Deep Reinforcement Learning

Hassen Aissa (341649), Matya Aydin (388895), Yassine Turki (344704), Adam Mesbahi (387382), Aziz Sebbar (389027), Mehdi Zoghlami (326381)
team name: Random_hyperparameters_generator

Abstract

In this work, we present a comprehensive empirical study of four state-of-the-art deep reinforcement learning (RL) algorithms—Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), and Twin Delayed Deep Deterministic Policy Gradients (TD3); across five classical control benchmarks: CartPole-v1, MountainCar-v0, MountainCarContinuous-v0, Acrobot-v1, and Pendulum-v1. Our goal is to evaluate each algorithm's sample efficiency, learning stability, and final performance under a unified experimental framework. Our Git repository is available here.

1 Problem statement

The relative performance of DQN, PPO, SAC and TD3 across both discrete-action (CartPole, MountainCar, Acrobot) and continuous-action (MountainCarContinuous, Pendulum) control tasks remains unquantified under consistent settings. This study addresses this gap by conducting a fair, head-to-head comparison of the four algorithms across five classical control benchmarks. By maintaining consistent experimental conditions, systematic hyperparameter tuning, and standardized evaluation metrics, we aim to provide clear guidance on where each algorithm excels and help professionals make informed algorithm selection decisions.

2 Experimental setup

We conduct all experiments using the OpenAI Gym toolkit [2]. Each environment is initialized with a fixed random seed for reproducibility and reset at the start of every training episode. We report the mean and standard deviation over 3 independent runs for each result.

2.1 CartPole-v1

The task is to balance an inverted pendulum on a cart by applying left or right forces.

- Observation space: 4-dimensional continuous vector (cart position, cart velocity, pole angle, pole angular velocity).
- Action space: 2 discrete actions (push cart left or right).
- Reward: +1 for each timestep the pole remains upright.
- Termination: Pole angle exceeds $\pm 12^\circ$, cart position exceeds ± 2.4 units, or episode length ≥ 500 steps.

2.2 MountainCar-v0

Drive a car up a steep hill by building momentum in a valley.

- Observation space: 2-dimensional continuous space with position $\in [-1.2, 0.6]$ and velocity $\in [-0.07, 0.07]$.
- Action space: 3 discrete actions: 0 (accelerate left), 1 (no acceleration), 2 (accelerate right).
- Reward: -1 for each timestep until the goal is reached.
- Termination: Goal reached (position ≥ 0.5) or episode length ≥ 200 steps.

2.3 MountainCarContinuous-v0

Drive a car up a hill with continuous throttle control.

- Observation space: same as MountainCar-v0.
- Action space: 1-dimensional continuous force $\in [-1, +1]$.
- Reward: +100 for reaching the goal (position ≥ 0.45), minus 0.1∗squared action magnitude each step.
- Termination: Goal reached or episode length ≥ 999 steps.

2.4 Acrobot-v1

Swing up a two-link pendulum to reach a target height.

- Observation space: 6-dimensional continuous vector (cos/sin of two joint angles and their angular velocities).
- Action space: 3 discrete torques (-1, 0, $+1$) applied at the joint.
- Reward: -1 per timestep until the tip of the second link reaches a height $= -\cos(\theta_1) - \cos(\theta_2 + \theta_1) > 1.0$.
- Termination: Tip height threshold met or episode length ≥ 500 steps.

2.5 Pendulum-v1

Swing and balance a pendulum upright using torque control.

- Observation space: 3-dimensional continuous vector ($\cos(\theta)$, $\sin(\theta)$, angular velocity).
- Action space: 1-dimensional continuous torque $\in [-2, +2]$.
- Reward: $-(\theta^2 + 0.1 \cdot \omega^2 + 0.001 \cdot u^2)$, where θ is the angle normalized to $[-\pi, \pi]$, ω is the angular velocity, and u is the applied torque.
- Termination: Episode length ≥ 200 steps (no terminal state based on pendulum position).

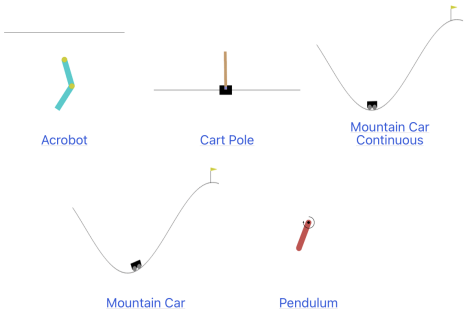

Figure 1: Classic control environments in gym

3 Algorithms

All the algorithm pseudo code implementations are available in Section A.

3.1 Deep Q-Network (DQN)

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Bigg[\left(r + \gamma \cdot \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\Bigg] \quad (1)$$

Overview: DQN [6] extends classical Q-learning [8] by approximating the action-value function $Q(s, a)$ with a deep neural network. It collects transitions (s, a, r, s'), stores them in an experience replay buffer, and samples random minibatches to break temporal correlations. A separate target network, updated periodically, provides stable bootstrap targets.
Motivation: DQN is well-suited to discrete-action tasks because it directly learns the action-value function for each possible action, reuses past experience via an off-policy replay buffer for improved sample efficiency, and stabilizes training with a separate target network that provides fixed bootstrap targets. In particular, this prevents the instability caused by simultaneously updating both the predictor and the target.

3.2 Proximal Policy Optimization (PPO)

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

Overview: PPO [7] is an on-policy policy-gradient method that alternates between gathering trajectories under the current policy and performing multiple epochs of optimization on a surrogate objective.
Motivation: PPO's key innovation is clipping the probability ratio to prevent overly large policy updates, which provides stable learning regardless of step size. The same algorithm works effectively for both discrete and continuous action spaces, maximizing the utility of each collected trajectory.

3.3 Soft Actor-Critic (SAC)

Overview: SAC [4] is an off-policy actor-critic algorithm that augments the reward with an entropy term, encouraging exploration:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[r(s_t, \mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right] \quad (2)$$

Where π is the policy, r is the reward function over the state-action pairs (s_t, a_t), α is the entropy coefficient and $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy defined as:

$$\mathcal{H}(\pi(\cdot|s)) = \mathbb{E}_{a\sim\pi(\cdot|s)}[-\log(a|s)] \quad (3)$$

It maintains a stochastic policy (actor) and two Q-value networks (critics) to reduce bias.
Motivation: SAC maximizes a combination of expected return and policy entropy, promoting sustained exploration; its off-policy actor–critic architecture with dual Q-networks mitigates value overestimation and makes efficient use of past data, delivering high sample efficiency and stability in continuous control domains.

3.4 Twin Delayed DDPG (TD3)

$$\mathcal{L}_{Q_i}(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\Bigg[\left(Q_{\theta_i}(s_t, a_t) - \left(r_t + \gamma \cdot \min_{i=1,2} Q_{\bar{\theta}_i}(s_{t+1}, a'_{t+1})\right)\right)^2\Bigg] \quad (4)$$

Overview: TD3 [3] refines DDPG [5] by addressing its overestimation and instability. It uses two critic networks and takes their minimum for target computation, adds clipped noise to target actions (target smoothing), and updates the policy less frequently than the critics.
Motivation: TD3 enhances deterministic policy gradients by using two critics to curb overestimation bias, adding clipped noise to target actions for smoother bootstrapping, and delaying policy updates relative to critic updates. The algorithm combines all this to produce more accurate value estimates and more stable policy improvement in continuous action settings.

3.5 Training

In order to facilitate the training for different algorithms and different environments, we opted for a generic code that could easily adapt to any gym environment and any RL algorithm. We use Optuna library [1] for the hyperparameter tuning. The advantage of this library is that it allowed for using different optimization methods to tune the hyperparameters other than grid search. For example, we could use random samplers or TPE (Tree-structured Parzen Estimator) samplers, which on each trial, for each parameter, fits one Gaussian Mixture Model (GMM) $l(x)$ to the set of parameter values associated with the best objective values, and another GMM $g(x)$ to the remaining parameter values. It chooses the parameter value x that maximizes the ratio $l(x)/g(x)$. this allows for a faster convergence to a good set of hyperparameters. For environments that many timesteps for optimization, this was impossible given our limited compute resources (free Google Colab). For these environments, we opted for random samplers. In addition to this, in order to guarantee for a meaningful evaluation, we extend the evaluation callback class of Optuna in order to provide evaluation across environments reset with 3 seeds. We log the mean and the standard deviation for more interpretability.
For the training curve, we use Tensorboard logging to make sure to keep track over the multiple runs.

4 Experiments and Results

4.1 DQN

4.1.1 Hyperparameters

DQN's performance depends highly on a selected set of hyperparameters (Table 1). The learning rate sets the pace at which our Q-value estimates get updated. Replay buffer size and batch size define the breadth and granularity of experience: a large buffer captures diverse situations, while batch size balances update variance against computational cost. How often you sample from this buffer (the training frequency) and how many gradient steps you take per sample control the degree of sample reuse: more frequent or repeated updates accelerate learning, but can overfit to recent experiences. Target_update_interval controls how often we update our target network, which provides stable bootstrap targets and prevents the "moving target" problem where both the prediction and target change simultaneously. If updated too frequently, learning becomes unstable; if too infrequently, learning becomes inefficient with outdated targets. The ϵ decay and learning_starts guides the agent from broad exploration toward focused exploitation. Finally, one of the most important parameters is the network architecture, especially in more complex environments like MountainCar. The depth and width of hidden layers directly govern the agent's capacity to represent nuanced value functions and extract rich features from high-dimensional inputs. It is important to note that DQN is only suitable for discrete environments, because it needs to evaluate Q-values for all possible actions to select the optimal one.

Hyperparameter	**Range**
Learning rate	$[10^{-4}, 10^{-3}]$
Batch_size	{64, 128, 256}
Discounting factor	$[0.97, 0.995]$
Gradient steps	{1, 4, 8}
Learning starts	{0, 500, 1000, 2000}
Training frequency	{1, 4, 8, 16}
Target update interval	{100, 250, 500}
Exploration fraction	$[0.05, 0.25]$
Exploration final steps	$[0.05, 0.15]$
Buffer Size	{10000, 50000, 100000}
Network architecture	{[128,128], [256,256], [512,512]}

Table 1: DQN Hyperparameter search space

4.1.2 CartPole

On CartPole (Figure 2), we saw that a learning rate of 10^{-3}, a replay buffer of 100 000 samples, and the training frequency of updating every 4 environment steps gave us the best results. With this configuration, agents routinely cleared the 200-step mark in under 30 000 timesteps. Variants with smaller buffers or lower learning rates were slower to converge. That said, the reward traces never completely smoothed out; even after hitting high scores, you can still spot point oscillations in the learning curve. This is because the uniform replay and infrequent target updates can't fully erase variance in the Q-gradient estimates.

4.1.3 Acrobot

For the Acrobot task (Figure 3), effective learning relies on careful tuning to capture the necessary swing dynamics. We employed a lower learning rate of 10^{-4}, which was crucial for enabling fine-grained policy adjustments. This is because we want subtle improvements in the swing-up trajectory to be preserved and not overwritten by larger updates, which can destabilize learning. The network architecture, a two-layer MLP with 256 units per layer, provided sufficient capacity to solve the environment. A smaller network failed to consistently reach a high reward. To maximize learning efficiency from interactions, particularly from partially successful trials, we performed four training updates for each environment step. Furthermore, a high discount factor ($\gamma = 0.99$) was maintained to emphasize the value of future rewards. We also encouraged the model to explore more (controlled by ϵ) to diversify the angles and learn well the environment before allowing the agent to converge on a robust and reliable momentum pattern. This combination of hyperparam-

eters consistently resulted in the agent achieving the full flip within approximately 60 000 timesteps across multiple experimental runs.

4.1.4 MountainCar-v0

For the challenging MountainCar task (Figure 4), we used a two-layer MLP, with 256 units in each hidden layer which was sufficient to learn the velocity-building strategies for reaching the goal, while remaining robust against overfitting to the infrequent success reward. Training updates were performed every 16 environment steps. This frequency was chosen to allow the agent to accumulate a sufficient batch of diverse experience between updates, which gave us stable learning by balancing the need for fresh data with the potentially destabilizing effect of very frequent updates on limited data. Exploration was managed using an epsilon decay schedule; epsilon was gradually reduced from 0.20 down to 0.07 over the first 20,000 timesteps. This relatively high initial exploration rate was crucial for discovering the necessary swing-back maneuver, while the subsequent, lower rate facilitated the refinement of this strategy into a consistently successful policy. With these hyper parameters, the agent reliably solved the MountainCar task, typically achieving the goal within approximately 240 000 timesteps.

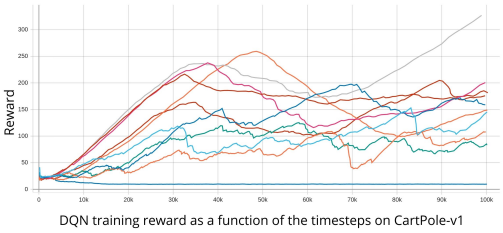

Figure 2: DQN reward as a function of training steps for different sets of hyperparameters on Cartpole

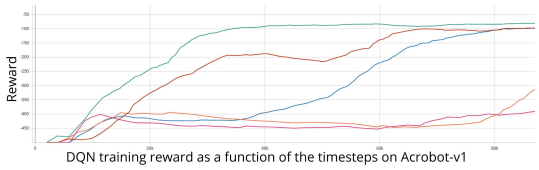

Figure 3: DQN reward as a function of training steps for different sets of hyperparameters on Acrobot

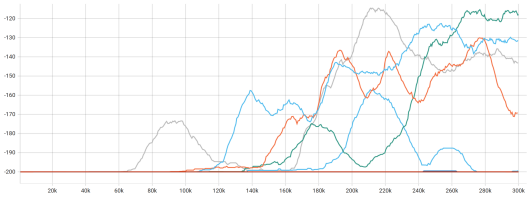

Figure 4: DQN reward as a function of training steps for different sets of hyperparameters on Mountaincar

4.2 SAC

As SAC is suited for continuous environments, it was only run on Pendulum and Mountaincar Continuous.

4.2.1 Hyperparameters

We tried a wide range of hyperparameters following the logic described in 3.5.
SAC's key hyperparameter was the entropy coefficient α. For low values of α, we simply maximize the expected return while high values of α allow the policy to be more uniform. It captures the well-known exploration vs exploitation tradeoff paradigm in reinforcement learning.

4.2.2 Pendulum

Hyperparameter	**Range**
Learning rate	$[10^{-4}, 10^{-2}]$
Batch size	{64, 128, 256}
Discounting factor	$[0.99, 0.9999]$
Entropy coefficient	$[10^{-3}, 10]$ (Log scale)
Gradient steps	[1, 10]
Training frequency	{16, 32, 64}
Buffer size	$\{10^3, 10^4, 10^5, 10^6, 10^7\}$
Network size	{(64, 64), (256, 256), (400, 300) }

Table 2: SAC Hyperparameter search space for Pendulum
These ranges resulted in trajectories presented in figure 5 where we can see that the best run converges after roughly 22k steps.

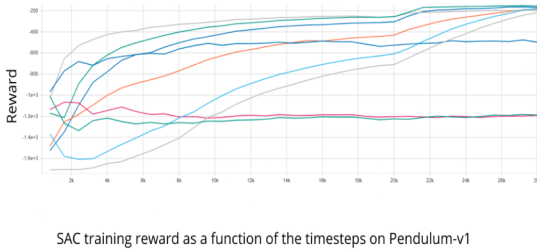

Figure 5: SAC optimization trajectories on Pendulum

The best trajectory was achieved with a learning rate of $8.13 \cdot 10^{-5}$, a batch size of 128, a buffer size of 1000, a training frequency of 12, 2 gradient steps, a network size of [64, 64] and an entropy coefficient of 0.1.
While visualizing results, an interesting thing to mention is that the terminal optimal state was achieved rotating from both left or right with the same hyperparameters. This can be explained by the fact that the entropy coefficient described in equation 3 also enhances multimodality in the state distribution. In the case of pendulum, there is a clear bimodality starting from the inital state as the environment is symmetric.

4.2.3 Continuous Mountaincar

In the Mountaincar environment, enhancing exploration is crucial. Otherwise, the car would not move. Drawing inspiration from 4.3.3, we added zero mean gaussian action noise with $\sigma = 0.01$ and shifted the range to higher entropy coefficients. Following the same logic, we also got improvements from initalizing a non-zero `log_std_init` on the policy. Drawing inspiration from Stable Baseline's hyperparameters value [1], we set the value to -3.67. Setting `SDE` to `True` also helped. The ranges of hyperparameters we tried are presented in Table 4. Other ranges were adapted from Table 2 by gauging the importance they had.

Hyperparameter	Range
Learning rate	$[10^{-4}, 10^{-2}]$
Batch size	{128, 256, 512}
Discounting factor	$[0.99, 0.9999]$
Entropy coefficient	{'auto', 0, 0.01, 0.1, 10, 100}
Gradient steps	{16, 32, 64}
Training frequency	{16, 32, 64}
Buffer size	{50000, 100000, 1000000}
Use SDE	True (fixed)
Action noise	$\mathcal{N}(0, 0.01)$ (fixed)

Table 3: SAC Hyperparameter search space for Continuous Mountaincar

It resulted in trajectories presented in figure 6 where we can see that some hyperparameters combinations fail to make the car move, thus achieving a reward of 0 while the best run achieves the maximal reward in rouhgly 20k steps.

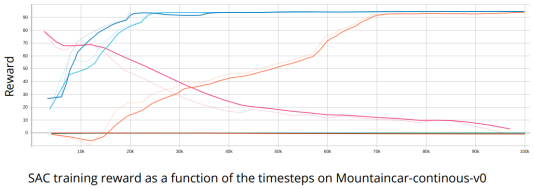

Figure 6: SAC optimization trajectories on MountainCar

The best trajectory was achieved with a learning rate of $3.3 \cdot 10^{-3}$, a batch size of 512, a discounting factor of 0.996, an entropy coefficient of 0.1, 16 gradient steps, a training frequency of 16 and a buffer size of 100000.

4.2.4 Further ablation studies

This section aims at providing better intuition about the main parameters driving SAC's performance. We still use the MountainCar Continous environment and fix every parameter to its optimal value found in 4.2.3 except for one to study its influence.

Entropy coefficient From Figure 7, we see that the only values of α that converge are $\alpha = 0.01$ and $\alpha = 0.1$. $\alpha = 0$ fails because it does not explore enough and higher values fail because they act too randomly. It is worth noting that the automatic option, which adapts the coefficient dynamically fails for mountaincar.

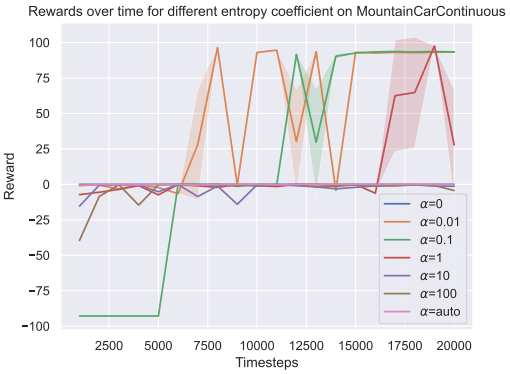

Figure 7: Reward over steps for different α

Buffer size From Figure 8, we see that convergence is achieved for all buffer size. The lowest one has the largest variance. This could be explained by the fact that the policy overfits recent transition patterns.

[1] https://github.com/DLR-RM/rl-baselines3-zoo/blob/master/hyperparams/sac.yml

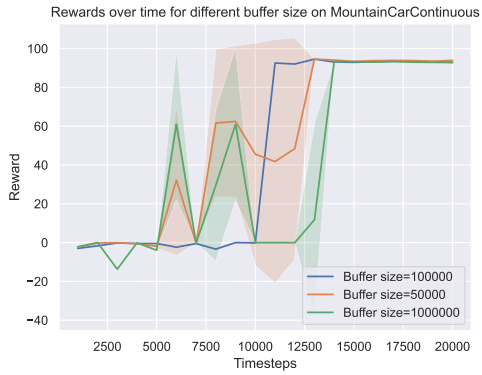

Figure 8: Reward over steps for different buffer size

Training frequency From Figure 9, we see that the largest training frequency fails to converge, which makes sense because the policy is not up to date. Lower training frequencies lead to more variance because the gradient is still noisy.

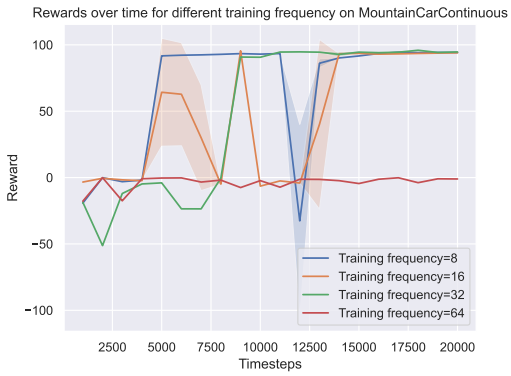

Figure 9: Reward over steps for different training frequencies

4.3 TD3

4.3.1 Hyperparameters

The TD3 algorithm relies on several key hyperparameters with a significant influence on performance. The learning rate (typically 1e-4 to 1e-3) controls how quickly the actor and critic networks learn by adjusting their weights, and the batch size (typically 256-512) defines the number of experiences to sample from the replay buffer per training iteration. The capacity of the replay buffer (typically 1e5-1e6 transitions) affects the quantity of past experience that is retained for learning. Exploration is governed by the action noise (e.g., Gaussian noise with σ=0.1-0.5) and its character (normal or Ornstein-Uhlenbeck). Target network update rate (τ) (usually 0.001-0.02) dictates how slowly target networks track the main networks. TD3-specific parameters are policy delay (generally 2-4 updates) which delay actor updates relative to critics, and target policy noise (generally 0.1-0.3) which smoothes value estimates. Discount factor γ(usually 0.95-0.99) balances between near and long-term rewards, and the neural network architecture (usually [256,256] or [400,300] layers) influences the function approximation power. Precise tuning of these parameters is central to stable learning, especially in sparse-reward environments like MountainCarContinuous as we will see in the training section.

4.3.2 Pendulum

Optimizing for pendulum wasn't very hard, the algorithm didn't need many iterations to optimize- about 20 000. Of course, it didn't converge in all cases as sometimes, there was not enough noise or exploration phase before starting learning. This led to the model not being able to perform better than random as shown in plot below.

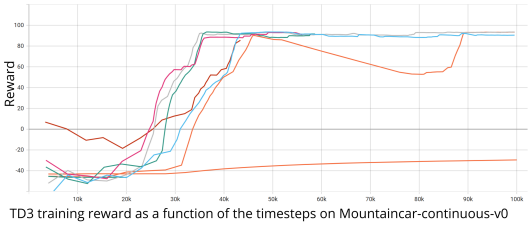

Figure 10: TD3 reward as a function of training steps for different sets of hyperparameters on Mountaincar

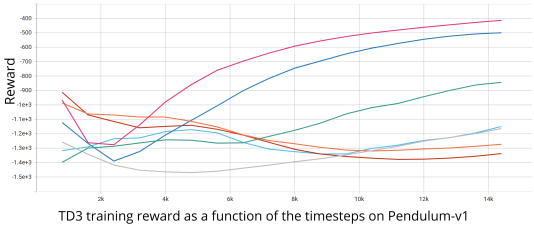

Figure 11: TD3 reward as a function of training steps for different sets of hyperparameters on Pendulum

4.3.3 Continuous Mountaincar

This behaviour is more important in Continuous Mountain Car environment where we not only need a quite high noise for exploration and long phase before starting to learn, but we also need a relay buffer size large enough to capture a diverse set of observations from which to learn. We learned this by first putting wide ranges of hyperparameters. For example the phase before learning was a range from 1000 to 20000 timesteps. This led to very poor and almost no optimization for most of the trials. The only trials that succeeded were those with hyperparameters encouraging exploration.

For that, we then refined the hyperparameter ranges in order to get better results. The plots confirm the above as we see that only the plots that start in a noisy fashion get an increase in the reward and those that start with flat behaviour, don't improve over the timesteps. We can see also that the plot with too much had a decrease in the reward before increasing at the end. This correction can be explained by the fact that at some point the relay buffer would have less noise and more of the learning that allowed for the increase so it should use these information to learn again. Other than that, we can see that the learning for both environments is very stable with mostly increasing rewards over timesteps and with clear convergence at the end. This proves the effectiveness of the TD3 algorithm due to the two critic networks and to the delayed update allowing for a more stable "target critic".

Hyperparameter	**TD3 Range**
Learning rate	$[0.0001, 0.01]$
Batch size	{64, 128, 256}
Buffer size	{10000, 100000}
Learning starts	{100, 1000, 5000}
Training frequency	{1, 2, 4}
Gradient steps	$[1, 5]$
Discounting factor	$[0.9, 0.9999]$
Policy delay	$[1, 4]$
Noise type	{normal, ou}
Noise std	$[0.4, 0.7]$
Network architecture	{[64,64], [256,256], [400,300]}

Table 4: TD3 Hyperparameter search space

4.4 PPO

PPO is an algorithm suited for continuous and discrete environments. We thus trained it on Pendulum which is continuous, Acrobot, and Cartpole, which are discrete.

4.4.1 Hyperparameters

The PPO algorithm relies on several hyperparameters, some of which significantly impact performance. The most important ones are described below.

The clipping parameter (ϵ) limits how much the policy can change at each step, preventing destabilizing updates. This is a key innovation of PPO. In RL, large updates in sharp loss landscapes can worsen performance—even when the update direction is good. The clipping term in the loss ensures that if an action was good ($r > 1$), we don't increase its likelihood excessively, and if it was bad, we don't reduce it too much.

A related hyperparameter is the maximum gradient norm, which clips gradients if their L_2 norm exceeds a threshold, stabilizing training for both the policy and value networks.

Another key parameter is the value function coefficient (c_1) in the PPO loss: $\mathcal{L}\text{total} = \mathcal{L}\text{policy} + c_1 \cdot \mathcal{L}\text{value} - c_2 \cdot \mathcal{L}\text{entropy}$. This coefficient balances learning between the policy and value networks. If too large, policy learning suffers; if too small, the value network is poorly trained, harming advantage estimates.

The entropy coefficient (c_2) controls exploration by encouraging stochastic actions. A high value promotes exploration, while a low value focuses on exploiting known good actions.

Other hyperparameters had less impact. The `use_sde` flag enables learned, state-dependent noise for exploration, improving performance in complex environments at a higher computational cost. The `learning_rate` affects optimizer updates but can be balanced by batch size or epochs. The `batch_size` and `n_epoch` control how much data is used and how often it's reused, impacting stability and convergence.

4.4.2 Pendulum

Finding the best hyperparameters for pendulum was not a challenging task. The interesting part for this environment is to notice that nearly every set of hyperparameters converged (Figure 13), but the most promising ones were those with a well-tuned max norm of gradient and clipping range.

Hyperparameter	**Range**
Discounting factor	$[0.85, 1]$
Clip range	$[0.1, 0.3]$
Entropy coefficient	$[0, 1]$
Value function coefficient	$[0.08, 12]$
Maximum norm of gradient	$[0.85, 0.95]$
gae_lambda	$[0.9, 1]$
Number of steps	{16, 512, 1024}
Number of epochs	{9, 10, 11}

Table 5: PPO Hyperparameter search space for pendulum

4.5 Acrobot

However, for the Acrobot environment (Figure 12), only a few converged.

Training stability and convergence were highly sensitive to the values of certain hyperparameters. In our experiments, we observed that two hyperparameters were especially important: the entropy coefficient and the clipping range. A suitable entropy coefficient encouraged sufficient exploration in the early phases of training, preventing the policy from prematurely collapsing into suboptimal deterministic behaviors. On the other hand, the clipping range played a crucial role in controlling the size of policy updates, ensuring that the optimization process remained within the safe bounds prescribed by the PPO algorithm.

Initially, we explored wide ranges for both hyperparameters. For example, entropy coefficients ranged from 0 to 0.1, and clipping ranges varied from 0.1 to 0.4.

We found that high entropy values led to stagnation in performance, while very large clipping ranges caused unstable updates and sudden drops in reward.
This was further confirmed through reward plots, which showed that the curves with appropriate entropy and clipping parameters exhibited steady and monotonic increases in reward, often leading to convergence. In contrast, runs with poor choices of these hyperparameters showed flat or even declining reward curves. Overall, the PPO algorithm demonstrated strong performance and stability in the Acrobot environment when these critical hyperparameters were carefully tuned.

Hyperparameter	**Range**
Discounting factor	$[0.95, 1]$
Clip range	$[0.1, 0.4]$
Entropy coefficient	$[0, 0.1]$
gae_lambda	$[0.9, 0.95]$
Number of steps	{16, 512, 1024}
Number of epochs	{9, 10, 11}

Table 6: PPO Hyperparameter search space for Acrobot

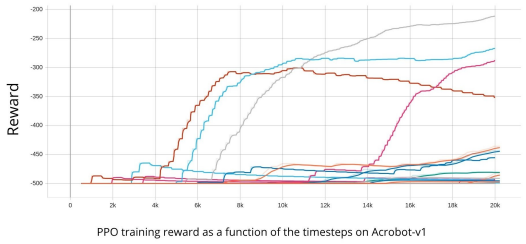

Figure 12: PPO reward as a function of training steps for different sets of hyperparameters on Acrobot

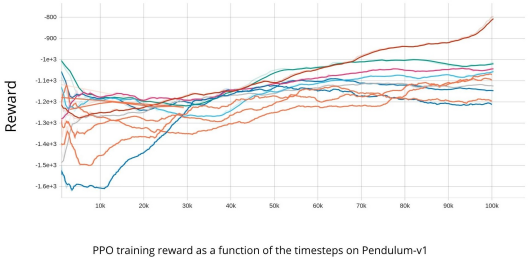

Figure 13: PPO reward as a function of training steps for different sets of hyperparameters on Pendulum

4.6 Evaluation

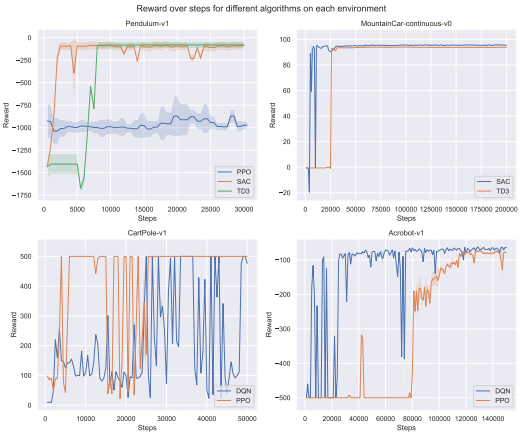

Figure 14: Model comparison on each environment

4.6.1 CartPole

PPO learns incredibly fast in CartPole. It reaches the maximum score of 500 steps in just a few thousand attempts. This happens because CartPole gives a nice, steady reward of +1 for each step the pole stays balanced. PPO's special "clipped advantage" method means it makes careful, measured improvements to its strategy, which results in a steady increase in reward. DQN struggles much more on the environment. It takes tens of thousands of attempts before it can balance the pole for 500 steps. This happens because DQN uses a more exploratory approach (sometimes making random moves to discover new strategies) and learns from past experiences stored in memory rather than just current actions. Even after it figures out how to succeed, its performance stays noisy and inconsistent.

4.6.2 Acrobot

For Acrobot, DQN solved the environment very fast. As soon as it records a flip and the success hits go to the replay buffer, the performance immediately jumps up near –100 because every single win gets reused again and again. We notice that it is a bit noisy because of the randomness in selecting batches for training. PPO can't learn until it actually swings up in fresh rollouts, so it stays at –500 for roughly 80K steps. However, once it does land a few flips, its reward climbs steadily and is more stable. So overall PPO learns slower but is steadier in the long run.

4.6.3 Pendulum

While SAC outperforms TD3 during the first iterations as it achieves the maximal reward faster, its variance is higher during the last steps, Both of these behaviors can be explained with the entropy term: by enhancing exploration during the early stages, we reach a trajec-

tory that leads to the optimal state earlier but the term also makes the trajectory shifts during the final stages.

4.6.4 MountainCar Continuous

We can draw similar conclusions than in 4.6.3 but we observe less variance in the last steps.

5 Conclusion

We evaluated DQN, PPO, SAC, and TD3 across five classical control tasks, analyzing their performance under consistent experimental settings. Our study highlights that no algorithm dominates universally; each has strengths depending on the task and action space. For beginners, we recommend starting with DQN for discrete action tasks due to its simplicity and stability. For continuous control, TD3 offers robust performance with stable updates and easier tuning. Once more familiar with RL dynamics, users can explore SAC, which converges faster but requires careful tuning. Understanding these trade-offs helps guide informed algorithm selection based on task complexity and user experience.

6 Discussion

We will answer here some of the questions that were discussed during poster session.

1- Which algorithm is more computationally expensive per iteration ?

Among the algorithms we discussed, SAC is the most computationally expensive per iteration. It maintains and updates multiple components: two Q-networks, their target networks, a stochastic policy network, and often a temperature parameter. TD3 is also relatively heavy as it uses two Q-networks and a deterministic policy. PPO, while on-policy, performs several gradient steps per batch using clipped updates, so it's moderately expensive. DQN is the least costly.

2- Which algorithm stores the policy more compactly?

In terms of how compactly the policy is stored, DQN is the most efficient because it has an implicit policy. In contrast, PPO, TD3, and SAC use explicit policy networks, which are larger and more complex. TD3's policy is deterministic, so it's more compact than PPO and SAC, which model stochastic policies. SAC stores the least compact policy since it also includes entropy regularization and a temperature parameter to encourage exploration.

3- Which one scales better for continuous actions?

DQN doesn't support continuous actions, so the choice is between SAC, TD3, and PPO. SAC is the most robust because it handles complex dynamics well. This is mainly due to entropy-based exploration and off-policy learning, but it's more computationally intensive. TD3 is also off-policy and efficient, with twin critics for stability. It's simpler and faster than SAC but less exploratory and struggles with high stochasticity. PPO supports continuous actions but is on-policy, requiring fresh data each update, which makes it less sample-efficient.

7 Limitations and improvements

Our experiments were limited by the computational constraints of free-tier Google Colab machines, restricting training duration and hyperparameter tuning. Additionally, we only used classical control environments from OpenAI Gym, which, while useful for controlled benchmarking, lack the complexity and partial observability of modern RL tasks. Using more powerful compute resources and a more diverse set of environments would enable more robust training and evaluation.

Acknowledgment

We would like to express our sincere gratitude to the teaching team of the EE-568 course for their continuous support and guidance throughout the duration of the semester.

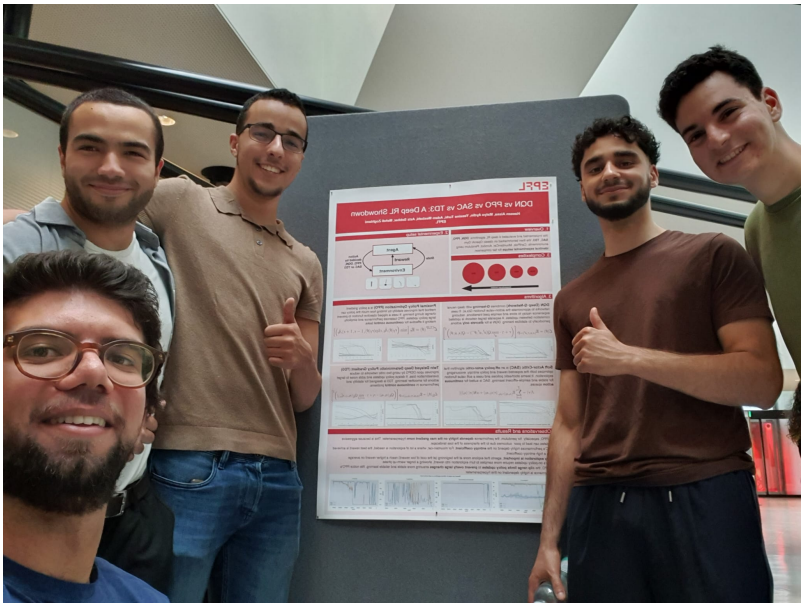

Figure 15: Poster presentation

References

[1] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG].

[2] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[3] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 1587–1596.

[4] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: 2018.

[5] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[6] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[7] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[8] Christopher Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8 (May 1992), pp. 279–292.

A Appendix

Algorithm 1 Deep Q-Network (DQN)

Require: Replay buffer $\mathcal{D}$, discount γ, update interval C, learning rate α, initial parameters θ, target parameters $\theta^- \leftarrow \theta$

for episode $= 1$ to M **do**
  Initialize state s_0
  for $t = 0$ to $T-1$ **do**
    Select $a_t \leftarrow \begin{cases} \arg\max_a Q(s_t, a; \theta), & \text{w.p. } 1-\epsilon, \\ \text{random action}, & \text{w.p. } \epsilon \end{cases}$
    Observe reward r_t and next state s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample minibatch $\{(s, a, r, s')\} \sim \mathcal{D}$
    Compute target $y \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta^-)$
    Update $\theta \leftarrow \theta - \alpha \nabla_\theta \big(Q(s, a; \theta) - y\big)^2$
    if training step mod $C = 0$ **then**
      $\theta^- \leftarrow \theta$
    end if
  end for
end for

Algorithm 2 Proximal Policy Optimization (PPO)

Require: Initial policy parameters θ, clip ϵ, epochs K, batch size B

for iteration $= 1$ to N **do**
  Collect trajectories $\{(s_t, a_t, r_t)\}$ using $\pi_{\theta_{\text{old}}}$
  Compute advantages $\hat{A}_t$ (e.g. GAE)
  for epoch $= 1$ to K **do**
    Randomly sample minibatch of size B
    Compute $r_t(\theta) \leftarrow \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
    Compute clipped loss

$$L^{\text{CLIP}} \leftarrow \mathbb{E}\Big[\min\big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\Big]$$

    Update θ by ascending $\nabla_\theta L^{\text{CLIP}}$ (plus value and entropy terms)
  end for
  $\theta_{\text{old}} \leftarrow \theta$
end for

Algorithm 3 Soft Actor-Critic (SAC)

Require: Replay buffer $\mathcal{D}$, discount γ, temperature α, learning rates α_Q, α_π, target update rate τ

Initialize parameters θ_1, θ_2, ϕ, targets $\theta_1^- \leftarrow \theta_1,\ \theta_2^- \leftarrow \theta_2$
for each environment step **do**
  Store transition (s, a, r, s') in $\mathcal{D}$
  Sample minibatch $\{(s, a, r, s')\} \sim \mathcal{D}$
  Compute target

$$y \leftarrow r + \gamma\, \mathbb{E}_{a' \sim \pi_\phi}\big[\min_{i=1,2} Q(s', a'; \theta_i^-) - \alpha \log \pi_\phi(a'|s')\big]$$

  for $i = 1, 2$ **do**
    Update critic $\theta_i \leftarrow \theta_i - \alpha_Q \nabla_{\theta_i} \big(Q(s, a; \theta_i) - y\big)^2$
  end for
  Update actor

$$\phi \leftarrow \phi - \alpha_\pi \nabla_\phi \mathbb{E}_{a \sim \pi_\phi}\big[\alpha \log \pi_\phi(a|s) - Q(s, a; \theta_1)\big]$$

  Soft-update targets:

$$\theta_i^- \leftarrow \tau\theta_i + (1 - \tau)\theta_i^- \quad (i = 1, 2)$$

end for

Algorithm 4 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Require: Replay buffer $\mathcal{D}$, discount γ, policy delay d, noise σ, clipping δ, target update rate τ

Initialize critics $Q_{\theta_1}, Q_{\theta_2}$, policy π_ϕ, targets $\theta_i^- \leftarrow \theta_i,\ \phi^- \leftarrow \phi$
for each environment step t **do**
  Store (s, a, r, s') in $\mathcal{D}$
  Sample minibatch $\{(s, a, r, s')\} \sim \mathcal{D}$
  Sample noise $\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -\delta, \delta)$
  Compute $\tilde{a}' \leftarrow \pi_{\phi^-}(s') + \epsilon$
  Compute target

$$y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i^-}(s', \tilde{a}')$$

  for $i = 1, 2$ **do**
    Update critic $\theta_i \leftarrow \theta_i - \alpha \nabla_{\theta_i} \big(Q_{\theta_i}(s, a) - y\big)^2$
  end for
  if $t \bmod d = 0$ **then**
    Update actor $\phi \leftarrow \phi + \alpha \nabla_\phi Q_{\theta_1}(s, \pi_\phi(s))$
    Soft-update targets:

$$\theta_i^- \leftarrow \tau\theta_i + (1 - \tau)\theta_i^-, \quad \phi^- \leftarrow \tau\phi + (1 - \tau)\phi^-$$

  end if
end for