG)** [4]. Starting from these, my main goal was to implement the DDPG in Python along the lines of the one present in [3] and [4] so that it can be tested, in an initial stage, with an environment less complex provided by the framework [2].

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

2 Deep Deterministic Policy Gradient

2.1 Application Field

This algorithm can be applied to situations that can not be solved using **DQN** algorithm [5] because of the presence of a continuous action spaces. A finely discretization of the action space to adapt the situation to DQN would lead to an explosion in the number of discrete actions and to the curse of dimensionality.

DDPG is an algorithm which uses deep function approximators in order to learn policies in high-dimensional, continuous action spaces and it is:

a) Model-Free: it has no prior knowledge about model, transitions or rewards;
b) Off-Policy: the policy is learnt from experiences sampled from another policy;
c) Actor-Critic: alternation between a policy evaluation and a policy improvement step.

Bellman equation and **Q-learning** are fundamental parts of this algorithm. It concurrently learns a Q-value function and a policy: it uses off-policy data and the Bellman equation to learn the Q-value function, and uses the Q-value function to learn the policy.

Usually, in Reinforcement Learning, if the optimal action-value function $Q^*(s, a)$ is known, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s, a)$$

When the number of discrete actions is finite, calculating the max poses no problem, because the Q-values can be calculated for each action separately, then directly compared. But when the action space is continuous, this process become highly non-trivial: it would need to be run at every steps of the episode, whenever the agent wants to take an action in the environment and this can not work.

Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action argument. For this reason, an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact can be set up, approximating it with $\max_a Q(s, a) \approx Q(s, \mu(s))$.

[1] The website of the course: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

2.2 Key Points

2.2.1 Reinforcement Learning Setup

The Reinforcement Learning Setup is the standard one. It consists of an agent interacting with an environment E in discrete timesteps. At each timestep t the agent receives an **observation** x_t, takes an **action** a_t and receives a scalar **reward** r_t. In this specific case, the actions are real-valued $a_t \in \mathbb{R}^N$ and it is assumed that the environment is fully-observed so $s_t = x_t$.

The agent's behavior is defined by a policy $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$ which maps states to a probability distribution over the actions. The problem is modeled as a **Markov decision process** with a state space $\mathcal{S}$, action space $\mathcal{A} = \mathbb{R}^N$, an initial state distribution $p(s_1)$, transition dynamics $p(s_{t+1}|s_t, a_t)$, and reward function $r(s_t, a_t)$.

The return is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$ with a discounting factor $\gamma \in [0, 1]$ and depends on the actions chosen, and therefore on the policy π.

The goal in reinforcement learning is to learn a policy which maximizes the expected return from the start distribution $J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$. We denote the discounted state visitation distribution for a policy π as ρ^π.

2.2.2 Learning Equations

One of the fundamental function in Reinforcement Learning, which is used also in DDPG, is the **action-value function**. It describes the expected return after taking an action a_t in state s_t and thereafter following policy π:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_i \geq t, s_i > t \sim E, a_i > t \sim \pi}[R_t | s_t, a_t] \tag{1}$$

DDPG exploits also the **Bellman equation** which is given by

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^\pi(s_{t+1}, a_{t+1})]] \tag{2}$$

where $s_{t+1} \sim E$ means that the next state is sampled from the environment E and $a_{t+1} \sim \pi$ shows that the next action is taken following the policy π. If this policy is deterministic we can describe it as a function $\mu : \mathcal{S} \leftarrow \mathcal{A}$ obtaining

$$Q^\mu(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))] \tag{3}$$

which depends only on the environment. This means that it is possible to learn Q^μ off-policy, using transition generated by a different stochastic behaviour policy β.

Focusing more and more on DDPG, the Bellman equation is the starting point for learning an approximator to $Q^*(s, a)$ of Q-Learning. The approximator is parametrized by θ^Q and the value network is updated and optimized by minimizing the loss:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t | \theta^Q) - y_t)^2] \tag{4}$$

where

$$y_t = r(s_t, a_t) + \gamma(1 - d_t) Q(s_{t+1}, \mu(s_t + 1) | \theta^Q) \tag{5}$$

where d_t is a flag which indicates whether state s_{t+1} is terminal.

It is clear from the eq. (4) that the loss is calculated starting from the transitions generated by the policy β. For this reason a great importance in this algorithm is given to **Replay Buffer** and **Target Networks**.

For the policy function, the objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}[Q(s,a)|_{s=s_t, a_t=\mu(s_t)}] \tag{6}$$

To calculate the policy loss, the derivative of the objective function with respect to the policy parameter is needed:

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_{\theta^\mu} \mu(s|\theta^\mu) \tag{7}$$

But since the algorithm is updating the policy in an off-policy way with batches of experience, the mean of the sum of gradients calculated from the mini-batch can be used:

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i [\nabla_a Q(s,a)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}] \tag{8}$$

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)
```

2.2.3 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** [6] 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 11 the results of the two approaches will be analyzed.

2.2.4 Target Networks

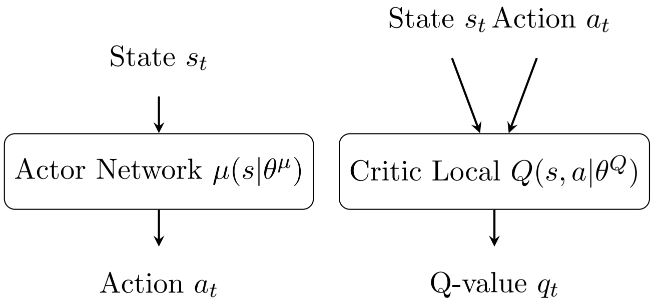
State s_t
Actor Network $\mu(s|\theta^\mu)$
Action a_t
State s_t Action a_t
Critic Local $Q(s,a|\theta^Q)$
Q-value q_t

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. (9)) and to update the actor policy using the sampled policy gradient (eq. (8) on the preceding page).

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{9}$$

where y_i is given by eq. (5) on page 4.

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 8

2.2.5 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 [4], the authors use Ornstein-Uhlenbeck Process [7] 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 [4] 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 next 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 11 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.

```
Algorithm 1: DDPG Algorithm
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$;
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$;
for episode = 1, M do
    Initialize or Reset a random process $\mathcal{N}$ for action exploration;
    Reset the environment and receive the initial observation state $s_1$;
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}$ and clipping it to the limit of the environment;
        Execute action $a_t$ and obtain $(r_t, s_{t+1}, d_t) \equiv$ (reward, next state, done flag);
        Store transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{R}$;
        if $\mathcal{R}$ has at least $M$ samples then
            Sample random minibatch of $N \ll M$ transitions $(s_t, a_t, r_t, s_{t+1}, d_t)$ from $\mathcal{R}$;
            Set $y_i = r_i + \gamma(1-d_i)Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$;
            Update the critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$;
            Update the actor policy using the sampled policy gradient:
                $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
            Update the target networks:
                $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
                $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
        end
    end
end
```

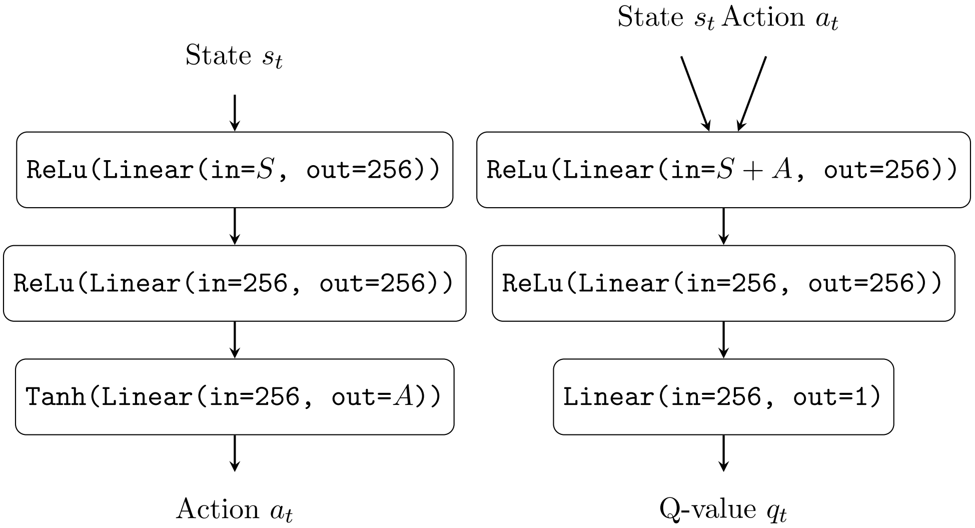
State s_t
ReLu(Linear(in=S, out=256))
ReLu(Linear(in=256, out=256))
Tanh(Linear(in=256, out=A))
Action a_t
State s_t Action a_t
ReLu(Linear(in=$S+A$, out=256))
ReLu(Linear(in=256, out=256))
Linear(in=256, out=1)
Q-value q_t

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.

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 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

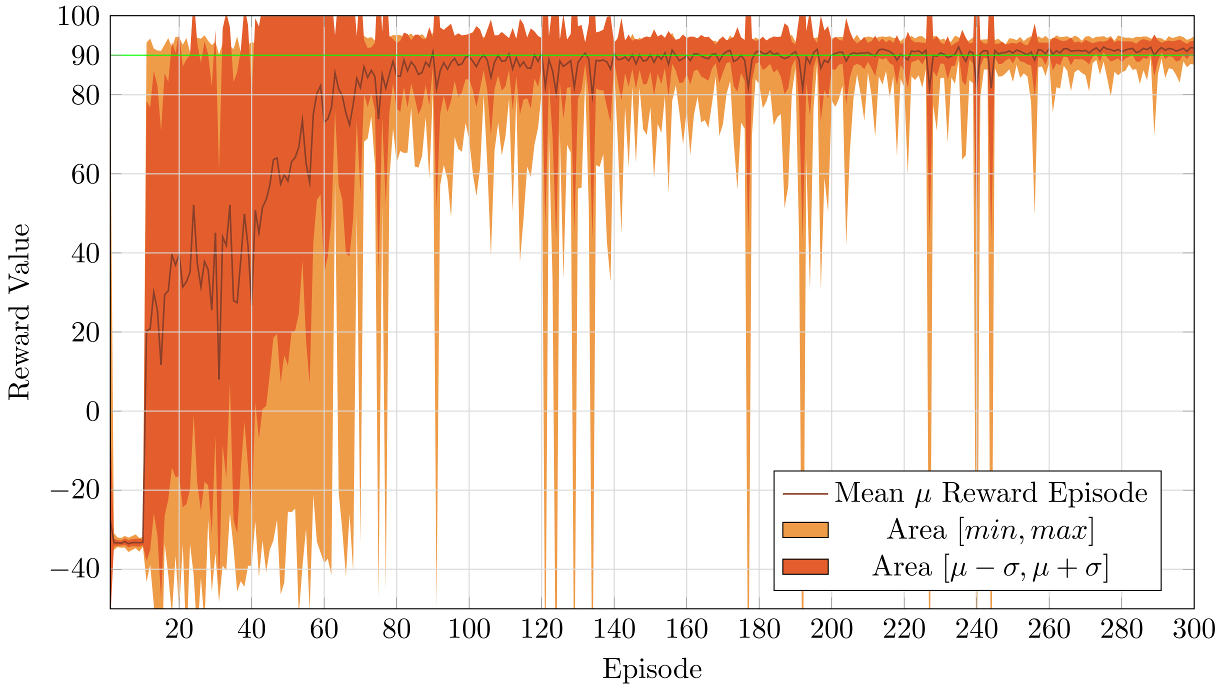

Figure 3: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

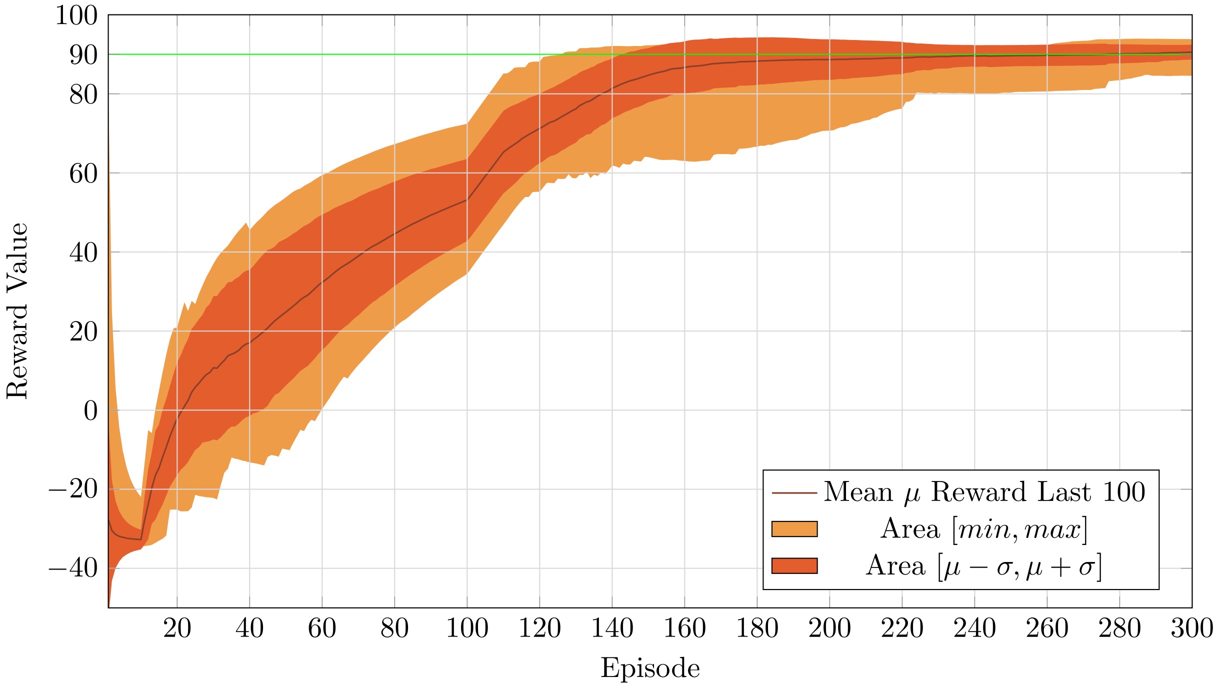

Figure 4: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

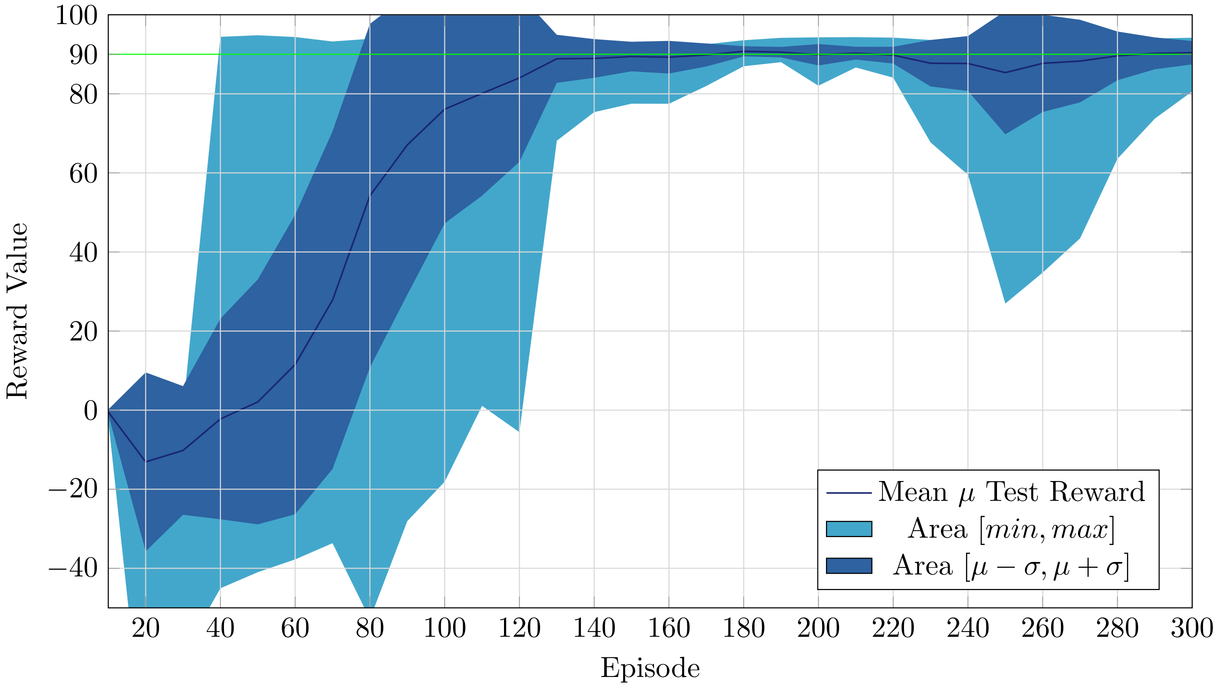

Figure 5: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

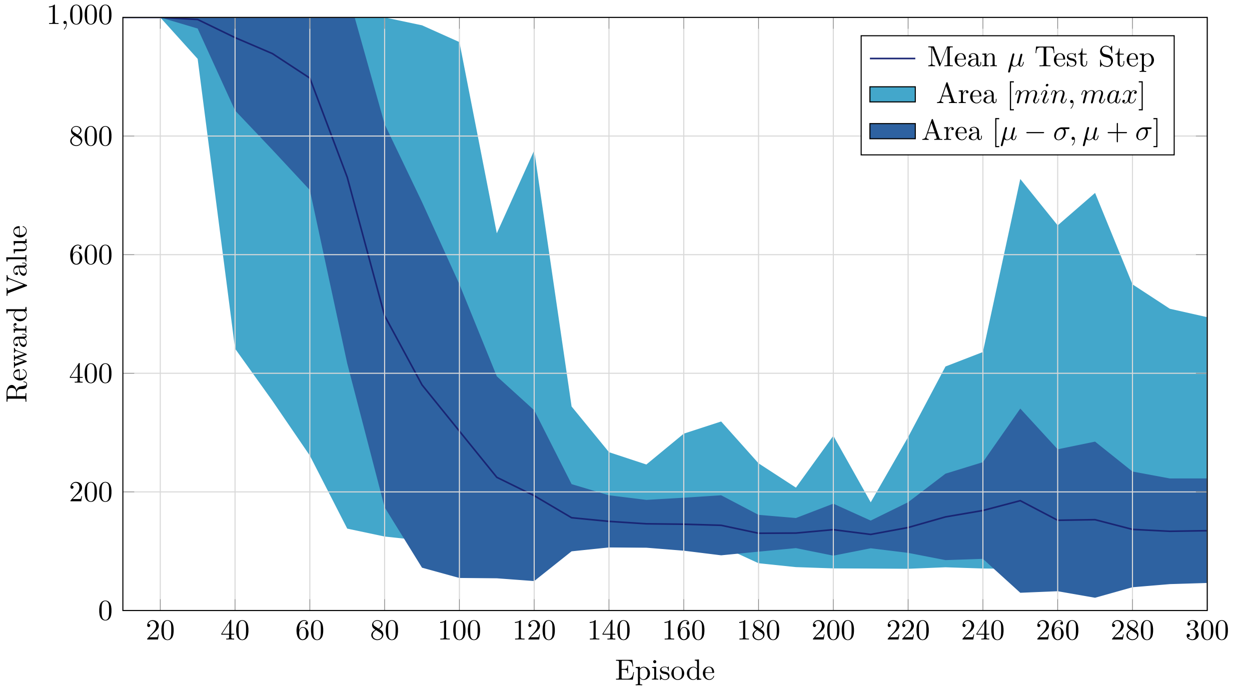

Figure 6: Mean, Standard Deviation Range and Min-Max range of the number of steps in the test phase (every 10 episodes) over 20 runs.

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.

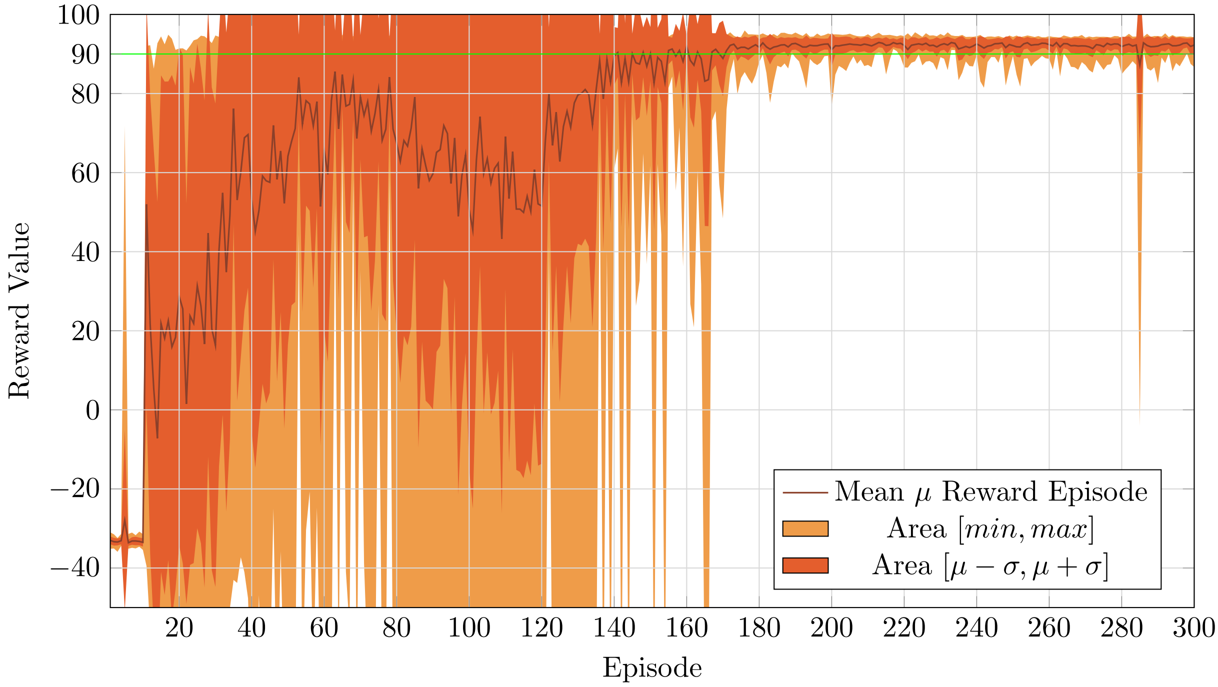

Figure 7: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

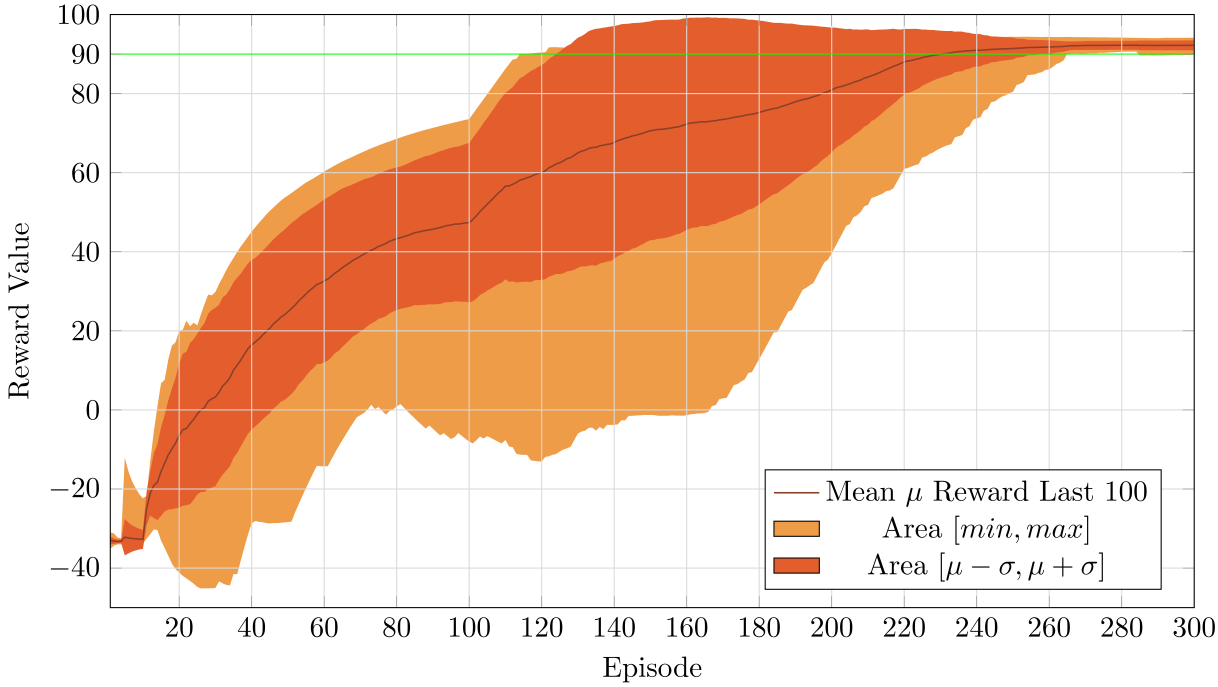

Figure 8: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

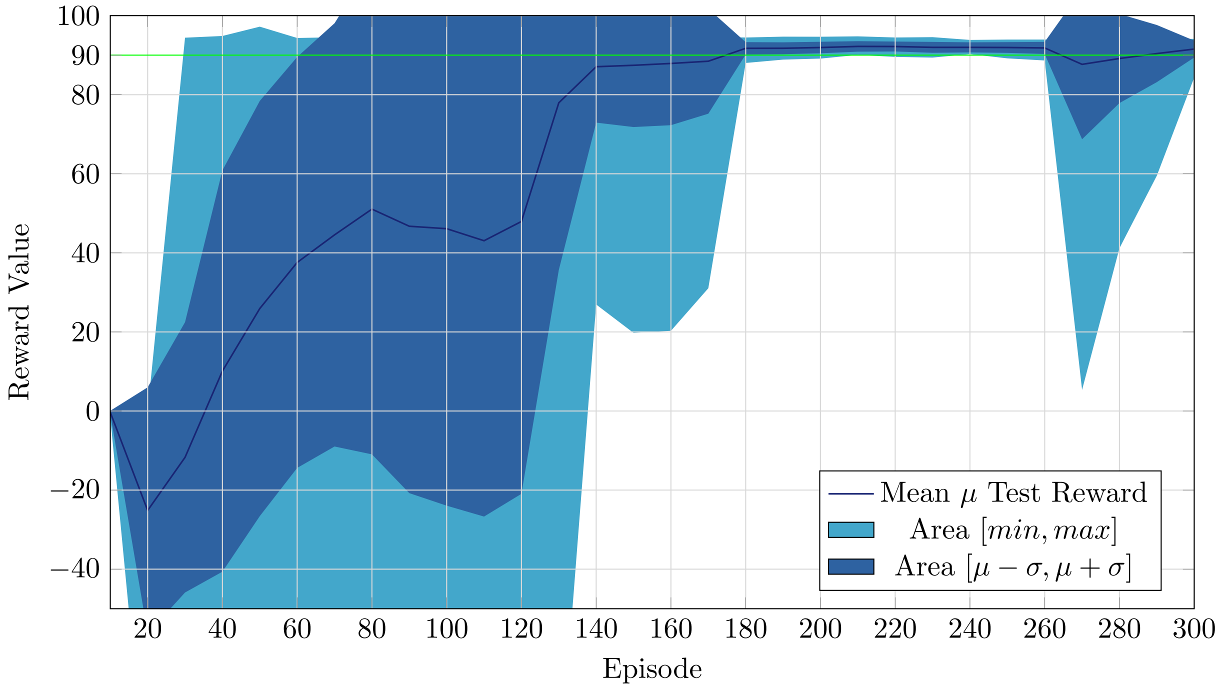

Figure 9: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

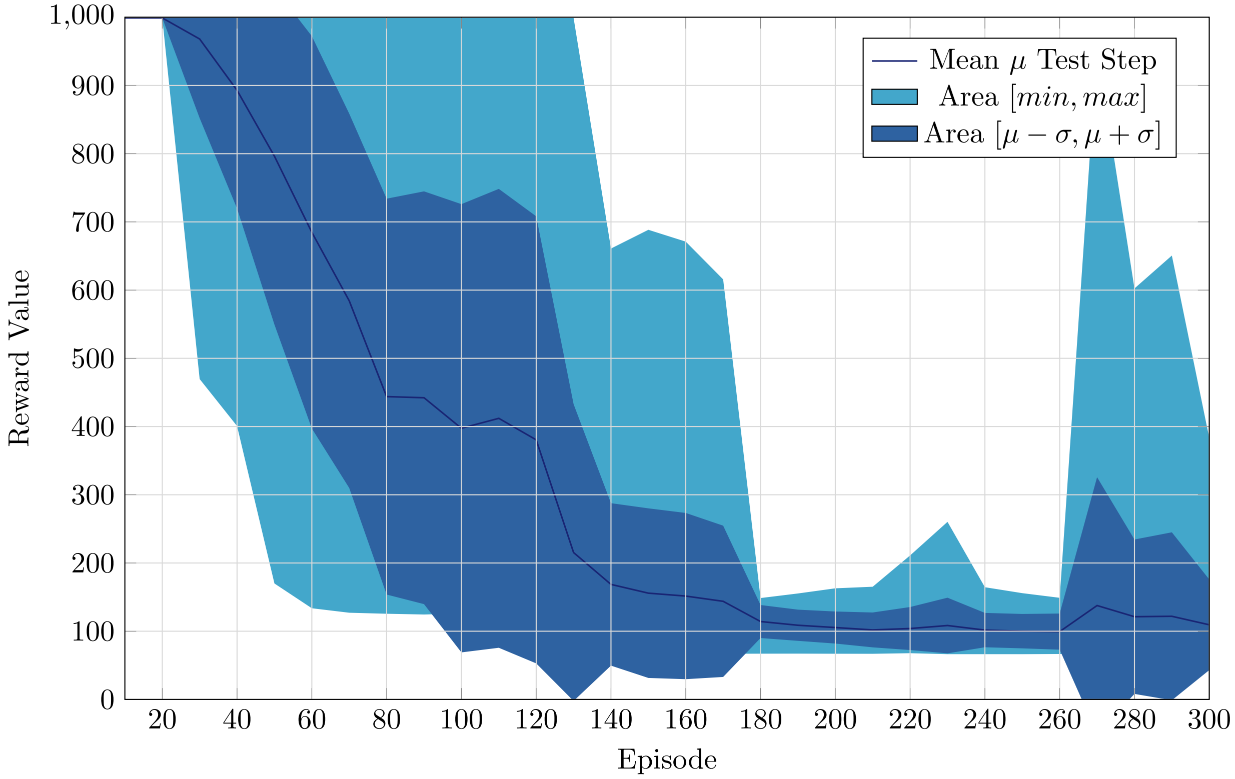

Figure 10: Mean, Standard Deviation Range and Min-Max range of the number of steps in the test phase (every 10 episodes) over 20 runs.

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 Uniform Replay Memory

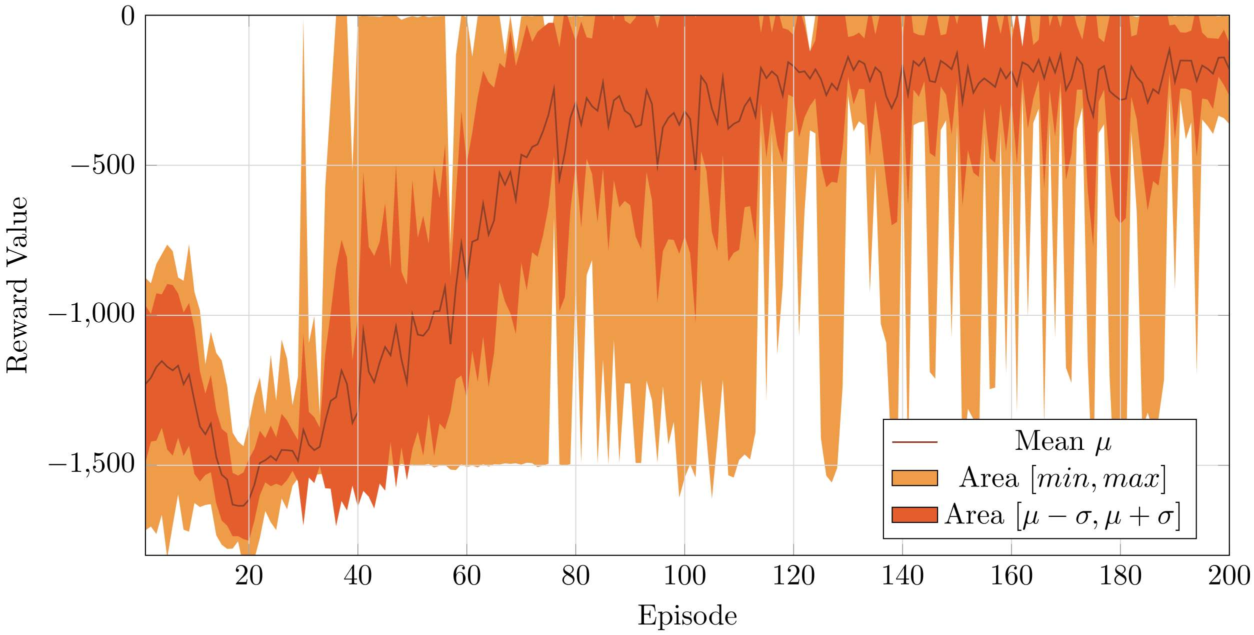

Figure 11: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

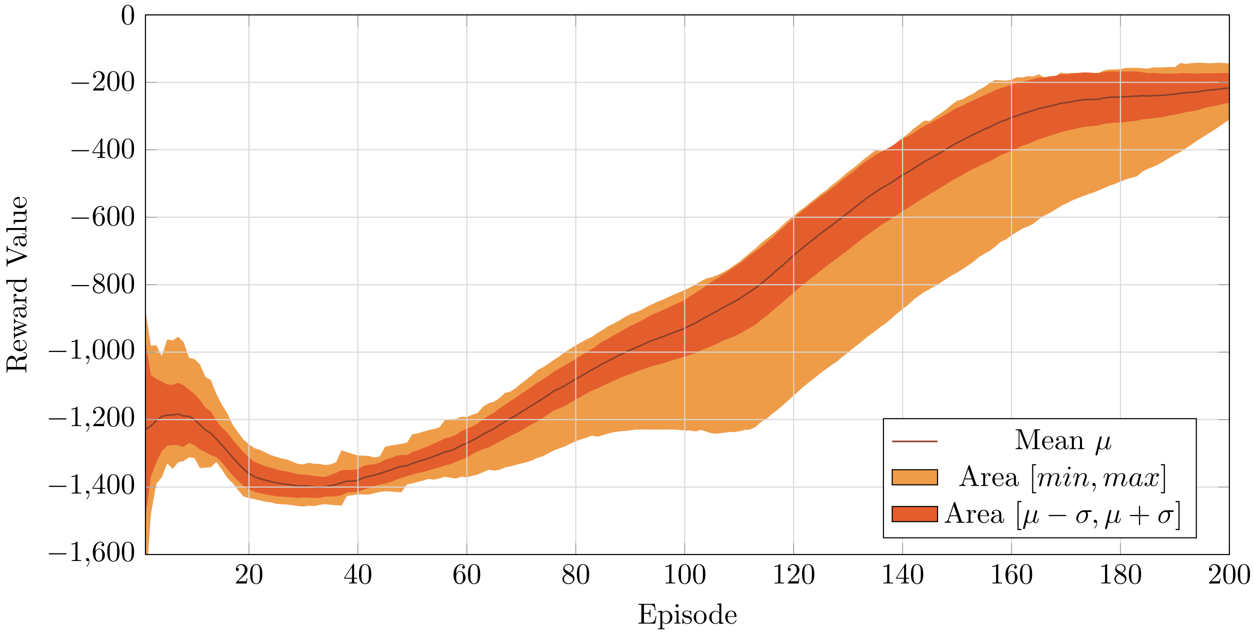

Figure 12: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

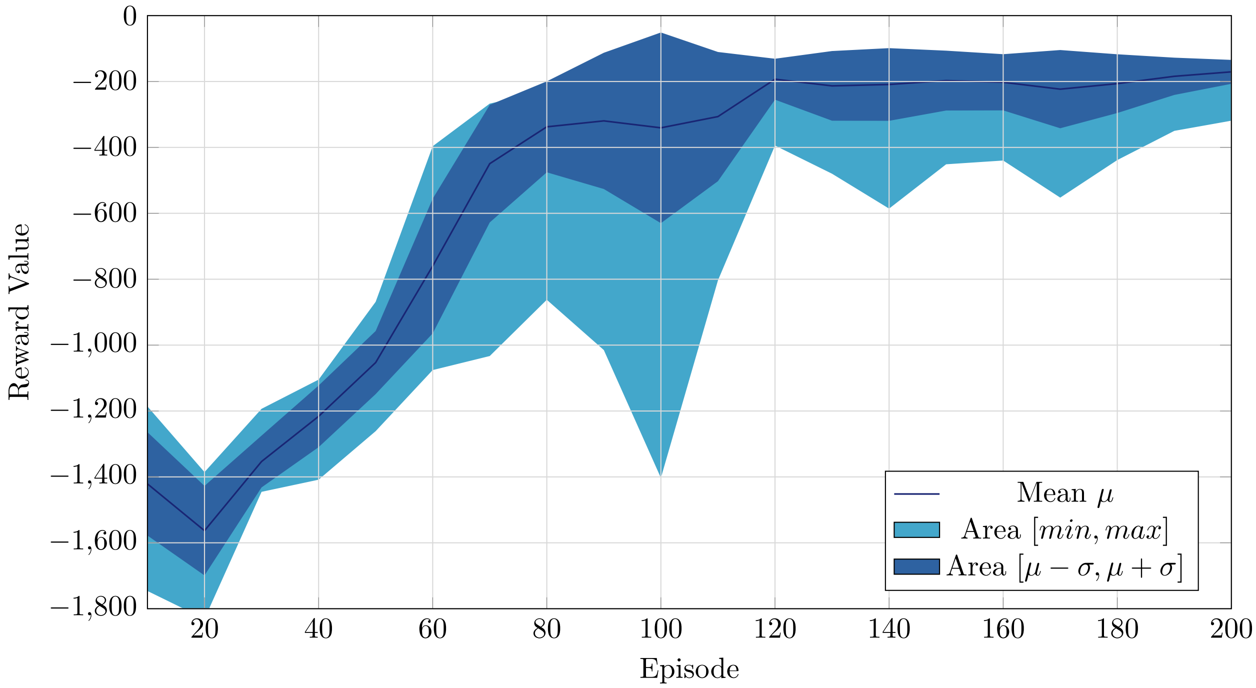

Figure 13: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

4.3.2 Prioritized Replay Memory

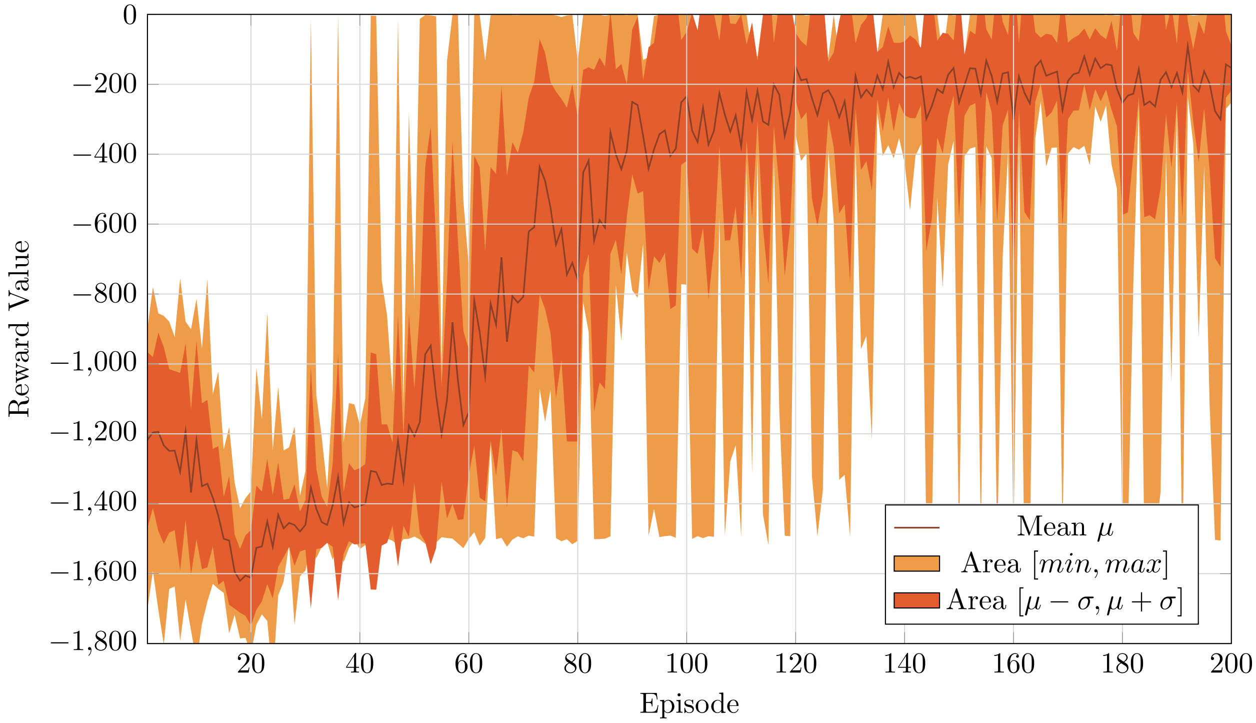

Figure 14: Mean, Standard Deviation Range and Min-Max range of the reward of each episode over 20 runs.

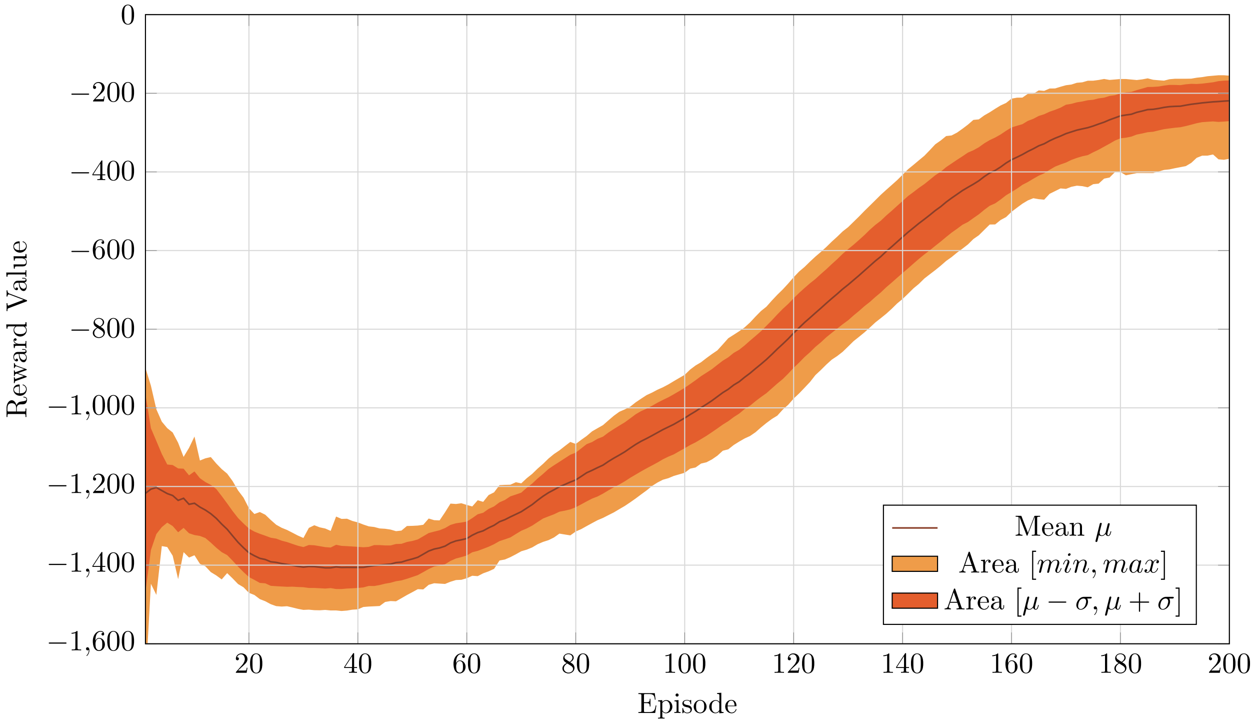

Figure 15: Mean, Standard Deviation Range and Min-Max range of the running reward mean of the last 100 episodes for each episode over 20 runs.

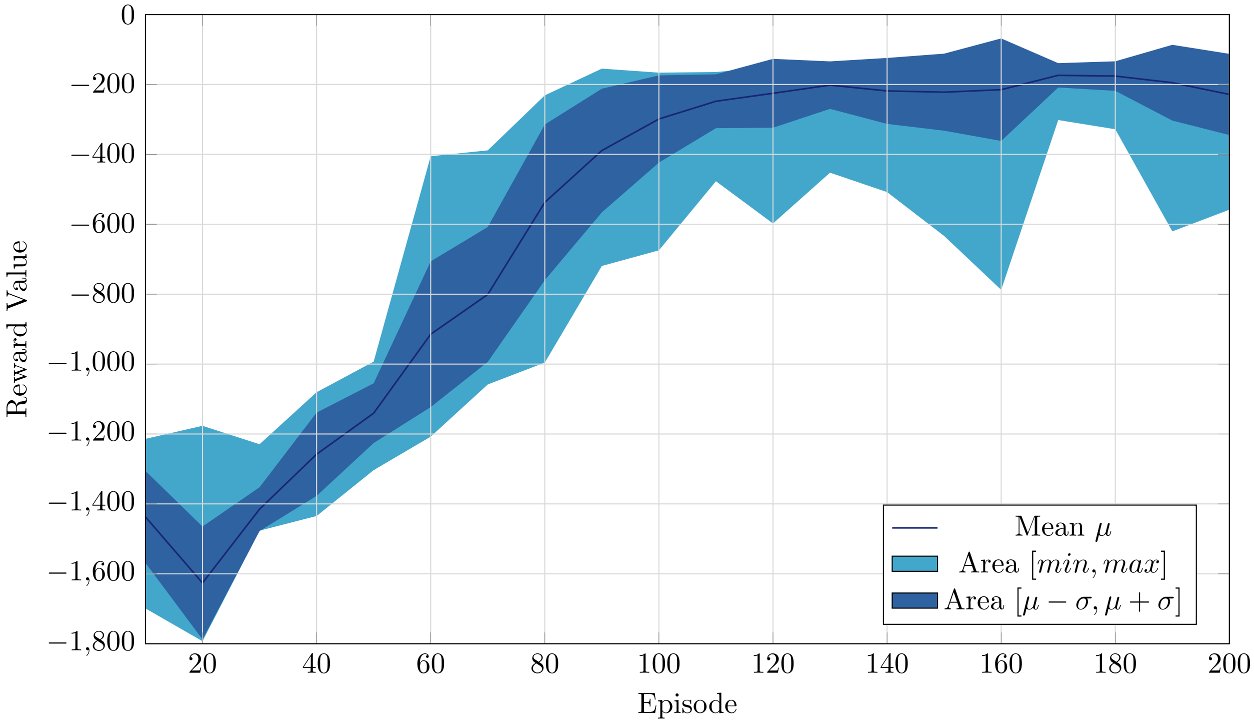

Figure 16: Mean, Standard Deviation Range and Min-Max range of reward mean of the test phase (every 10 episodes) over 20 runs.

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 14 is not better than fig. 4 on page 12, 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] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. *arXiv preprint arXiv:1606.01540*, 2016.

[3] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *arXiv preprint arXiv:1807.00412*, 2018.

[4] 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.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

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

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