Practical Course Report

Implementation of Deep Deterministic Policy Gradient

Arash Torabi Goodarzi Zofia Antonina Bentyn

Date of Submission: 01. March 2024
Advisor: Sebastian Griesbach, M. Sc.

Julius-Maximilians-Universität Würzburg
Department of Mathematics and Computer Science
Chair of Reinforcement Learning and Computational Decision-Making

1 Introduction

Reinforcement Learning (RL) has been one of the main and most successful learning approaches in the recent years, whenever the interaction of an intelligent agent to its environments is regarded.

The Deep Q Network (DQN) algorithm has been able to gain tremendous success and human level performance in certain game environments with discrete action spaces. This algorithm can however not be applied to learn a policy for environments with continuous action spaces in a straightforward manner, since the approximated policy in DQN is an action-value function maximizing approach, which in continuous fields would require an iterative optimization process in every learning step. [LHP+15]

T. Lillicrap et al. have provided a new RL approach specific to environments with continuous action spaces called Deep Deterministic Policy Gradient (Deep DPG or DDPG). [LHP+15]

As the first part of our eXtended Artificial Intelligence Lab program, we have implemented the DDPG algorithm and have used it to train an intelligent agent for several different environments provided by Farama Foundation's Gymnasium project including several MuJoCo environments.

In Chapter 2 of this report, we will discuss and go through some basic preliminaries and the needed background from Q-Learning and and DQN to then discuss the DDPG algorithm in Chapter 3.

In Chapter 4 we will go through the structure of our implementation and explain the usage of different classes as well as explain the implemented training function which is the core of the DDPG implementation.

Lastly, in Chapter 5 we will present our experimental results from different environments and discuss the advantages and shortcomings of DDPG.

2 Preliminaries and Background

In this chapter, we will lay some groundwork and determine our writing conventions and definitions. Next, we will explain the DQN algorithm as the background for DDPG and discuss its shortcomings concerning continuous action spaces.

2.1 The Setup of RL

The agent's interaction with its environment is modelled as a Markov Decision Process (MDP).

Definition 1 (Markov Decision Process)**.** *A Markov Decision Process consists of an environment and an agent. The environment specifies a set of states S, referred to as the observation space, a set A of available actions, known as the action space, and a reward function $R : S \times A \to \mathbb{R}$. During an episode at each time-step $t \in \{0, 1, 2, \dots\}$ the agent perceives a state $s_t \in S$ and selects an action $a_t \in A$. The environment then transitions into a new state $s_{t+1} \in S$ and provides the agent with a reward $R_{t+1} \coloneqq R(s_t, a_t)$. Markov decision process is a 4-tuple (S, A, P_a, R_a), where the P_a stands for the probability of process moving to the new state s' via chosen action.*

For a transition from state s using an action a to state s' and getting reward r the probability $\mathbb{P}(s', r|s, a)$ is defined as follows:

$$\mathbb{P}(s', r|s, a) = \mathbb{P}(s_t = s', R_t = r|s_{t-1} = s, A_{t-1} = a)$$

for all $s' \in S$, $r \in R$, $s \in S$ and $a \in A$.
At each time-step t the return G_t is the sum of all the future rewards

$$G_t = \sum_{i=t+1}^{T} R_i = \sum_{i=0}^{T-t-1} R_{t+i+1},$$

where T is the final time-step. An agent strives to maximize this return. For continuous tasks, where $T = \infty$, the discounted return is considered instead. This is defined as

$$G_t \coloneqq \sum_{i=0}^{\infty} \gamma^i \cdot R_{t+i+1} = R_{t+1} + \gamma \cdot G_{t+1}$$

2.2 Policies and Value Functions

The behavior of an agent is determined by the policy it follows. A policy is a function $\pi : S \times A \to [0, 1]$ that maps a state-action pair $(s, a) \in S \times A$ to a probability $\pi(s, a)$.

That is the probability, that the agent will choose action a when state s is encountered. For a fixed $s \in S$, π_s denotes a probability distribution over all of the available actions $A(s)$ in state s. If a policy uses some parameters, the resulting policy for a given vector of parameters θ is denoted as $\pi^{(\theta)}$.

To further examine the behavior of an agent, *value functions* with regard to the policy are considered. These describe the expected return of an agent, when it is following a given policy. There are two typical types of value functions of which include *state value function* and *state-action value function.* The state value function describes how good any given state is, for an agent following π. Formally:

Definition 2 (State Value Function)**.** *Given a policy π, the state value function $V_\pi : S \to \mathbb{R}$ of π is a function mapping a given state s to the expected return of an agent following π starting at state s. More specifically*

$$V_\pi(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1}|S_t = s]$$

for $s \in S$.

taking action a in state s and follow policy pi afterwards.

The state-action value function describes how good it is for an agent taking action a in state s and follow policy π afterwards. Formally:

Definition 3 (State-Action Value Function)**.** *Given a policy π, the state-action value function $q_\pi : S \times A \to \mathbb{R}$ of π is a function mapping a state action pair (s, a) to the expected return of an agent following π, when taking action a at starting state s. More specifically*

$$q_\pi(s, a) = \mathbb{E}[G_t|s_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1}|s_t = s, A_t = a]$$

for $(s, a) \in S \times A$.

For policies π and π', we write $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$ and we consider π as an optimal policy if and only if $\pi \geq \pi'$ for every possible π'. Accordingly, an optimal state value function V_* is defined as

$$V_*(s) = \max_\pi V_\pi(s)$$

or for an optimal policy π' $V_*(s) = V_{\pi'}(s)$. Similarly, an optimal state-action value function q_* is defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

or for an optimal policy π', $q_*(s, a) = q_{\pi'}(s, a)$. The optimal state-action value function is also called the *optimal Q-function* and its value for a given state action pair is called the *Q-value.* Which can be shown using the Bellman equation [BBC57].

Theorem 4 (Bellman Equation)**.** *Let $s \in S$ be the state of the environment at time t, for the optimal action value function q_* it holds*

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \cdot \max_{a'} q_*(s', a')]$$

for any action $a \in A$, with s' being the state at which the maximum action a' is applicable.

2.3 Q-Learning and DQN

If an optimal Q-function is given, we can derive an optimal policy from it by finding the best action for every state. Q-learning is an iterative off-policy technique for finding such optimal Q-function. The strategy of the algorithm is to explore the environment and find the best Q-value for every state action action pair using a balance of exploration and exploitation. Using Theorem 4, the Q-function, stored for a state action pair $(s, a) \in S \times A$, is updated at each training episode with the formula

$$q(s, a) \leftarrow (1 - \alpha)q(s, a) + \alpha \cdot [R + \gamma \max_{a} q(s', a')]$$

with current reward R and for some learning rate $\alpha \in (0, 1)$. Second part of the equation, more specifically the estimated discounted reward for future steps, defines temporal difference target, with which the Q-values get updated until the optimum is obtained. See chapter 6.5 of [SB18] for an extensive explanation as well as a pseudocode of the Q-learning algorithm.

The idea for the DQN algorithm is to approximate the Q-function using a neural network (NN) called *critic network* as the size of table storing Q-values for state action pairs gets large with large observation and action spaces rendering classical Q-learning infeasible. [MKS$^+$13] The Q-value as in Q-learning is however still considered for the training of the network to calculate the loss. The critic takes a state s as an input and calculates $|A|$ outputs approximating $q_*(s, a)$ for every $a \in A$. Furthermore, to break the correlation between following states, the trick of *replay memory* is utilized. At every time, a fixed number $N \in \mathbb{N}$ of the past experiences is stored, from which the training data is sampled. Each experience entry is a tuple $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ corresponding to a transition in time step t, where in state s_t action a_t was taken resulting in reward r_{t+1} transitioning to state s_{t+1}.

Since the critic network in DQN makes an output for every action available in the action space, it is not possible to utilize this method for continuous action spaces in a straight forward manner. One way to overcome this issue is to discretize real-valued action space dividing it into several intervals. This method does however have many limitation among which the curse of dimensionality, that is the exponential increase of the number of actions with the degrees of freedom in the environment and also information loss about the structure of the action space. [LHP$^+$15] One other way to apply DQN to continuous action spaces is to optimize for a at each time step, which is infeasible from a time complexity point of view.

3 Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) attempts to overcome the problems of DQN by applying its ideas to the deterministic policy gradient algorithm. In a way, it is safe to understand that DDPG to DPG is what DQN is to Q-Learning. In the first section of this chapter we will examine the class of deterministic policy gradient algorithms as presented by Silver et al. in [SLH$^+$14] as a basis to then examine the DDPG algorithm in the second section.

3.1 Deterministic Policy Gradient

Deterministic policy gradient (DPG) algorithms are a class of algorithms developed by Silver et al. to deal with continuous action spaces, which oppose stochastic policy gradient algorithms and are simpler and more efficient to calculate.

The goal in DPG is to maximize the total expected discounted reward of an actor following policy π. Suppose the states are sampled from a state distribution ρ. Let θ be a vector of parameters used by the policy. The function to maximize is then

$$J(\pi^{(\theta)}) = \int_S \rho^{\pi^{(\theta)}}(s) \cdot r(s, \pi^{(\theta)}(s))ds = \mathbb{E}_{s\sim\rho^\pi}[r(s, \pi^{(\theta)}(s))]. \tag{3.1}$$

To maximize this expected return, which corresponds to the Q-function as defined in Section 2.2 in DPG, we use the gradient of the Q-function in oppose to the maximizing iteration process we used in Q-learning to derive our policy. The reason for this is that the action space we are working with is a continuous one of which taking the maximum is infeasible. Hence, we move our policy in the direction of the Q-function's gradient. To do so, the following theorem is used.

Theorem 5. *The gradient of the Q-function as in Equation* (3.1) *is*

$$\nabla_\theta J \approx \mathbb{E}_{s\sim\rho^{\pi^{(\theta)}}}[\nabla_\theta \pi^{(\theta)}(s) \cdot \nabla_a q_\pi(s, a)] \tag{3.2}$$

where $a = \pi^{(\theta)}(s)$. This term is also the gradient of the policy's performance.

This theorem is proven in [SLH$^+$14].

3.2 Deep DPG

In DDPG, the ideas of DQN for function approximation are applied to the DPG algorithm. Here two neural network function approximators are used. One is called the *actor* which approximates an optimal policy $\pi^*(s)$. The other is called *critic* and approximates the expected return of the actor, when taking action a in state s, $q_\pi(s,a)$.

To update the critic network, the mean squared error of the guessed Q-values and the actual returns are utilized as in DQN. To update the actor network however, the gradient of the policy as in Equation (3.2) is used via gradient ascent. Complete loss equations for both the critic and the actor can be found in the Section 4.5 section. This gradient calculated approximately using the gradient of the mean of the Q-values of explored state action pairs. To break the correlation between following states in these two updates, the replay memory technique of DQN is utilized.

To incorporate an exploration-exploitation balance in the training process, each chosen action by the actor is variated by some small variance called noise, using a randomization process. This can be done, since the action spaces are continuous. The original implementation by [LHP+15] uses the Ornstein–Uhlenbeck process [UO30], but other noise variation processes have been shown be as useful.

Highlighting the structural differences of the neural networks used in DQN and DDPG, it is worth noticing that the critic does not need to compute a Q-value for all of the possible actions, but only the one action chosen by the actor at each time, so there is no longer a need for discretization of action spaces. The reason we can do this now, is that the actor network can also be trained using the policy's gradient as in Equation (3.2). Furthermore, a pseudo-code of the learning algorithm is provided in Section 4.5.

In the next chapter we are going to walk through our implementation of DDPG in detail and explain our project structure.

4 Implementation

In this chapter we explain our project structure in detail and explain how it implements DDPG. For the ease of use we oriented our implementation for the training of environments that implement a Gymnasium environment interface. Our implementation was fully written using Python in the PyCharm IDE.

4.1 Project Structure

Our project is organized into a python package called DDPG, which can be used as a standalone implementation of the DDPG algorithm to train Gymnasium environments. The DDPG package Consists of the following:

- A package called "Agent", that can be used as a standalone implementation of a DPPG agent consisting of an actor and a critic.
- A "DDPG Trainer", that implements the DDPG training loop for a given agent and environment.
- A "DDPG Evaluator", that evaluates a trained DDPG agent.
- A "Plotter", that can be used to collect and plot the reward progress in the DDPG Trainer and DDPG Evaluator.
- A "Replay Buffer", which implements the buffer memory technique of DDPG.

Each of these parts are explained in further detail below.

We decided to utilize structures and parameters described in the article that brings the concept of the DDPG to the RL scene [LHP+15]. Following the original implementation, both actor and critic nets are using rectified linear unit (ReLU), as an activation function for the two hidden layers, that have 400 and 300 nodes accordingly. Furthermore, excluding the final layers, parameters of residual layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the number of inputs of the given layer. Lastly, the target networks, which are copies of the original networks, are updated using the $\tau = 0.001$ value. The process of soft updates allows creating an illusion of a “steady” goal, enabling better convergence.

Lastly, structure of the network’s is based on nn.Sequential model from PyTorch library.

4.2 Agent

The Agent package is a sub-package of DDPG that can be used as a standalone package to represent a DDPG agent and consists of the following modules.

Actor

This network architecture, also considered as policy network, is supposed to determine the optimal policy. It computes the action a, being given some state s. The result is then evaluated by the critic network, and using the critic output, actor network is updated by performing gradient ascent, as the actor network aims to maximize the output of the critic network.

Concerning the actual implementation, our project again reflected recommendation from the original paper [LHP+15]. Therefore, the value of 10^{-4} was used as a learning rate for the actor network, final layer was chosen to be tanh, as it maps the output action to a $(-1; 1)$ range, allowing for more stability. Lastly, the initialization of the final layer parameters was taken from a uniform distribution [-3 $\times$ 10^{-3}, 3 $\times$ 10^{-3}].

To create and instance of the Actor class, the shape of the observation space and the action space are needed. Furthermore, the structure of the network can be manipulated via the following parameters.

- fc1_units: The number of nodes in the first hidden layer
- fc2_units: The number of nodes in the second hidden layer
- device: The device on which the calculation will be done.

Critic

The role of the critic network is to evaluate the actions of the actor network. Using predicted action, the critic outputs Q-value, which represents the expected return of provided state-action pair. The critic network is updated using temporal difference (TD) error. This error represents the difference between the predicted Q-value and Q-value resulting from the next state s_{t+1} and the action chosen by the actor network.

Following the original implementation, 10^{-3} was used for learning rate of the critic network and a weight decay of 10^{-2} was added. Similarly, our discount factor γ is equal to 0.99, and again the parameters initialization of the final layer is following the uniform distribution [-3 $\times$ 10^{-4}, 3 $\times$ 10^{-4}]. Finally, to complete the model, the actions were included into the critic network after the first hidden layer.

Instantiating the Critic class is identical to that of the Actor.

Noise

Noise is sub-package of the Agent package that defines an interface for noise classes and also implements this interface in two ways.

- Random Noise: Generates an random normal distribution noise in a given format from a sample.
- OU Noise: Implements the Ornstein–Uhlenbeck process as explained in Appendix .1

Agent

The Agent class is the class that represents an actual DDPG agent consisting of an actor and a critic and is capable of updating the weights of its networks given a sample batch. To create an instance of this class, following arguments are needed:

- Actor: An instance of the Actor class.
- Critic: An instance of the Critic class.
- Actor Optimized: A PyTorch Optimizer instance to optimize the actor
- Critic Optimizer: A PyTorch Optimizer instance to optimize the critic
- Noise: An instance of one of the implementations of the Noise interface
- discount: The discount factor for future predicted rewards
- tau: The learning factor for the networks.

4.3 Replay Buffer

The Replay Buffer, also known as "Experience Replay" in the theoretical description, was implemented with the size of 10^6. Main purpose for application of this element is to break temporal correlation between successive experiences gathered during interactions with en environment. During training, our implementation selects from the Replay Buffer mini-batches of size 64, which allow for a varied set of experiments.

The instantiation of the Replay Buffer class needs the capacity of the buffer and the batch size of each sampled batch as arguments.

4.4 Plotter

The Plotter class is a class that collects and plots reward histories through the episodes of a training or evaluation processes. An instance of this class, created with the number of episodes to plot as an argument, can be passed to a trainer instance as in Section 4.5 or an evaluator instance as in Section 4.6 to collect data.

4.5 DDPG Trainer

After initializing the agent with its respective actor, critic, optimizers and noise objects, the DDPG Trainer class can be instantiated. The train method is then used to train the given agent on the given environment for a number of E episodes. It is optional to provide a plotter and an evaluator when instantiating this class. If the plotter is provided, the reward data will be collected by that plotter and if the evaluator is provided, an evaluation round of 10 episodes will be done after every 10 episodes of training. Furthermore, the device argument of this class may be changed, if the training should be done on an acceleration device (e.g. 'GPU). By default, the training is done using the CPU.

The following pseudocode notation is based on the original paper [LHP+15] and presents the workflow of our implementation.

Algorithm 1: The DDPG Learning Loop Pseudocode

```
1  for episode = 1 to $E$ do
2      Receive initial observation state $s_1$
3      for time = 1 to T do
4          Take an action $a_t$ according to the Actor NN though the agent
5          Store transition tuple $(s_t, a_t, s_{t+1}, r_t, d_t)$ in Replay Buffer
6          Sample a random mini-batch $B$ of 32 transitions from Replay Buffer
7          Update networks as follows:
8          Compute target network $y_i$ for the critic:
               $$y = r + \gamma(1-d)Q_{target}(s', \mu_{target}(s'))$$
9          Minimize the loss of a critic NN:
               $$L_Q = \frac{1}{B}\sum_{(s,a,r,s',d)\in B}(y - Q(s,a))^2$$
10         Maximize the loss of an actor NN:
               $$L_\mu = \frac{1}{B}\sum_{(s,a,r,s',d)\in B}Q(s,\mu(s))$$
           Update the target networks:
               $$\theta_{Q_{target}} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q_{target}}$$
               $$\theta_{\mu_{target}} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu_{target}}$$
11     end for
12 end for
```

4.6 DDPG Evaluator

The DDPG Evaluator class can also be instantiated similarly to a DDPG Trainer and can be used to evaluate the performance of an agent. If a Plotter object is provided, the rewards of the evaluation episodes will be collected by that plotter. The device argument of this class can also be set similarly to the DDPG Trainer.

5 Results

Provided implementation of DDPG was tested on four environments with different profiles. Two of which belong to the "classic control environments", and another two to the "Multi-Joint dynamics with contact environments", also known as "MuJoCo". Further information on the environments characteristics can be found on the Gymnasium documentation website. For these results the noise that we used was sampled from normal distribution. However, for the classical control environments we decided to also test how the random noise would compete with the OU-Noise, in terms of the learning speed.

Using the plotter class, described in Chapter 4, the reward history gathered during training and evaluation steps was summed and displayed onto a plot describing reward return value, for the each episode. We run the whole process three times and, with help of Seaborn package [Was21], we plotted the mean of rewards per episode with 95% confidence interval.

The main areas of focus for interpreting the results are convergence, presence of upwards curve, and variability of reward value between episodes. Furthermore, it is important to point out that we did not scale the rewards. That means that the lower magnitude of the positive reward value can not indicate poor learning quality, without further context. Some of the environments are operating on small rewards, leading to overall shift of the Y-axis.

5.1 Pendulum

In the results plot Figure 5.1 one notices the upward curve in the reward value and lack of convergence. Variability of the reward value stays high throughout the whole learning process. This can be justified in the early stages by higher exploration of the environment and possible strategies. In later episodes, the variability is occurring due to the agent starting at a random point in each episode, and receiving a negative reward for every state that is not the goal state. Finally, around the 100th episode one can notice a decline in exploration, which suggest that an agent found a working policy, and therefore learned sufficiently.

Comparing the Figure 5.2 to Figure 5.1 one notices how the usage of noise sampled from uniform distribution impacts the learning process positively. Not only it escalated quicker, but also did not unlearn the correct actions it already took. Here, the environment is rather straightforward, which means that smaller exploitation is sufficient for the agent to learn quickly. However, in case of Pendulum that might not be useful, it might be probable, that with greater amount of episodes, the implementation with OU-noise would reach the results of the one with random noise.

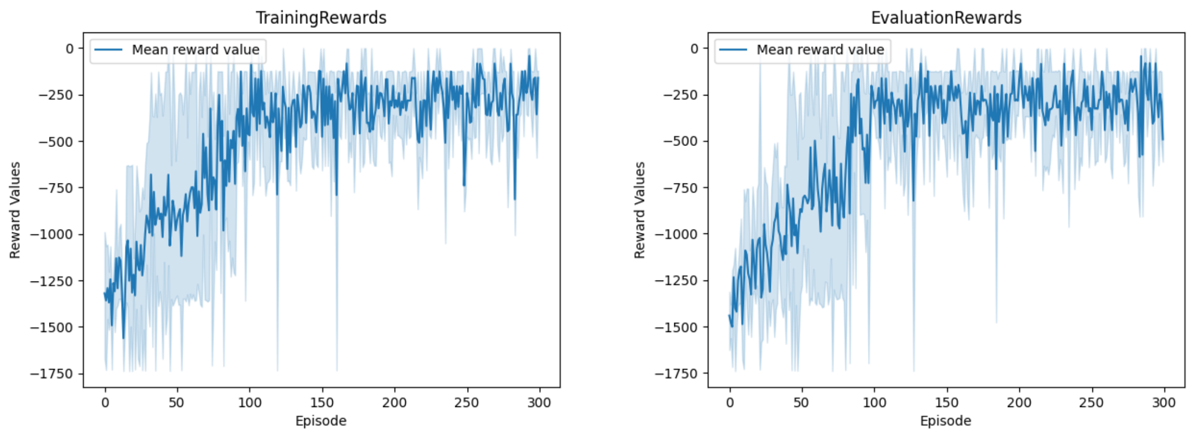

Fig. 5.1: Training(left) and evaluation(right) reward for the Pendulum-v1 environment, using random noise.

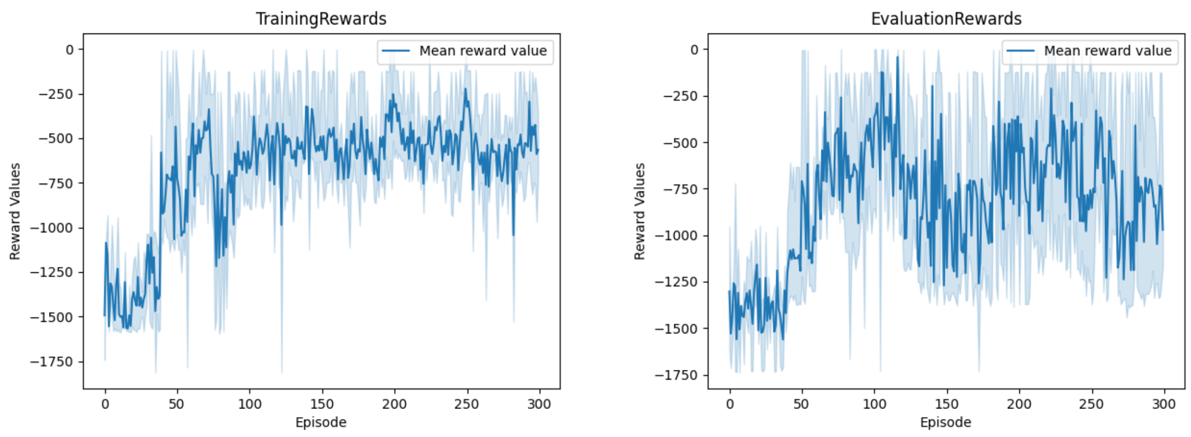

Fig. 5.2: Training(left) and evaluation(right) reward for the Pendulum-v1 environment, using Ornstein-Uhlenbeck noise.

5.2 MountainCarContinuous

Analyzing the Figure 5.3, we can see that the 60th episode mark is about the point where the agent learns the goal and becomes able of exploiting reward provided for solving the environment. The variability is significantly less prominent in comparison to the Pendulum environment. There are a few spikes in reward value in the beginning of the training process, but after about 60th episodes mark, the agent converges at its solution.

Examining the changes done by applying the Ornstein-Uhlenbeck noise, one notices that the agent reaches far more negative reward, which indicates higher exploitation. Another point worth of notice is that the converge value, for the agent using OU-noise, sits at a higher point. It is not unlikely that using greater value of noise variance in the random noise implementation, would yield similar results.

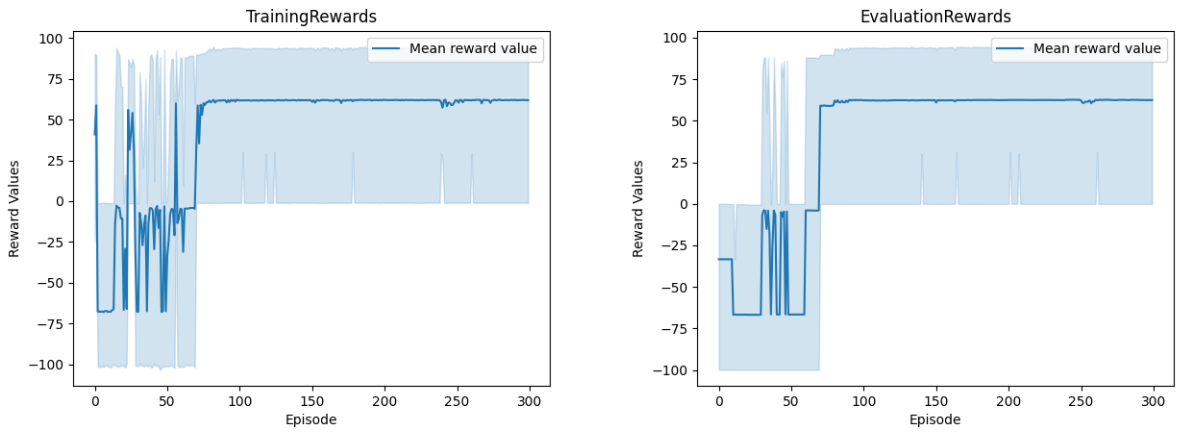

Fig. 5.3: Training(left) and evaluation(right) reward for the MountainCarContinuous-v0 environment, using random noise.

5.3 HalfCheetah

Moving onto MuJoCo environments, results of training an agent for solving the HalfCheetah environment, Figure 5.5, again show an upward trend in reward value, as well as high variability between episodes. At the 100th mark, the agent seem to have found a solution that works for it and does not improve much past that point.

5.4 Pusher

Agent for the Pusher environment, Figure 5.6, was trained on 800 episodes, which makes the convergence more likely. Yet again we observe upward trend of the reward value, and a somewhat stable behaviour after the 250th episode.

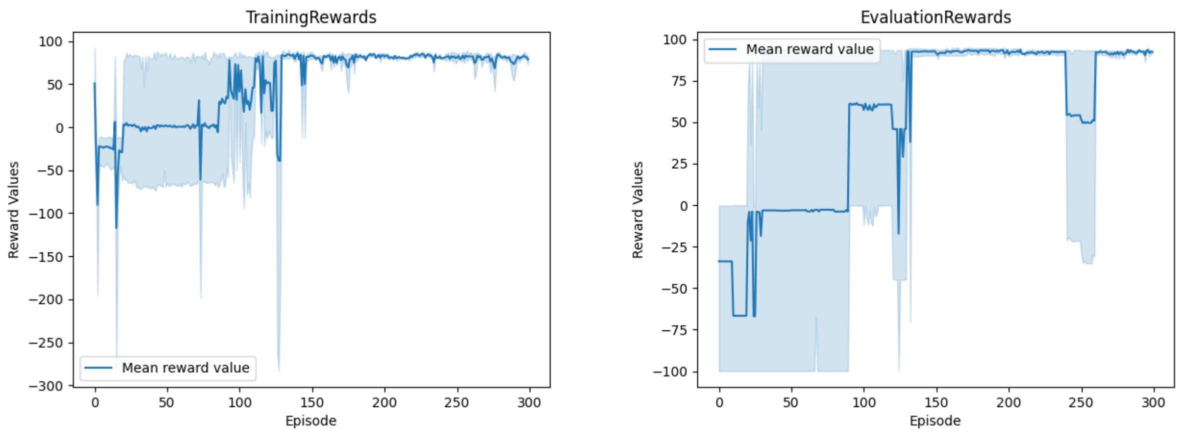

Fig. 5.4: Training(left) and evaluation(right) reward for the MountainCarContinuous-v0 environment, using Ornstein-Uhlenbeck noise.

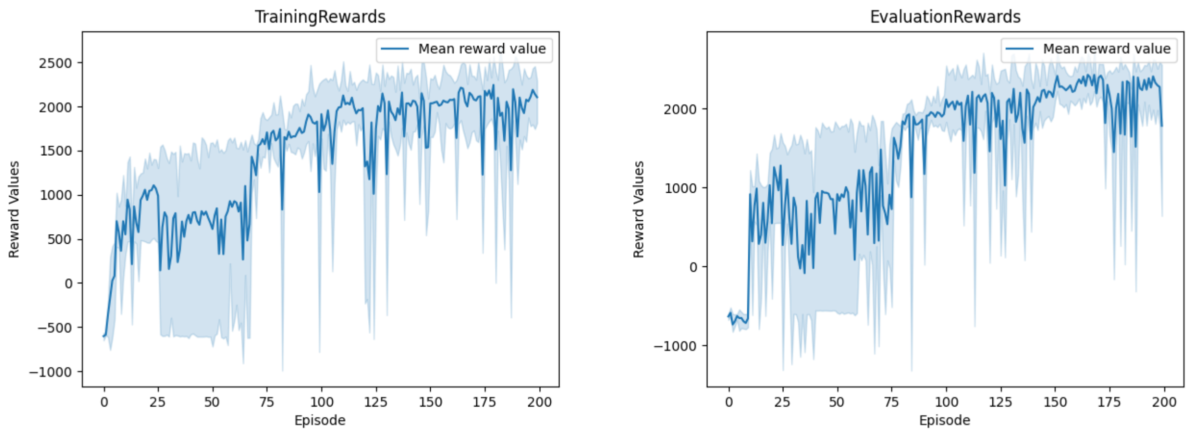

Fig. 5.5: Training(left) and evaluation(right) reward for the HalfCheetah-v4 environment, using random noise.

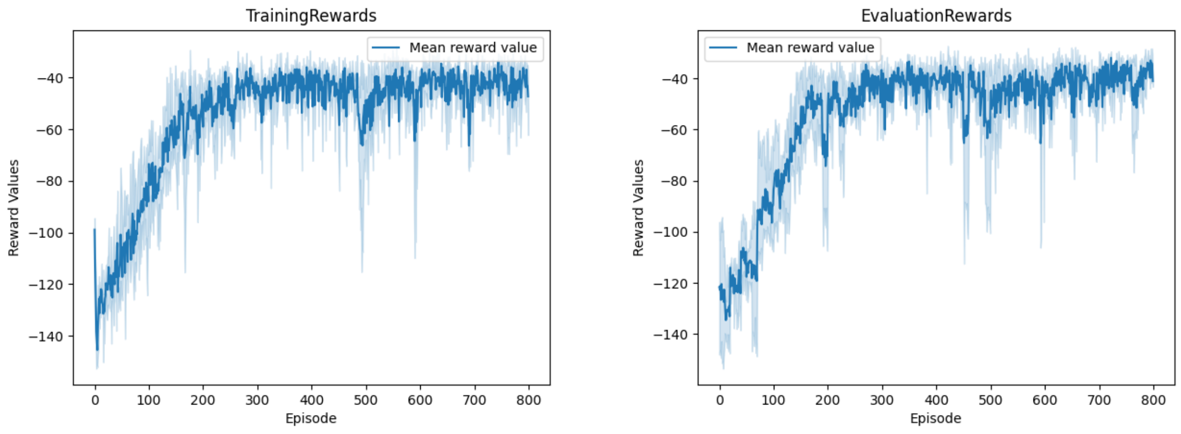

Fig. 5.6: Training(left) and evaluation(right) reward for the Pusher-v4 environment, using random noise.

Concluding this chapter, our agents have demonstrated learning capabilities. However, it would be worth exploring various noise configurations to optimize the learning process per environment. Different noise distributions, as well as shifts in their parameters, could provide enhanced performance of the DDPG algorithm.

Bibliography

[BBC57] R. Bellman, R.E. Bellman, and Rand Corporation: *Dynamic Programming.* Rand Corporation research study. Princeton University Press, 1957, ISBN 9780691146683.

[LHP$^+$15] 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 (Cornell University)*, September 2015. `http://export.arxiv.org/pdf/1509.02971`.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller: Playing atari with deep reinforcement learning, 2013.

[SB18] R.S. Sutton and A.G. Barto: *Reinforcement Learning, second edition: An Introduction.* Adaptive Computation and Machine Learning series. MIT Press, 2018, ISBN 9780262039246.

[SLH$^+$14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller: Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, June 2014.

[UO30] G. E. Uhlenbeck and L. S. Ornstein: On the theory of the brownian motion. *Physical Review*, 36(5):823–841, 1930, 10.1103/physrev.36.823.

[Was21] Michael L. Waskom: seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021, 10.21105/joss.03021. `https://doi.org/10.21105/joss.03021`.

.1 Ornstein–Uhlenbeck Noise

Our implementation realized the OU process equation as a class, with parameters θ, which indicates the rate at which the process reverts to the mean, σ, that describes the mean of the process, and μ set to the size of the action space, which controls the extent of random fluctuations parameter. Here once again, following the values used in the original paper we set $\theta = 0.15$, and $\sigma = 0.2$.

Following the original equation [UO30]:

$$dX_t = \theta(\mu - X_t)dt + \sigma dW_t$$

our implementation generates an array of the magnitude of an action space with randomly generated numbers, on the place of dW_t, which stands for a standard Brownian motion on $t \in [0, \infty)$, but apart from that stays true to the form.