PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

# Métodos Aproximados

IIC3675 - Aprendizaje Reforzado

**Group 40**

**Group members:**
William Aarland, Pascal Lopez Wilkendorf

**Delivery date:** 1st June 2025

## 1 a) Solving `MountainCar` Domain Using Sarsa and Q-Learning with Linear Function Approximation

In this task, we solve the `MountainCar` domain using both the Sarsa and Q-Learning algorithms, applying linear function approximation. Each algorithm was run for 1000 episodes and repeated 30 times. We averaged the episode rewards across runs and plotted the average reward per episode. The hyperparameters used were $\gamma = 1$, $\epsilon = 0$, and $\alpha = 0.5/8$.

As shown in Figure 1, there is no clearly superior algorithm for this domain. However, Sarsa appears to slightly outperform Q-Learning in terms of average episode reward, particularly in the later episodes. This suggests that Sarsa may have a marginal advantage in stability or convergence under the chosen settings. The performance of both methods is relatively similar, which is expected, as the main difference is that Sarsa is on-policy method whereas Q-Learning is off-policy.

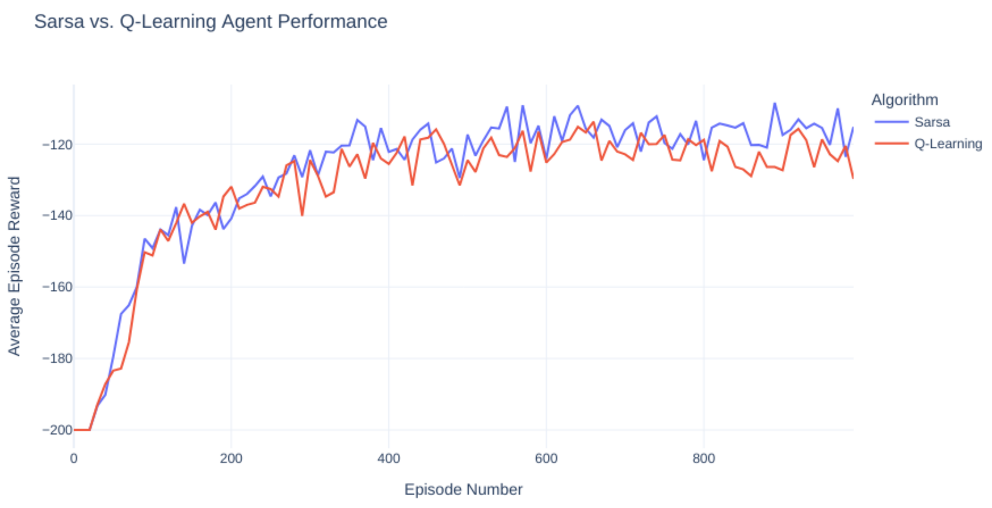

Figure 1: Comparison of average episode reward between Sarsa and Q-Learning on the `MountainCar` domain using linear function approximation. Each point represents the mean of 30 runs.

**Note** In `MountainCar-v0`, the episode reward is simply the negative of the episode length, so plotting average episode reward is equivalent to plotting average episode length as requested.

## 2 b) Solving `MountainCar-v0` using DQN

In this task, we solve the `MountainCar-v0` domain using Deep Q-Networks (DQN), with the raw observations from the environment (without tile coding), as specified in the assignment. We used the `Stable_Baselines3` Python RL framework [1], which provides a simple and modular implementation of DQN.

**Hyperparameter search.** We conducted a hyperparameter search by sampling different configurations of the most common hyperparameters. Based on the results from these sampled we narrowed down the hyperparameter search by fixing $\gamma = 0.99$, and **Net_Arch** $= [128, 128]$, and trying out different combinations of learning rate, and exploration rate. The sampled combinations and their results are shown in Table 1. The columns are learning rate $\alpha$, exploration fraction $\epsilon$, discount factor $\gamma$, network architecture (where the list length is the number of layers, and each element is the number of neurons in the corresponding layer), mean reward over all episodes, and mean reward over the last 100 episodes.

| **Learning** | **Exploration** | **Gamma** | **Net_Arch** | **Mean Reward** | **Last 100 Mean** |
|---|---|---|---|---|---|
| 0.001 | 0.1 | 0.99 | [128, 128] | -188.52 | -126.70 |
| 0.0001 | 0.1 | 0.99 | [128, 128] | -183.64 | -143.79 |
| 0.0005 | 0.1 | 0.99 | [64, 64] | -180.05 | -158.24 |
| 0.001 | 0.2 | 1.0 | [64, 64] | -176.00 | -166.49 |
| 0.0005 | 0.1 | 0.99 | [128, 128] | -161.76 | -177.42 |
| 0.0001 | 0.1 | 1.0 | [64, 64] | -198.95 | -198.75 |
| 0.0001 | 0.3 | 0.99 | [128, 128] | -199.81 | -199.73 |
| 0.001 | 0.1 | 0.9 | [64, 64] | -200.00 | -200.00 |
| 0.0005 | 0.2 | 0.9 | [64, 64] | -200.00 | -200.00 |
| 0.0001 | 0.1 | 0.8 | [128, 128] | -199.97 | -200.00 |
| 0.001 | 0.2 | 0.99 | [128, 128] | -200.00 | -200.00 |
| 0.0005 | 0.3 | 0.9 | [64, 64] | -200.00 | -200.00 |
| 0.0001 | 0.2 | 0.99 | [128, 128] | -199.69 | -200.00 |
| 0.001 | 0.1 | 0.8 | [64, 64] | -200.00 | -200.00 |
| 0.001 | 0.2 | 0.8 | [64, 64] | -200.00 | -200.00 |
| 0.001 | 0.1 | 0.9 | [128, 128] | -199.95 | -200.00 |
| 0.001 | 0.3 | 0.99 | [128, 128] | -200.00 | -200.00 |
| 0.0005 | 0.2 | 0.99 | [128, 128] | -199.87 | -200.00 |
| 0.0005 | 0.3 | 0.99 | [128, 128] | -200.00 | -200.00 |

Table 1: Hyperparameter trial results for DQN on `MountainCar-v0`. Sorted by Last 100 Mean.

**Final configuration and results.** Based on the hyperparameter search, we selected the following configuration as final:

- Learning rate $\alpha$: 0.001
- Exploration fraction $\epsilon$: 0.1
- Discount factor $\gamma$: 0.99
- Network architecture: [128, 128]

Using this configuration, we ran DQN 30 times for 1500 episodes. The average episode reward progression over episodes is shown in Figure 2. Note: some of the runs had more than 1500 episodes; this was later standardized for final evaluation.

**Discussion.**

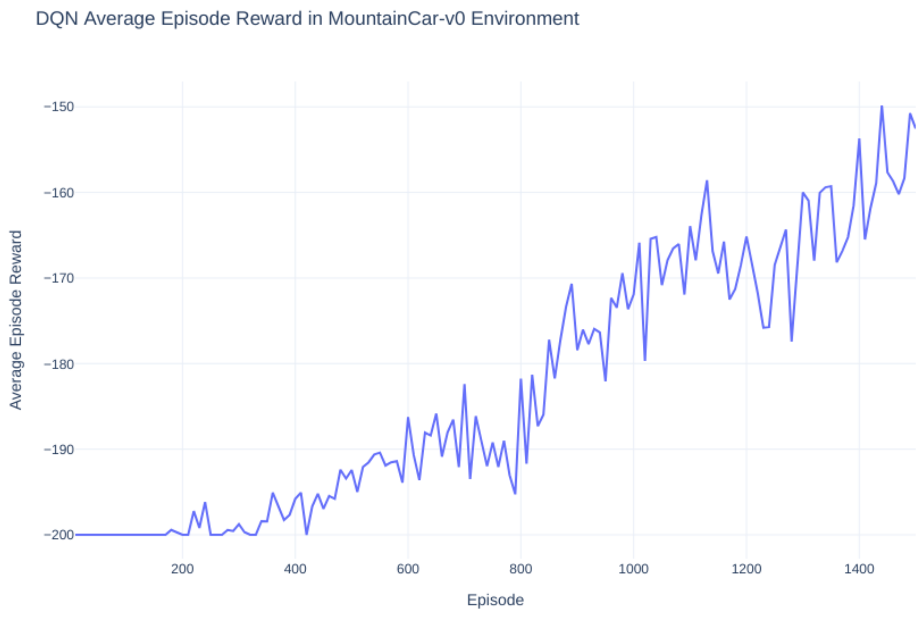

Figure 2: DQN average episode reward on `MountainCar-v0` over 1500 episodes.

- **Is MountainCar-v0 an easy or difficult domain for DQN? Why?** `MountainCar-v0` is a relatively difficult domain for DQN because it requires a degree of long-term planning: the agent must first go backwards to build momentum, which is not immediately rewarded. This makes the reward sparse and the value estimation challenging.

- **Difficulties in hyperparameter tuning and how we addressed them.** The main difficulty was choosing an appropriate learning rate and exploration rate that would allow learning to occur without getting stuck in suboptimal behavior. We addressed this by systematically sampling combinations and analyzing both the overall episode mean and the last 100-episode mean.

- **Most important hyperparameters and why.** The most important hyperparameters were the learning rate $\alpha$ and the network architecture. The learning rate controlled the stability and speed of learning, while a sufficiently expressive network architecture ([128, 128]) was required to approximate the Q-values effectively.

- **Final hyperparameter configuration.** As stated above: $\alpha = 0.001$, $\epsilon = 0.1$, $\gamma = 0.99$, architecture [128, 128].

- **Comparison with Sarsa and Q-Learning with linear approximation.** DQN performed worse than Sarsa and Q-Learning with linear approximation for this problem. Contrary to the theory, where Sarsa, and Q-learning yields a lower episode reward in comparison to DQN. The linear approximation should struggle to capture the non-linear value function required for this domain, where DQN encapsulated the non-linearity with a neural network. For further investigation, the DQN can be run for more episodes to see if it outperforms Sarsa, and Q-learning. With a quick glance at Table 1, some of the DQN settings yields approximately the same results as Sarsa and Q-learning when looking at the mean of the last 100 episodes. If DQN stagnates around the same rewards as Sarsa, and Q-learning remains to be seen, or if DQN yields better results with longer run-time.

**Future Improvements** When choosing the hyperparameter settings, some trials were run for more than 1500 episodes. Convergence, number of episodes, and stagnation should have been investigated more closely before choosing the final hyperparameter setting for DQN. This was unfortunately not done in this experiment.

## 3 c) Solving `MountainCarContinuous-v0` Using Actor-Critic with Linear Function Approximation

In this task, we solve the `MountainCarContinuous-v0` domain using an actor-critic algorithm with linear function approximation. The actor's policy is modeled as a Gaussian distribution, with both the mean and the standard deviation learned during training. We used tile coding to extract features from the observations, via the provided `FeatureExtractor` class.

The hyperparameters were set according to the assignment instructions: $\gamma = 1.0$, $\alpha_v = 0.001$ for the critic, and $\alpha_\pi = 0.0001$ for the actor. The algorithm was run for 30 runs, each consisting of 1000 episodes. We report the average episode length every 10 episodes.

The results are shown in Figure 3. The agent starts with maximum episode length (around 1000), and progressively learns to reach the goal faster. By the end of training, the average episode length has reduced to approximately 200 steps, showing that the learned policy is effective.

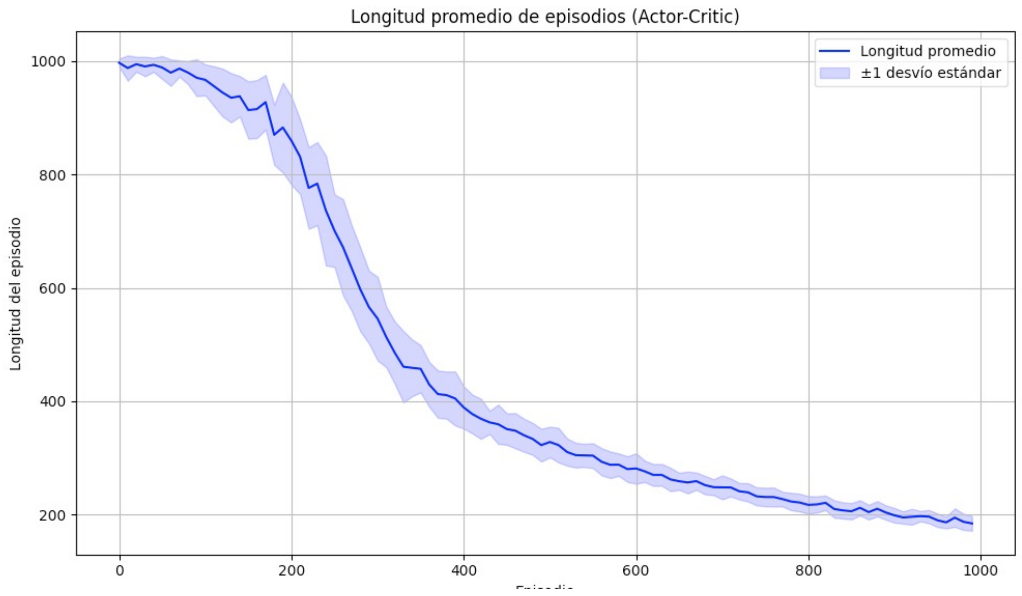

Figure 3: Average episode length of actor-critic with linear function approximation on `MountainCarContinuous-v0`. The shaded area indicates ±1 standard deviation over 30 runs.

## 4 d) Solving `MountainCarContinuous-v0` using DDPG

In this task, we solve the `MountainCarContinuous-v0` domain using Deep Deterministic Policy Gradient (DDPG), with the raw observations from the environment (without tile coding), as specified in the assignment. We used the `Stable_Baselines3` Python RL framework [1].

**Hyperparameter search.** We followed a similar approach as in the DQN experiment: we randomly sampled a set of hyperparameter configurations and evaluated them by running each configuration for 1000 episodes. The configuration that achieved the best performance was selected based on the mean reward of the last 100 episodes.

The sampled hyperparameter combinations and their results are shown in Table 2. The columns are learning rate, action noise standard deviation $\sigma$, noise mean, replay buffer size, batch size, target network update rate $\tau$, discount factor $\gamma$, mean reward over all episodes, and mean reward over the last 100 episodes.

| LR | $\sigma$ | Mean | BufSize | Batch | $\tau$ | $\gamma$ | MeanR | Last100R |
|---|---|---|---|---|---|---|---|---|
| 0.0001 | 0.1 | 0.0 | 1e6 | 32 | 0.01 | 1.00 | -2.30 | -1.00 |
| 0.001 | 0.1 | 0.0 | 1e6 | 64 | 0.001 | 0.95 | -2.88 | -1.04 |
| 0.001 | 0.1 | 0.2 | 1e4 | 32 | 0.001 | 0.95 | -12.38 | -1.31 |
| 0.001 | 0.2 | 0.0 | 1e6 | 64 | 0.001 | 0.99 | -7.39 | -4.02 |
| 0.0005 | 0.1 | 0.2 | 1e5 | 32 | 0.01 | 0.99 | -5.00 | -4.06 |
| 0.0001 | 0.2 | 0.0 | 1e6 | 64 | 0.001 | 0.99 | -4.20 | -4.19 |
| 0.0001 | 0.2 | 0.1 | 1e4 | 32 | 0.01 | 0.95 | -5.35 | -5.08 |
| 0.0005 | 0.1 | 0.2 | 1e4 | 128 | 0.01 | 0.95 | -8.01 | -5.30 |
| 0.001 | 0.1 | 0.2 | 1e4 | 32 | 0.001 | 1.00 | -19.26 | -6.50 |
| 0.001 | 0.2 | 0.2 | 1e5 | 128 | 0.01 | 1.00 | -12.90 | -7.49 |
| 0.001 | 0.2 | 0.2 | 1e6 | 256 | 0.005 | 1.00 | -9.72 | -7.74 |
| 0.0001 | 0.3 | 0.0 | 1e4 | 32 | 0.001 | 0.99 | -9.10 | -9.01 |
| 0.001 | 0.3 | 0.0 | 1e4 | 256 | 0.001 | 0.99 | -12.85 | -9.04 |
| 0.0001 | 0.3 | 0.0 | 1e6 | 128 | 0.001 | 0.95 | -9.13 | -9.08 |
| 0.0001 | 0.2 | 0.2 | 1e5 | 256 | 0.001 | 0.99 | -8.91 | -9.35 |
| 0.0005 | 0.3 | 0.1 | 1e5 | 128 | 0.005 | 0.95 | -10.53 | -9.83 |
| 0.001 | 0.3 | 0.2 | 1e6 | 128 | 0.001 | 0.99 | -13.69 | -12.47 |

Table 2: Hyperparameter trial results for DDPG on `MountainCarContinuous-v0`.

**Final configuration and results.** Based on the hyperparameter search, we selected the following configuration as final:

- Learning rate: 0.0001
- Noise standard deviation $\sigma$: 0.1
- Noise mean: 0.0
- Replay buffer size: $1 \times 10^6$
- Batch size: 32
- $\tau$: 0.01
- $\gamma$: 1.0

Using this configuration, we ran DDPG ten times for 1000 episodes. The evolution of the average episode reward is shown in Figure 4. As can be seen, the agent quickly learns to solve the task after approximately 300 episodes, after which it consistently achieves near-optimal performance.

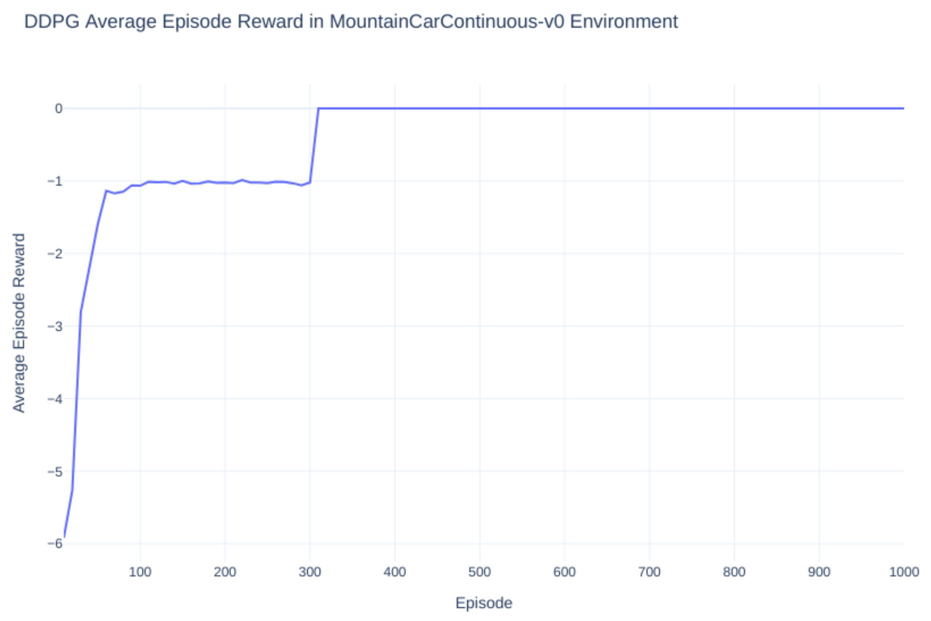

Figure 4: DDPG average episode reward on `MountainCarContinuous-v0` over 1000 episodes.

**Discussion.**

- **Is this problem easy or difficult for DDPG? Why?** `MountainCarContinuous-v0` is a relatively easy problem for DDPG once a good set of hyperparameters is found. The continuous action space aligns well with DDPG's architecture, and the reward structure provides useful feedback for learning the necessary policy.
- **Difficulties in hyperparameter tuning and how they were addressed.** The main difficulty was selecting the appropriate action noise parameters and learning rate to balance exploration and stable learning. We addressed this by exploring different configurations as shown in Table 2.
- **Most important hyperparameters and why.** The most important hyperparameters were the learning rate, noise and $\sigma$. A low learning rate helped stabilize training, while appropriate noise was critical for sufficient exploration in the early episodes.
- **Final hyperparameter configuration.** As listed above (learning rate 0.0001, noise $\sigma$ 0.1, $\tau$ 0.01, $\gamma$ 1.0, etc.).
- **Comparison to actor-critic with linear approximation.** We did not implement actor-critic with linear approximation (Task c). However, based on the results from DDPG, it is likely that DDPG would perform better, as it can represent more complex continuous policies and value functions compared to a linear model.

## References

[1] Antonin Raffin et al. ‘Stable-baselines3: Reliable reinforcement learning implementations’. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.