American University of Armenia

Zaven and Sonia Akian College of Science and Engineering

TerraGenesis

DS232 Reinforcement Learning | Project Report

Levon Gevorgyan, Tatev Stepanyan

Instructor: *Davit Ghazaryan*

Fall 2025

Contents

Chapter 1

Problem Definition

TerraGenesis is a popular mobile idle simulator game about terraforming and colonizing other planets using real NASA science, where players manage temperature, pressure, water, and oxygen to create habitable worlds, build cities, cultivate life, and develop unique ecosystems for alien species or humanity. It's a deep strategy game focusing on balancing planetary metrics, managing resources, and overcoming challenges to transform barren worlds into thriving planets [10].

Inspired by this idea, in this project, we have recreated a simplified version of such an environment game in the scope of the *Reinforcement Learning* course, applying modern RL algorithms. We aimed to understand whether the agent can learn strategies that lead to a stable, habitable environment or fail under challenging conditions [9].

Chapter 2

Environment Description

In this project, the idea of the ***TerraGenesis*** game was taken and transformed into a custom environment made by *OpenAI Gymnasium* [2] to create a TerraGenesis world, which is focused specifically on a one-planet environment, where the planet called *Terra* is the agent that tries to perform appropriate actions to stabilize its condition and move the environment toward a healthier state. The continuous space actions performed in this simulation are described as *Oxygen, Water, Temperature,* and *Pressure,* which also serve as main parameters alongside *Biomass* and *Habitability* that describe the state of the environment.

The task of the environment is *dynamic, single-agent, partially observable, non-deterministic, sequential, continuous,* and *known*. The agent's actions are described as the first four parameters, allowing the agent to adjust those parameters in a way that improves the environment. Obstacles such as *Asteroids, Solar Flares, Volcanoes, Droughts,* and *Floods* can happen to our planet randomly, each decreasing the main important parameters: *Biomass* and *Habitability.*

Our agent's goal is to *maintain Biomass* and *increase Habitability,* which are directly connected to each other. The initial state of the environment starts randomly, with random parameters generated for the agent. The simulation stops based on the maximum steps given to an agent, or based on the fact that biomass and habitability decrease so much that failure is determined, or the agent keeps biomass and habitability stable for several consecutive states, indicating that target conditions have been achieved.

Chapter 3

Algorithms

We have selected three modern RL algorithms designed for continuous state and action spaces. All these models run using `Stable-Baseline3 2.0.0` [8], also mentioned in the `requirements.txt`.

1. ***Soft Actor-Critic (SAC)***

   SAC is an off-policy RL algorithm using entropy maximization, which encourages exploration by rewarding stochastic policies. It is well-suited for noisy and unpredictable environments such as our TerraGenesis environment [5], we have also applied *Optuna-tuned hyperparameters* [1].

2. ***Deep Deterministic Policy Gradient (DDPG)***

   DDPG is an off-policy deterministic actor-critic algorithm [6]. It relies on noise injection (in our case, *OU noise*) to ensure exploration during training. It is more sensitive to hyperparameters but still provides a useful baseline for performance in a continuous environment.

3. ***Twin Delayed DDPG (TD3)***

   TD3 improves upon DDPG using a twin critic, delayed policy updates, and target policy smoothing. This helps reduce overestimation bias and stabilize training, making it effective for continuous-control tasks such as adjusting planetary parameters [3].

Chapter 4

Training Setup

Our training configuration is fully specified in the `config.yaml` file. It includes key settings for the training procedure, environment parameters, and hyperparameters used for each RL algorithm.

Training and model implementation were carried out using *PyTorch* [7] and logged through *TensorBoard* [4].

4.1 *Training Parameters*

- *Timesteps:* 1,000,000
- *Parallel Environments:* 4
- *Save Frequency:* every 50,000 steps
- *Evaluation Frequency:* every 50,000 steps
- *Evaluation Episodes:* 5
- *Seed:* 42 (for deterministic reproducibility)

4.2 *Environment Settings*

- *max_steps:* 400
- *stable_steps_required:* 5
- *normalize observations:* true
- *clip_obs:* 10.0

4.3 *Hyperparameters for Algorithms*

1. ***SAC***
    - *learning_rate:* 4.48e-05
    - *buffer_size:* 1e6
    - *gamma:* 0.9816
    - *net_arch:* [128, 128]
2. ***DDPG***
    - *learning_rate:* 1e-4
    - *noise:* OU noise applied during training
    - *net_arch:* [400, 300]
3. ***TD3***
    - *learning_rate:* 3e-4
    - *policy_delay:* 3
    - *target_policy_noise:* 0.1
    - *net_arch:* [256, 256]

The training logic is implemented in the `train.py` file, which handles vectorized environment creation, logging to *TensorBoard* and *CSV*, auto-saving the best-performing models, and applying normalized environments via `utils.py`.

Chapter 5

Results and Plots

To understand the performance of each of the algorithms and also compare them, we analyzed the plots on *TensorBoard* generated during the training.

5.1 *Episode Length*

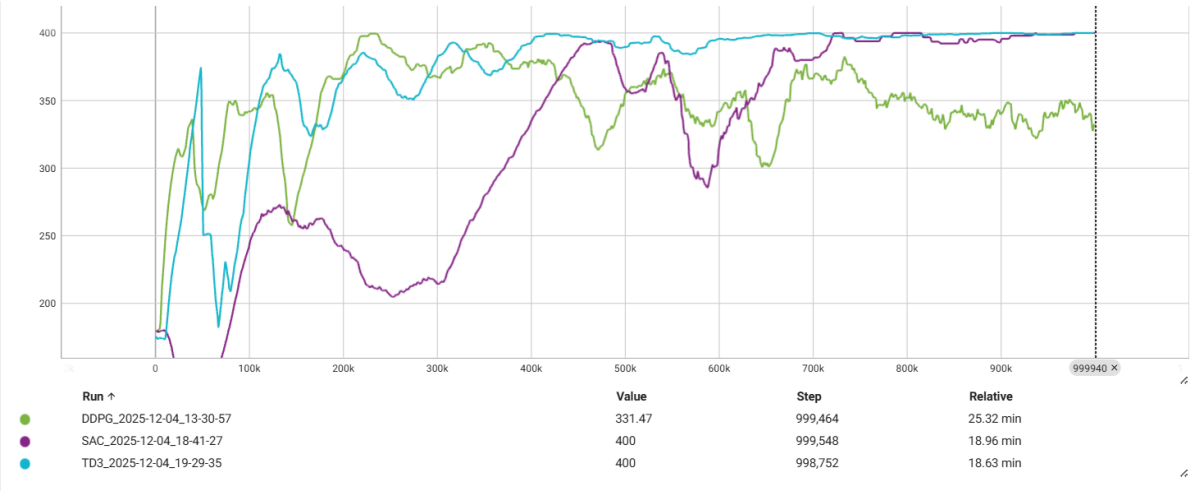

Figure 5.1: Episode length mean

As shown in Figure 5.1, the episode length graph demonstrates clear differences in stability among the three algorithms. *SAC* gradually increases its episode lengths and eventually stays close to the maximum allowed value, which indicates that it consistently avoids catastrophic failures as training progresses. *TD3* also improves quickly and maintains relatively long episodes, although with a few noticeable drops throughout training. *DDPG*, on the other hand, fluctuates heavily, showing instability in its policy.

5.2 *Episode Reward*

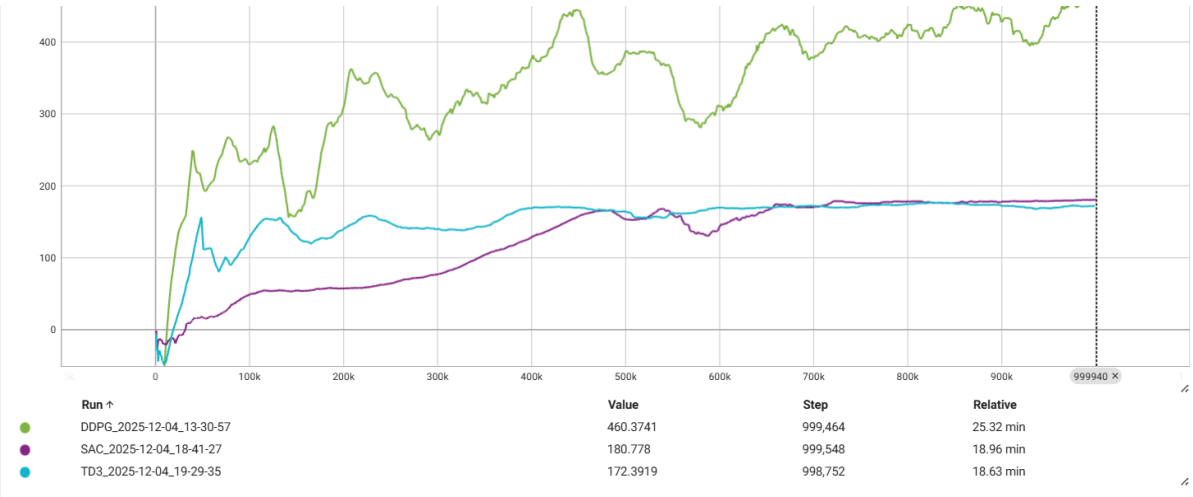

Figure 5.2: Episode reward mean

As illustrated in Figure 5.2, the reward graph reflects a similar pattern. *SAC*'s reward increases steadily and stabilizes at a consistent value. *TD3* reaches comparable reward levels but with more variability. *DDPG* shows the most irregular behavior, spiking to high rewards but collapsing afterwards, making it the most unstable of the three. Although the winner is still *DDPG*.

5.3 *Training Speed*

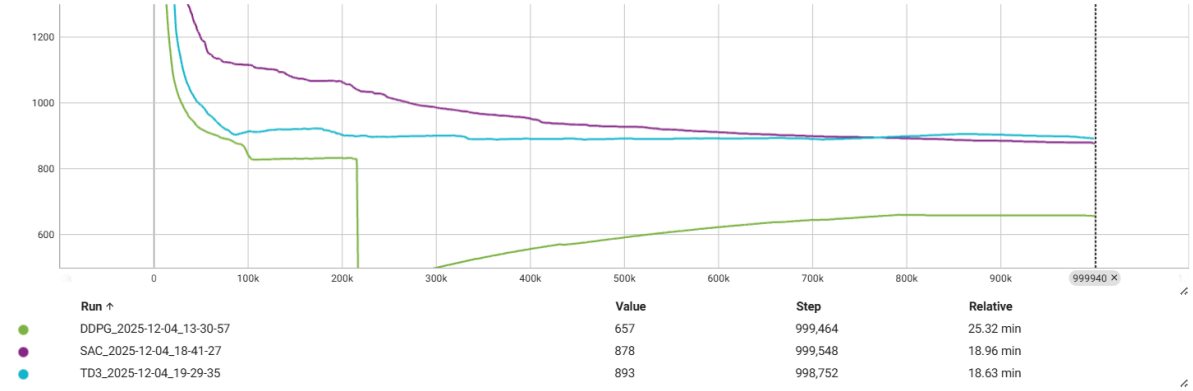

Figure 5.3: Training speed (FPS)

As seen in Figure 5.3, all three algorithms run at comparable FPS throughout the training process. The gradual decrease over time reflects the growing computational load, but no algorithm shows a significant slowdown relative to the others.

5.4 *Actor Loss*

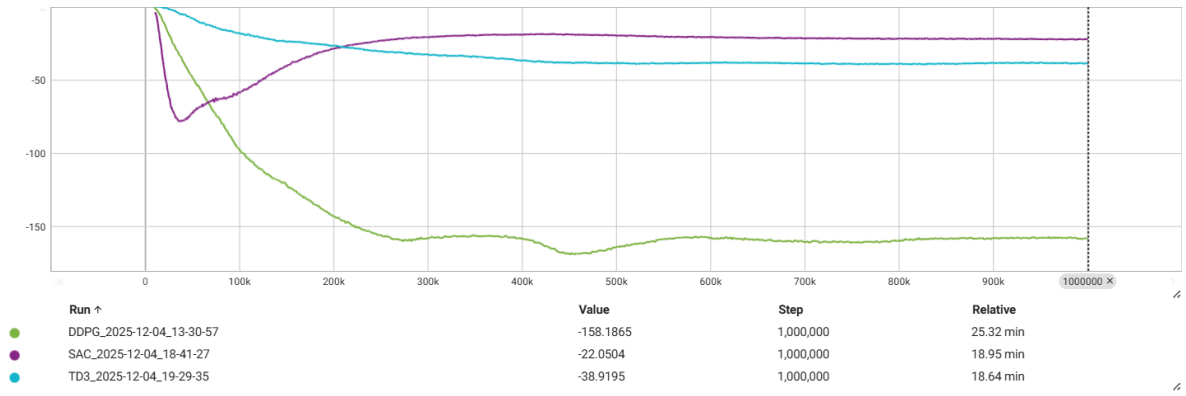

Figure 5.4: Actor loss

Figure 5.4 highlights the convergence behavior of the algorithms. *SAC*'s actor loss declines smoothly and stabilizes, indicating a well-converged policy. *TD3* follows a similar but slightly noisier pattern. *DDPG*'s actor loss decreases erratically without stabilizing, reflecting the instability observed in its reward and episode length trends.

Chapter 6

Observations and Analysis

In addition to the quantitative results shown above, several qualitative findings emerged during the development and training of the TerraGenesis environment. These findings highlight important challenges, unexpected behaviors, and adjustments that were required to obtain stable learning.

6.1 *Reward Hacking and Suicidal Policy Behavior*

During the early stages of training, particularly with *SAC* and *DDPG*, the agent frequently exploited weaknesses in the reward structure. Instead of improving planetary conditions, the agent learned *suicidal* behaviors such as rapidly decreasing core parameters to trigger early episode termination, stabilizing at unsafe values that technically satisfied local reward maxima, and making extreme oscillatory actions to exploit cumulative reward. These forms of reward hacking occurred because the reward design was initially too sensitive to short-term parameter changes, allowing the agent to maximize reward by harming long-term *biomass* or *habitability.* To address this, the reward function was redesigned to penalize extreme changes more aggressively and to encourage consistent, sustainable parameter adjustments. After restructuring the reward and adding stability requirements, the suicidal policies disappeared.

6.2 *Instability of DDPG and Improvements*

DDPG showed significant instability throughout the early training experiments. The training curve contained sharp spikes in reward, noisy and inconsistent actor loss, and high sensitivity to the *OU noise* scale. This behavior is consistent with known weaknesses of DDPG in high-variance continuous environments. After several adjustments, including noise re-scaling, normalized observations, and improved

episode termination conditions, the algorithm became substantially more stable. Although still less performative than SAC and TD3, its learning curve became smoother.

6.3 *SAC Hyperparameter Sensitivity*

Although *SAC* is typically more stable, it was surprisingly difficult to train effectively on our environment. The algorithm was extremely sensitive to learning rate, entropy coefficient (temperature), discount factor near 1, batch size and replay buffer warmup. Poor hyperparameters caused SAC either to premature convergence to near-random behavior or entropy amplification — so much that it produced chaotic exploratory policies. For this reason, we implemented a dedicated hyperparameter optimization script using *Optuna*. The optimized hyperparameters produced the first reliably stable SAC runs, resolving the earlier reward hacking and collapse issues.

6.4 *Challenges with Continuous Actions and Scaling*

Because the environment uses continuous control for its four main adjustable parameters *(Oxygen, Water, Temperature, Pressure)*, designing appropriate action ranges and update dynamics was non-trivial. We found several issues: actions originally had too strong effects, making the environment extremely volatile; clipping and normalization to `[-1, 1]` were necessary to stabilize training; overly aggressive actions led to giant swings in habitability/biomass, which destabilized the replay buffer; applying environmental effects (such as disasters) required careful integration to avoid unintended feedback loops.

That is why we added *action clamping, smooth update dynamics, normalization in observations, action cost terms*, which collectively reduced chaotic behavior and improved learning stability.

6.5 *Environment Complexity and Stopping Conditions*

Initially, the simulation lacked clear terminal conditions, allowing the agent to abuse long episodes. Adding *maximum episode length, failure cutoff thresholds* and *stability-based success termination* resulted in more meaningful reward trajectories. These conditions also prevented the agent from maintaining harmful but stable states simply to prolong the episode and accumulate reward.

Chapter 7

Challenges and Future Improvements

One of the main challenges in this project was dealing with the randomness of the environment, which created instability similar to challenges reported in continuous-control RL studies [3] [6]. Sudden catastrophic events such as asteroids or floods caused sharp drops in *habitability* and *biomass,* making the learning process highly unstable for some algorithms, especially *DDPG*. Reward shaping and observation normalization required careful tuning [9]. The computational load was another challenge due to long training horizons.

The following limitations are present. *First,* single-planet environment limiting generalization and policy diversity. *Second,* as reward shaping strongly influences agent behavior, small changes can lead to reward hacking or suicidal policies, as has been observed. *Third,* partial observability, the environment hides internal dynamics of disasters and ecosystem interactions.

Model-based reinforcement learning would allow the agent to internally simulate planetary dynamics, significantly reducing sample inefficiency and making long-term planning more feasible in environments where delayed consequences are common. Additionally, incorporating uncertainty modeling — such as ensemble methods, or stochastic world models — could improve the agent's ability to handle rare but catastrophic events like *solar flares* or *asteroid impacts.*

Another promising direction is improving the realism of environmental processes. Instead of instantaneous parameter updates, future versions could implement differential equation–based climate evolution, resource cycles, atmospheric chemistry interactions, or even biological growth models inspired by ecological simulations. These additions would create a richer and more scientifically grounded environment, making optimal policy learning both more challenging and more meaningful.

Chapter 8

Conclusion

The results show clear differences in how the three algorithms adapt to the randomness of the ***TerraGenesis*** environment. *SAC* consistently develops the most stable behavior [5], maintaining habitability even when disasters occur. *TD3* performs moderately well, learning a generally reliable policy but still showing noticeable fluctuations [3]. *DDPG* remains the most unstable, often failing shortly after reaching good performance [6].

A key pattern is that algorithms with stochastic or smoothed updates cope better with catastrophic events. *SAC* and *TD3* recover more effectively from drops in Biomass or Habitability, while *DDPG* tends to make abrupt parameter changes that lead to instability. This matches the convergence patterns described in the reinforcement learning literature [9].

The interactive ***UI*** further supports these observations via visualizations.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masaaki Koyama. Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019. arXiv:1907.10902.

[2] Farama Foundation. Gymnasium: A toolkit for developing and comparing reinforcement learning algorithms. `https://gymnasium.farama.org/`, 2023.

[3] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning (ICML)*, 2018. arXiv:1802.09477.

[4] Google. Tensorboard: Visualizing learning. `https://www.tensorflow.org/tensorboard`, 2025.

[5] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International Conference on Machine Learning (ICML)*, 2018. arXiv:1801.01290.

[6] 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. *International Conference on Learning Representations (ICLR)*, 2016. arXiv:1509.02971.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[8] Antonin Raffin, Ashley Hill, Adam Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning imple-

mentations. https://github.com/DLR-RM/stable-baselines3, 2023. Version 2.0.0.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[10] TerraGenesis Wiki Contributors. Terragenesis wiki: Main page. https://terragenesis.fandom.com/wiki/Main_Page, n.d. Accessed: 2025-12-07.