# Deep Reinforcement Learning for Continuous Control of Multi-Agent Robotic Arms

Catarina Pires
*Deep Reinforcement Learning Nanodegree, Udacity*

***Abstract*—This project addresses the problem of continuous control in high-dimensional action spaces using deep reinforcement learning. The task consists of training twenty identical double-jointed robotic arm agents, each with its own copy of the environment, such that they maintain their end-effectors at a target location. for this purpose, the Deep Deterministic Policy Gradient (DDPG) algorithm [1], an actor–critic method suitable for continuous action domains, was implemented combined with Prioritized Experience Replay (PER) [2] to improve sample efficiency. In this environment, a reward of +0.1 is provided at each step that an agent's end-effector remains in the target zone, and the task is considered solved once the agents achieve an average score of +30 over 100 consecutive episodes, averaged across all agents. Using this approach, the environment was successfully solved in 164 episodes.**

## I. Introduction

Reinforcement learning has emerged as a powerful framework for training agents to solve complex decision-making problems by interacting with an environment. Unlike supervised learning, where labels are provided, reinforcement learning relies on trial and error, with agents improving their behaviour by maximizing cumulative rewards. In recent years, the combination of deep neural networks with reinforcement learning has enabled agents to tackle high-dimensional, continuous State and action spaces that were previously intractable. These advances have shown promising applications in robotics, autonomous vehicles, and continuous control tasks, where precise actions must be chosen from an infinite range of possibilities.

In this project, deep reinforcement learning is applied to the Continuous Control environment from the Unity ML-Agents toolkit. The objective is to train twenty identical double-jointed robotic arms, each interacting with its own copy of the environment, to keep their end-effectors within a target zone. This setup poses a significant challenge due to the high dimensionality of the action space and the need for coordinated, stable control policies. To address this, the Deep Deterministic Policy Gradient (DDPG) algorithm [1], an actor–critic method well-suited for continuous action domains, is implemented and combined it with Prioritized Experience Replay (PER) [2] to improve sample efficiency. The report details the methodology, experimental setup, and results achieved in solving the environment.

## II. Methods

Unlike the previous Navigation project, this environment presents a continuous, high-dimensional action space that cannot be efficiently handled by value-based methods such as Deep Q-Networks (DQN) [3]. Discretizing the action space would lead to an exponential growth in possible actions, rendering such approaches impractical. Consequently, this problem is more naturally addressed with policy-based methods, which can directly parametrise and optimize actions in continuous domains.

As such, the algorithm chosen to be implemented was the Deep Deterministic Policy Gradient (DDPG) algorithm [1] combined it with Prioritized Experience Replay (PER) [2] to improve sample efficiency.

### A. Deep Deterministic Policy Gradient

DDPG is a model-free, off-policy reinforcement learning algorithm designed for continuous action spaces. It combines ideas from Deep Q-Networks (DQN) and policy gradient methods to learn both a policy and a value function simultaneously.

**Key Components:** DDPG uses an actor-critic architecture with four neural networks:

- Actor network $\mu(s|\theta^\mu)$: Maps States to actions deterministically.
- Critic network $Q(s, a|\theta^Q)$: Estimates the Q-value for State-action pairs.
- Target actor network $\mu'(s|\theta^{\mu'})$: Slowly tracking copy of the actor.
- Target critic network $Q'(s, a|\theta^{Q'})$: Slowly tracking copy of the critic.

**Core Mechanisms:** The algorithm addresses the challenge of continuous action spaces by learning a deterministic policy rather than a stochastic one. It uses experience replay to break correlations in the data and target networks to stabilize training. The actor is trained using the deterministic policy gradient theorem, while the critic is trained using temporal difference learning similar to DQN.

**Training Process:** The actor learns to maximize the Q-values predicted by the critic, while the critic learns to accurately estimate Q-values using the Bellman equation. Target networks are updated slowly using soft updates ($\tau << 1$) rather than hard copies, providing more stable learning targets.

The algorithm 1 shows how DDPG alternates between collecting experience and learning from it. The key insight is using the deterministic policy gradient theorem to update the actor network while using standard temporal difference learning for the critic. The soft target updates with $\tau$ much

less than 1 ensure stable learning by preventing the target networks from changing too rapidly.

**Algorithm 1** Deep Deterministic Policy Gradient (DDPG)

Initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with random weights $\theta^Q$ and $\theta^\mu$
Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $\mathcal{R}$
Initialize random process $\mathcal{N}$ for action exploration
**for** episode $= 1, M$ **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation State $s_1$
    **for** $t = 1, T$ **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
        Execute action $a_t$ and observe reward $r_t$ and new State $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{R}$
        **if** $|\mathcal{R}| >$ batch_size **then**
            Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{R}$
            Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
            Update 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:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$
            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 if**
    **end for**
**end for**

### *B. Prioritized Experience Replay*

Prioritized Experience Replay is an enhancement to the standard experience replay mechanism used in deep reinforcement learning. Instead of sampling transitions uniformly from the replay buffer, PER samples transitions based on their temporal difference (TD) error, giving priority to experiences that are more "surprising" or informative.

**Key Motivations:** Standard uniform sampling from experience replay treats all transitions equally, but some experiences are more valuable for learning than others. Transitions with high TD errors indicate where the agent's current understanding is most wrong, making them particularly valuable for learning.

**Core Components:** PER uses a priority value $p_i$ for each transition $i$, typically based on the TD error:

- *TD Error Priority*: $p_i = |\delta_i| + \epsilon$, where $\delta_i$ is the TD error and $\epsilon$ prevents zero priorities.
- *Rank-based Priority*: Transitions ranked by TD error magnitude.
- *Proportional Priority*: Direct use of TD error magnitude.

**Sampling Mechanism:** Transitions are sampled with probability $P(i) = p_i^\alpha / \Sigma_k p_k^\alpha$, where $\alpha$ controls how much prioritization is used ($\alpha = 0$ gives uniform sampling, $\alpha = 1$ gives full prioritization).

**Importance Sampling Correction:** Since PER changes the sampling distribution, it introduces bias that must be corrected using importance sampling weights:

$$w_i = (1/N \times 1/P(i))^\beta \tag{1}$$

where $\beta$ controls how much the bias correction is applied (typically annealed from a lower value to 1).

The algorithm 2 describes the DDPG algorithm combined with the use of PER.

The key differences from standard DDPG are:

- *Priority-based sampling*: Instead of uniform sampling, transitions are selected based on their TD error magnitude.
- *Importance sampling correction*: The loss functions are weighted by $w_i$ to correct for the sampling bias.
- *Priority updates*: After computing new TD errors, the priorities in the replay buffer are updated.
- *Importance sampling correction*: The loss functions are weighted by $w_i$ to correct for the sampling bias.
- *Beta annealing*: The importance sampling correction $\beta$ is gradually increased to 1.0 during training.

This approach typically leads to faster convergence and better sample efficiency, as the agent focuses on learning from the most informative experiences rather than treating all transitions equally.

## III. Environment

The Unity Machine Learning Agents (ML-Agents) toolkit [4] is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. For the purpose of this project, one of the Unity's rich environments was used to design, train, and evaluate the chosen deep reinforcement learning algorithms. Note that this environment is similar but not identical to the Reacher environment present in [4],

In this environment 1, a reward of +0.1 is provided of each step that a double-jointed arm is in a goal location. Thus, the objective of the agent is to maintain its position at the target location for as many time steps as possible. The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

**Algorithm 2** Deep Deterministic Policy Gradient with Prioritized Experience Replay

Initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with random weights
Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize prioritized replay buffer $\mathcal{D}$ with capacity $N$
Initialize priority parameters: $\alpha$, $\beta_0$, $\beta$ schedule
Initialize random process $\mathcal{N}$ for action exploration
**for** episode $= 1, M$ **do**
    Receive initial observation state $s_1$
    **for** $t = 1, T$ **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        Compute initial TD error: $\delta = |r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}) - Q(s_t, a_t|\theta^Q)|$
        Set priority $p_t = (\delta + \epsilon)^\alpha$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$ with priority $p_t$
        **if** $|\mathcal{D}| >$ batch_size **then**
            Sample minibatch of $K$ transitions based on priorities:
            $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$
            Compute importance sampling weights:
            $w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$
            Normalize weights: $w_i \leftarrow \frac{w_i}{\max_j w_j}$
            Compute target values:
            $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
            Compute TD errors:
            $\delta_i = y_i - Q(s_i, a_i|\theta^Q)$
            Update critic with importance sampling weights:
            $L = \frac{1}{K} \sum_i w_i (\delta_i)^2$
            $\theta^Q \leftarrow \theta^Q - \lambda_Q \nabla_{\theta^Q} L$
            Update actor policy:
            $\nabla_{\theta^\mu} J \approx \frac{1}{K} \sum_i w_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}$
            $\theta^\mu \leftarrow \theta^\mu + \lambda_\mu \nabla_{\theta^\mu} J$
            Update priorities in replay buffer:
            $p_i \leftarrow (|\delta_i| + \epsilon)^\alpha$ for all sampled transitions
            Update target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$
            Anneal $\beta$: $\beta \leftarrow \min(1.0, \beta_0 + (1.0 - \beta_0) \cdot \frac{\text{step}}{\text{total\_steps}})$
        **end if**
    **end for**
**end for**

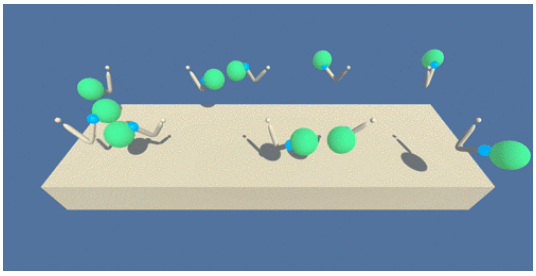

Figure 1. Environment

## IV. Implementation Details

The agent was implemented using the Deep Deterministic Policy Gradient (DDPG) algorithm, extended with Prioritized Experience Replay (PER). The agent architecture consists of an actor and a critic, each with a corresponding target network. The actor network maps observed States to continuous actions, while the critic network evaluates State–action pairs through Q-values. Both networks are optimized using the Adam optimizer with distinct learning rates: $5x10^{-5}$ for the actor and $4x10^{-4}$ for the critic. Target networks are updated using soft updates with a factor $\tau = 10^{-3}$. To encourage exploration, an Ornstein–Uhlenbeck process is added to the actor's actions, with parameters $\theta = 0.15$ and $\sigma = 0.2$. Noise is gradually reduced over time by decaying an $\epsilon$-scaling factor.

The replay buffer is implemented with prioritized sampling, where the probability of drawing a transition is proportional to its temporal-difference (TD) error. This allows the critic to focus on correcting the largest errors while importance-sampling weights compensate for bias in the learning updates. At each step, experiences are stored in the buffer, and the agent begins learning once the buffer holds at least one batch of experiences. The critic loss is computed as a weighted mean squared TD error using PER weights, while the actor is updated using the deterministic policy gradient. Gradients for the critic are clipped to a maximum norm of 1.0 to improve stability. Training proceeds in episodes, with actions sampled from the actor policy plus noise, and multiple learning passes (10) performed every 20 steps.

The agent was implemented in PyTorch using the DDPG algorithm with Prioritized Experience Replay. The main hyperparameters and training settings are summarized in Table I.

As for the Prioritized experience replay, the parameters used can be found in Table II.

## V. Results

The following results, Figure 2, demonstrate the successful application of DDPG with Prioritized Experience Replay to the multi-agent continuous control task. The algorithm achieved the target performance criterion of an average

| Parameter | Value | Description |
|---|---|---|
| Replay buffer size | $1 \times 10^6$ | Maximum number of stored experiences |
| Batch size | 128 | Number of experiences sampled per update |
| Discount factor ($\gamma$) | 0.99 | Future reward discount |
| Soft update factor ($\tau$) | $1 \times 10^{-3}$ | Rate for updating target networks |
| Actor learning rate | $5 \times 10^{-5}$ | Adam optimizer learning rate for actor |
| Critic learning rate | $1 \times 10^{-4}$ | Adam optimizer learning rate for critic |
| Update frequency | 20 steps | Number of environment steps between updates |
| Learning passes per update | 10 | Number of gradient updates performed at each learning step |
| Gradient clipping | 1.0 | Maximum gradient norm for critic updates |
| OU noise ($\theta$) | 0.15 | Mean reversion rate for Ornstein–Uhlenbeck noise |
| OU noise ($\sigma$) | 0.2 | Volatility parameter for Ornstein–Uhlenbeck noise |
| OU noise ($\epsilon$) | 1.0 | explore->exploit noise process added to act step |
| OU noise ($\epsilon$ decay) | $1 \times 10^{-6}$ | decay rate for noise process |

Table I. Hyperparameters and training settings used for the DDPG agent with Prioritized Experience Replay.

| Parameter | Value | Description |
|---|---|---|
| $\alpha$ (priority exponent) | 0.6 | Controls how strongly TD-error affects sampling probability |
| $\beta_i$ (initial bias correction) | 0.4 | Initial importance-sampling correction exponent |
| $\beta_f$ (final bias correction) | 1.0 | Final importance-sampling correction exponent |
| $\beta$ update steps | 1000 | is annealed from 0.4 $\rightarrow$ 1.0 over training |
| $\epsilon$ (priority offset) | 110−6 | Small constant added to TD-error to avoid zero probability |

Table II. Parameters used for Prioritized Experience Replay (PER).

score of +30 over 100 consecutive episodes in 164 training episodes.

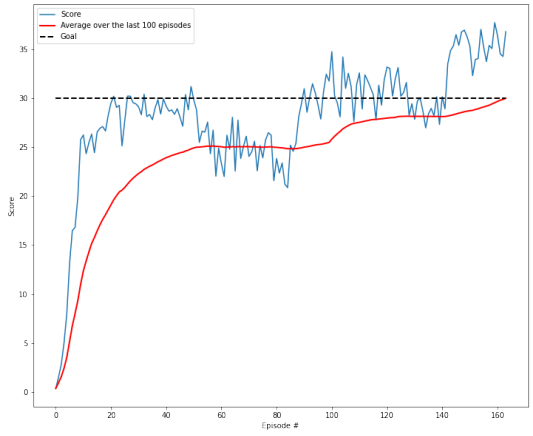


Figure 2. Results

## VI. Conclusion and Future Work

This project successfully applied Deep Deterministic Policy Gradient (DDPG) with Prioritized Experience Replay (PER) to solve a multi-agent continuous control task. The approach achieved the target performance of +30 average score over 100 episodes across twenty robotic arm agents in 164 training episodes. The integration of PER with DDPG enabled sample-efficient learning by prioritizing high temporal difference error experiences, while the actor-critic architecture effectively learned precise control policies for maintaining end-effectors within target zones. The results demonstrate the effectiveness of combining prioritized experience replay with policy gradient methods for complex continuous control problems.

However, several alternative algorithms could potentially improve upon the current implementation:

- **Proximal Policy Optimization (PPO)** [5]: PPO's clipped surrogate objectives provide more stable policy updates and could lead to more consistent learning across all agents. Its simpler implementation compared to DDPG might facilitate easier hyperparameter tuning and more robust training dynamics.
- **Asynchronous Advantage Actor-Critic (A3C)** [6]: A3C's asynchronous training could leverage the multi-agent nature of this environment for faster convergence through parallel experience collection. The shared global network approach could improve knowledge transfer between agents while enhancing exploration diversity.
- **Distributed Distributional DDPG (D4PG)** [7]: D4PG extends DDPG with distributional value learning and distributed training, potentially providing richer value representations and better uncertainty modeling. Its combination of prioritized replay, multi-step returns, and distributional learning could achieve superior sample efficiency compared to the current implementation.

## References


[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[4] "GitHub - Unity-Technologies/ml-agents: The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents using deep reinforcement learning and imitation learning. — github.com," https://github.com/Unity-Technologies/ml-agents, [Accessed 30-05-2024].

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PmLR, 2016, pp. 1928–1937.

[7] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients," *arXiv preprint arXiv:1804.08617*, 2018.