REINFORCEMENT LEARNING PROJECT
Master in Data Science and Advanced Analytics

NOVA Information Management School
Universidade Nova de Lisboa

Evaluating Reinforcement Learning Algorithms for Control Tasks

A Comparative Study of RL Methods on Lunar Lander and Car Racing

Group Elements

Alexandre Gonçalves, 20240738

Gaspar Pereira, 20230984

João Henriques, 20240499

Rita Wang, 20240551

Victoria Goon, 20240550

Git-hub page: https://github.com/gpimenta42/RL_project

Course Professor:

Nuno Alpalhão

Spring Semester 2024-2025

Table of Contents

1. Introduction

Reinforcement Learning (RL) has emerged as a powerful paradigm for solving sequential decision-making problems where agents learn optimal behaviors through interaction with dynamic environments. In recent years, RL has been successfully applied across domains such as robotics, games, finance, and autonomous systems. Among the most illustrative examples are control tasks in simulated environments like Lunar Lander and Car Racing. These environments offer well-defined and diverse settings for exploring and comparing classical RL methods with deep RL approaches.

The objective of this project is to implement and evaluate multiple RL algorithms (ranging from tabular to deep learning-based approaches) on two distinct OpenAI Gym environments: Lunar Lander and Car Racing. The primary goals are to compare their performance, examine learning stability, and understand how environment characteristics influence algorithmic success. Specifically, we apply SARSA, Q-learning, DQN, Rainbow-DQN, Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC).

Our methodology is grounded in recent academic literature that explores RL in these settings. For the Lunar Lander environment, tabular methods like SARSA and Q-learning have proven effective when discretization is carefully designed to preserve relevant state dynamics [5]. Deep Q-learning and its variants, such as Rainbow-DQN, offer performance improvements by combining techniques like multi-step returns and prioritized replay [3][4]. Additionally, deep actor-critic methods such as PPO have also been shown to perform well in Lunar Lander by improving learning stability [2]. Meanwhile, for Car Racing, which features a continuous state and action space, actor-critic methods such as PPO and SAC are more suitable due to their capacity to handle high-dimensional inputs and continuous control [1][6]. The literature consistently highlights the effectiveness of PPO and SAC in achieving robust learning and generalization in complex control tasks.

To address our project's objective, we started by outlining the methodology used to select the environments and implement the chosen algorithms, as detailed in the next section.

2. Methodology

We chose two environments from the Gymnasium Box2D library: Lunar Lander and CarRacing. While both environments have continuous observation spaces, in CarRacing the observation is a 96×96×3 RGB image, whereas in LunarLander it is an 8-dimensional vector. For Car Racing, we used continuous actions, and for Lunar Lander, discrete actions (both being the default options). Using two different types of observation and action spaces allowed us to explore the applications of reinforcement learning algorithms in diverse scenarios.

2.1 Environments

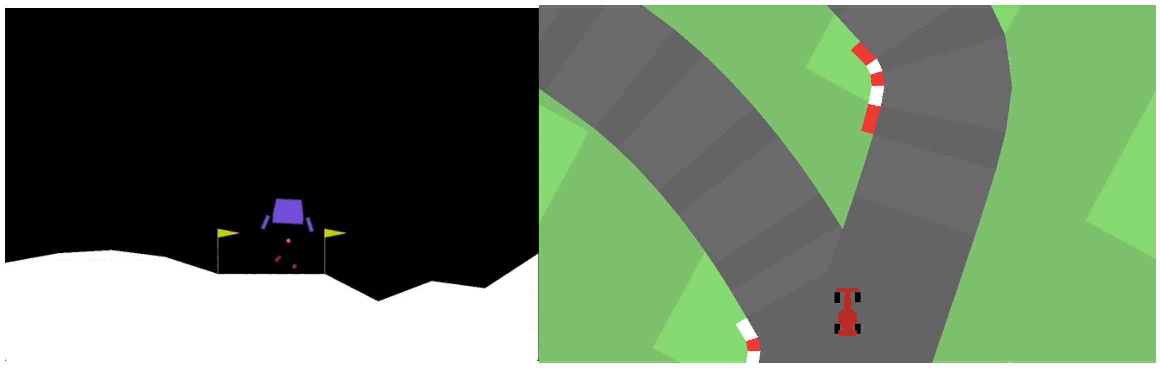

Figure 2.1: Visual Representation of the Lunar Lander and Car Racing Environments

2.1.1 Lunar Landing

The LunarLander-v3 environment simulates a 2D spacecraft tasked with landing between two flags on a flat surface. The agent controls three engines and must learn to land smoothly while minimizing fuel usage and avoiding crashes. The state space includes six continuous variables: position (x, y), velocity (x_dot, y_dot), angle, and angular velocity, along with two binary indicators signaling whether each leg is in contact with the ground. The action space is discrete, with four thrust options: no action (0), fire left orientation engine (1), fire main engine (2), and fire right orientation engine (3).

Reward shaping encourages safe landings: agents are rewarded for moving closer to the pad, maintaining stability, and softly touching down. Crashes and inefficient engine use are penalized. According to the environment's documentation, an episode is considered a successful solution if the agent achieves a total reward of at least 200 points. Episodes terminate upon crashing, flying out of bounds, becoming idle, or completing a landing.

We used the native continuous observation space but also experimented with discretizing the 8-dimensional state space into 8 bins per dimension. The two boolean dimensions (left_leg, right_leg) were excluded from binning and handled as binary values directly. This setup provided a balance between state resolution and computational efficiency, enabling us to evaluate tabular value-based algorithms.

2.1.2 Car Racing

The CarRacing-v3 environment simulates a 2D autonomous driving car that learns to drive a randomly generated track, which comprises of N tiles of varying range. The agent receives $\frac{1000}{N}$ for each track tile visited, and -0.1 penalty per frame to encourage faster completion. An episode ends when the agent has visited all tiles. If not, the episode also terminates after 1000 frames or when the agent goes too far off the track, in which case a penalty of -100 is applied. The episode is considered to be successful when the agent is able to get a reward of above 900 [8].

Although the action space can be discrete, for this project we decided to use continuous actions, which consists of steer, gas and brake, represented by a 3-D Vector (Check Table 1 for value ranges).

2.2 Algorithms

To explore different classes of RL algorithms, we implemented four value-based methods Q-learning / SARSA and Deep Q-learning (DQN) / Rainbow DQN; and implemented two actor-critic methods - Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC).

2.2.1 SARSA and Q-Learning

SARSA and Q-learning are tabular, value-based algorithms that learn action-value functions using temporal-difference (TD) updates. Both estimate expected future rewards by combining immediate feedback with bootstrapped estimates.

The key difference is that SARSA is on-policy, updating based on the action actually taken, while Q-learning is off-policy, using the best possible action for updates. These algorithms are well-suited for discrete action spaces and provide strong baselines for comparison. They typically explore the environment using an epsilon-greedy policy, which balances exploration (trying new actions) during early training and exploitation (choosing the best-known action) as learning progresses.

2.2.2 DQN and Rainbow-DQN

DQN uses a neural network that receives the continuous state as input and outputs the Q-values for each possible discrete action. During training, the network's weights are updated using the gradient of the loss function, typically the mean squared error (MSE) between the target and predicted Q-values (that is, the TD error) for a mini-batch of n state-action pairs sampled from an experience replay buffer. This random sampling allows the agent to learn from past experiences while breaking the correlation between consecutive samples, which would otherwise bias the training process. The target Q-values are computed using a second network with the same architecture (the target network) which is updated periodically with the weights of the main network to further stabilize training. The agent's policy is typically derived from the Q-values predicted by the main network using also an epsilon-greedy strategy.

We also implemented Rainbow DQN, an enhanced version of DQN that integrates several improvements to boost learning efficiency and stability. Notably, Rainbow DQN replaces the traditional epsilon-greedy policy with "noisy" networks, which inject learnable noise into the network's weights to encourage exploration throughout training. This results in more consistent and adaptive exploration compared to fixed or decaying epsilon strategies. Additional enhancements—such as multi-step returns, prioritized experience replay, dueling architecture, distributional Q-learning, and double Q-learning—are summarized in Table 2 of the appendix.

2.2.3 Proximal Policy Optimization (PPO)

PPO (Figure 1) is an on-policy algorithm that trains a neural network using an actor-critic architecture to learn a policy that maximizes an objective function $L^{PPO}(\theta)$, and improves expected return while updating the policy in a stable manner [9]. The actor maps an observation s_t to an action a_t while following a policy $\pi_\theta(a_t|s_t)$, and the critic estimates the value function $V_\theta(s_t)$.

For training, the agent interacts with the environment to collect a batch of experiences, known as a rollout, and then PPO performs multiple gradient updates, each using a mini-batch sampled from this rollout, over a defined number of epochs, to update the policy parameters θ [11]. Note that although PPO is conceptually designed to maximize the objective function, in practice, this is achieved by minimizing the loss function, which is the negative of $L^{PPO}(\theta)$.

This loss function is composed of a clipped surrogate objective $L_t^{CLIP}(\theta)$, which is what prevents the policy from changing too drastically in a single update, ensuring a more stable learning. A value function loss $L_t^{VF}(\theta)$ to train the critic and an entropy bonus $S[\pi_\theta](s_t)$ to encourage exploration during training (Check Table 3 for more details).

2.2.4 Soft Actor Critic (SAC)

SAC is an actor-critic algorithm designed for continuous action spaces. It is derived from algorithms like Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3), which extend Q-learning to the actor-critic framework.

SAC also follows an actor-critic architecture but introduces an important innovation: it aims to maximize both the expected return and the entropy of the policy. The entropy term encourages stochasticity in the policy, promoting better exploration and robustness.

Unlike PPO, which is an on-policy algorithm, SAC is off-policy and uses a replay buffer to store past experiences. These stored transitions are used to update both the actor and the critic networks, allowing for more sample-efficient learning.

To stabilize training, SAC maintains target networks for the critics. These target networks are updated using *polyak* averaging, a technique that softly updates the target weights toward the current network weights, helping to reduce instability and overestimation.

2.3 Algorithm Implementation

We used the Stable-Baselines3 (SB3) library to implement the different algorithms, as it provides widely used and well-maintained implementations of RL algorithms based on PyTorch. Since Rainbow DQN is not included in SB3, we adapted a PyTorch implementation from Park et al. (2025).

To ensure a fair comparison between algorithms we standardized training parameters: for Q-learning and SARSA we trained both of them for 100,000 episodes and for the deep learning algorithms we trained them for 500,000 timesteps. Hyperparameter tuning was conducted using grid-search, in some cases with Optuna.

2.3.1 Lunar Landing

For the LunarLander-v3 environment, five reinforcement learning algorithms were implemented: Q-Learning, SARSA, DQN, Rainbow-DQN and PPO.

Q-Learning and SARSA were chosen in this environment to experiment with a tabular algorithm. DQN, Rainbow-DQN and PPO were chosen for comparison between deep learning value-based and actor-critic algorithms.

We explored several hyperparameters, namely the learning rate, discount factor, exploration parameters (for Q-Learning and SARSA), and PPO-specific parameters (e.g., entropy coefficient). For DQN and Rainbow-DQN we experimented with different learning rates, batch sizes and frequency of target updates.

2.3.2 Car Racing

For CarRacing to reduce the dimensionality of the observation space, we preprocessed the environment by converting the input images to grayscale, resizing them to 84×84 pixels, and stacking four consecutive frames to provide temporal information.

We implemented SAC and PPO, as both are well-suited for continuous control tasks with continuous actions and we wanted to compare an off-policy to an on-policy algorithm.

For PPO, we started by using the default parameters from SB3 and then experimented with different entropy coefficients (0.0001 , 0.0005 and 0.001) to see if increased exploration would improve performance. With the entropy coefficient of 0.0005, we also tried increasing the rollout length to 4096 steps and batch size to 128, under the hypothesis that updating the policy with a larger and more diverse batch of data might lead to better gradient estimates and improved learning stability. For SAC, we also experimented with the default parameters and adjusted the learning rate, entropy coefficient and learning starts to improve performance. Note that for both algorithms we used the default CNN policy provided by SB3, and did not experiment with a recurrent policy, as we believed the frame stacking would be sufficient for temporal context.

2.4 Model Evaluation

Each trained agent was evaluated over 30 episodes. This number was chosen to provide a statistically meaningful estimate of average performance while keeping evaluation time feasible. Additionally, we tracked the proportion of successful episodes (returns ≥ 200 for Lunar Lander and ≥ 900 for Car Racing), as it has a clearly defined success criterion. While evaluations primarily used deterministic mode to ensure fair and reproducible comparisons, stochastic policy evaluation (action sampling) was also applied in Car Racing to gain insight into how robustly the policy performs under diverse and slightly perturbed conditions, which better reflects real-world deployment.

3. Results and Discussion

3.1 Lunar Landing

The best results of the trained agents in the Lunar Lander environment are summarized in the following Table 3.1. The hyperparameters used for training the agents are detailed in Table 4.

Table 3.1: Evaluation Metrics of Trained Agents Over 30 Episodes for Lunar Lander

Algorithm	Average Return	Standard deviation	Success rate
Q learning	-150,76	57,30	0%
SARSA	-242,28	227,06	0%
DQN	195,57	71,07	67%
Rainbow DQN	262,36 *	49,81 *	93% *
PPO	172,96	102.27	67%

Both Q-Learning and SARSA failed to demonstrate consistent improvements in our experiments. The mean evaluation return across episodes remained highly negative, with noisy fluctuations and no clear upward trend (Figure 2 and Figure 3). This suggests both algorithms struggled to refine effective policies, likely due to the high-dimensional, continuous nature of the environment, even after discretization. While prior literature suggests that tabular methods can be effective in Lunar Lander when discretization preserves the environment's structure [6], our uniform binning approach failed to capture critical dynamics. Suboptimal performance may also originate from overly aggressive epsilon decay, which limited exploration too early and led to premature convergence. These results reinforce the need for more sophisticated discretization techniques, and because we got better results with function approximation, we explored more of these methods.

In fact, function approximation methods achieved significantly better results. When comparing the training curves, we observe that all algorithms were able to improve the mean evaluation return over time (see Figure 4). Rainbow DQN appeared to converge faster and more reliably than DQN. In contrast, standard DQN showed more instability, as the mean evaluation return began to decline after 300,000 timesteps. PPO, while slower than both DQN and Rainbow DQN under the current configuration, demonstrated steady progress and may continue improving with longer training.

Rainbow DQN achieved the highest mean evaluation return of 262.36 (±49.81) and a 93% success rate, showcasing the effectiveness of combining multiple enhancements—namely multi-step learning (3 steps), prioritized experience replay (α=0.2, β=0.6), distributional Q-learning (51 atoms), dueling and double Q-learning, and noisy networks. These results were achieved by reducing the learning rate to 0.0001 (from the baseline of 0.001 in Park et al., 2025) to improve generalization and stability, as the original setting led to overfitting. Additionally, we increased the target network update frequency to every 1000 steps (matching the maximum episode length), ensuring the target network was updated after each episode. The return distribution over 30 evaluation episodes for Rainbow-DQN is shown in Figure 5. While most runs achieved returns above 200, two episodes resulted in significantly lower returns (around 30), suggesting that the model could benefit from extended training to improve stability.

The neural network-based algorithms significantly reduced the average episode length from the maximum of 1000 steps (where the agent failed to land in time) to approximately 300 steps, indicating that the agent learned to perform successful landings more efficiently and in a timely manner.

Rainbow DQN and PPO incorporate mechanisms that promote dynamic exploration during training. Rainbow DQN uses learnable noise in its network (noisy nets) to encourage exploration in action selection, while our implementation of PPO included an entropy bonus (0.01) in its objective function to maintain policy stochasticity. These strategies helped prevent premature convergence and supported continued learning, in contrast to standard DQN, which relied on a fixed epsilon-greedy strategy and exhibited a decline in return after a suboptimal peak around 300,000 steps.

3.2 Car racing

The PPO model with default parameters showed unstable training, briefly peaking mid-training before declining sharply (Figure 6), likely due to overfitting from the null entropy coefficient. While it did achieve a mean reward of 810.83 in deterministic evaluation, the agent failed to recover after going off track, indicating poor robustness.

Adding an entropy coefficient of 0.001 showed high variability during training (Figure 7) and resulted in low mean rewards. In a sample evaluation episode, the agent struggled to stay on track and veered off course early and stayed spinning, failing to recover.

Lowering the entropy coefficient to 0.0001, showed the most consistent and highest performance. It converged quickly at around 100,000 steps, and maintained stable and high rewards throughout training, except at the end (Figure 8). It achieved a mean evaluation reward of 874.59, with most episodes scoring near 900 and few low outliers, indicating strong stability (Figure 9). Although under stochastic evaluation, the reward dropped to 791.36, as the low entropy results in a largely deterministic policy.

Lastly, training with a larger rollout buffer and batch size and entropy coefficient of 0.0005 led to a steady and consistent improvement during training (Figure 10), even though the mean evaluation reward was only 640.52. Interestingly, the agent's behaviour appeared to be more refined, as it slows down before turning to avoid going off track, while in the model before it would occasionally collide with track edges. This could suggest that larger rollouts may help to develop higher-quality behaviour, but it requires longer training to reach their full potential and possibly outperform the current best model.

We also tuned the default SAC hyperparameters to improve generalization. Specifically, the learning rate was reduced from 0.0003 to 0.0001 to mitigate overfitting. The learning_starts parameter was increased from 100 to 50,000 to allow the agent to collect a more diverse set of experiences before training. With a low learning_starts, the agent often got stuck racing in circles (which avoided going off-track and incurring the -200 penalty) without sufficiently exploring other strategies. The entropy coefficient was bounded to a maximum of 0.01 initially, making the policy more greedy early on and encouraging the agent to focus on maximizing rewards from the beginning.

Figure 11 compares the training performance of SAC and the best PPO configuration. PPO, using a lower entropy coefficient of 0.0001, converged significantly faster than SAC, although both achieved similar performance at 500,000 timesteps. While entropy maximization is beneficial in this environment—helping the agent adapt to subtle track variations—too much entropy can slow convergence. The following table summarizes the best evaluation results for the best configurations for each algorithm.

Table 3.3: Evaluation Metrics of trained Agents Over 30 Episodes for Car Racing

Algorithm	**Average Return**	**Standard deviation**	**Success Rate**
PPO	874,59	167,95	43,33%
SAC	810,93	145,29	23,33%

It's also worth noting that while the results in the table reflect deterministic evaluation, the SAC model actually performed better when evaluated with a stochastic policy, improving from 810.93 (±145) to 848.38 (±119) mean return, and from 23,33% to 30% success rate. This suggests that when the policy is trained with entropy maximization in mind, evaluating it stochastically — rather than deterministically — can better reflect the benefits of exploration and result in improved generalization and performance.

Videos showcasing the top-performing agents in LunarLander (Rainbow DQN) and CarRacing (PPO and SAC) are available on the following page. In the CarRacing environment, it is noticeable that the agents—particularly those trained with higher entropy coefficients (SAC)—exhibit somewhat erratic or shaky driving behavior. This is a consequence of entropy maximization during training, which encourages policies to remain stochastic and exploratory. As a result, instead of committing deterministically to smooth control actions, the agent often samples from a broader action distribution, leading to jittery or less fluid driving trajectories.

4. Conclusion

In this project, we explored a range of reinforcement learning (RL) algorithms across two distinct environments: LunarLander-v3 and CarRacing-v3. By selecting environments with contrasting characteristics—one featuring a low-dimensional, vector-based state space with discrete actions, and the other offering high-dimensional image-based observations and continuous action spaces—we were able to assess the adaptability and performance of various RL approaches under diverse conditions.

Tabular methods such as Q-learning and SARSA performed poorly in the Lunar Lander environment, even after discretization, underscoring the limitations of uniform binning in capturing the nuances of continuous dynamics. In contrast, deep RL algorithms such as DQN, Rainbow DQN, and PPO significantly outperformed tabular methods. Among them, Rainbow DQN achieved the best results (93% success rate), leveraging a combination of improvements including multi-step learning, prioritized experience replay, distributional Q-learning, dueling networks, and noisy layers. PPO also demonstrated stable learning and competitive performance.

In the more complex CarRacing environment—characterized by high-dimensional visual input and continuous action control—actor-critic methods proved more effective. Both PPO and SAC achieved strong performance, particularly after tuning entropy-related parameters to better manage the exploration–exploitation trade-off. Notably, PPO converged more quickly and achieved a higher success rate (40%), suggesting that more deterministic policies and on-policy learning can be beneficial in structured environments with consistent feedback. This contrasts with SAC's off-policy strategy, which, while sample-efficient, can be more sensitive to hyperparameter choices and slower to converge in some settings.

Several limitations remain. These include relatively short training horizons (especially for CarRacing), limited rounds of hyperparameter tuning, and the exclusion of temporal modeling techniques such as recurrent or attention-based architectures. Moreover, evaluations were performed in deterministic mode, potentially underrepresenting the benefits of stochastic policies in exploration.

Future work could extend training to 1 million steps, explore recurrent PPO variants, apply frame-skipping strategies, incorporate advanced preprocessing (e.g., background removal), and evaluate alternative action distributions like the Beta policy. These extensions may enhance generalization, learning stability, and overall task performance.

References

[1] Yuwono, M. et al. Self-Driving Car Racing: Application of Deep Reinforcement Learning. arXiv preprint arXiv:2410.22766, 2024. https://doi.org/10.48550/arXiv.2410.22766

[2] Río, A. del et al. Comparative Analysis of A3C and PPO Algorithms in Reinforcement Learning. IEEE Access, 2024. https://ieeexplore.ieee.org/document/10703056

[3] Rodrigues, A., & Vieira, V. Optimizing Agent Training with Deep Q-Learning on a Self-Driving Environment. In 2020 IEEE Symposium Series on Computational Intelligence (SSCI). https://ieeexplore.ieee.org/document/9308525

[4] Tammewar, A. et al. Improving the Performance of Autonomous Driving Through Deep RL. Sustainability, 2023. https://www.mdpi.com/2071-1050/15/18/13799

[5] Gadgil, S., Xin, Y., & Xu, C. Solving the Lunar Lander Problem under Uncertainty using Reinforcement Learning. In IEEE SoutheastCon, 2020. arXiv preprint arXiv:2011.11850. https://arxiv.org/abs/2011.11850

[6] *Marques, J. M. M. G. Soft Actor-Critic for trajectory following in car racing simulation. M.Sc. Thesis, Instituto Superior Técnico, University of Lisbon, 2023. https://fenix.tecnico.ulisboa.pt*

[7] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N., … Stable Baselines3 Contributors. (2021). *PPO — Stable Baselines3 documentation*. In *Stable Baselines3: Reliable reinforcement learning implementations*. Retrieved June 26, 2025, from https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

[8] Petrazzini, I. G. B., & Antonelo, E. A. (2021). *Proximal policy optimization with continuous bounded action space via the Beta distribution* (arXiv:2111.02202). arXiv. https://arxiv.org/abs/2111.02202

[9] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal policy optimization algorithms* (arXiv:1707.06347). arXiv. https://doi.org/10.48550/arXiv.1707.06347

[10] Montoya, J. M., Daunhawer, I., Vogt, J. E., & Wiering, M. (2021). *Decoupling State Representation Methods from Reinforcement Learning in Car Racing*. In A. P. Rocha, L. Steels, & J. van den Herik (Eds.), *ICAART 2021 – Proceedings of the 13th International Conference on Agents and Artificial Intelligence* (Vol. 2, pp. 752–759). SciTePress. https://doi.org/10.5220/0010237507520759

[11] Curt-Park. (n.d.). *GitHub - Curt-Park/rainbow-is-all-you-need: Rainbow is all you need! A step-by-step tutorial from DQN to Rainbow*. GitHub. https://github.com/Curt-Park/rainbow-is-all-you-need

Appendix

Table 1: Car Racing Continuous Actions

Action	Range	Description
Steer	$[-1.0, 1.0]$	-1 = full left; 0 = straight; $+1$ = full right
Gas	$[0.0, 1.0]$	0 = no throttle; 1 = full throttle
Brake	$[0.0, 1.0]$	0 = no brake; 1 = full brake

Table 2: Rainbow DQN implementation

Prioritized Experience Replay	Transitions with higher temporal-difference (TD) error are given higher sampling priority, allowing the model to focus learning on more informative experiences.
Double Q Learning	Target Q-values are computed using the **main network** to select action and the **target network** to evaluate them, reducing overestimation bias.
Dueling network	The network separately estimates the **state value** and the **advantage** of each action. The Q-value is computed as their combination, enabling better learning in states where actions have similar outcomes.
Multi-step learning	Target Q-values are computed using a sequence of multiple future rewards, instead of just the immediate next reward. This accelerates learning in environments with delayed or sparse rewards.
Distributional Q-learning	Rather than predicting a single expected Q-value, the network outputs a **distribution over possible returns** for each action. This provides richer information and captures uncertainty (e.g., reward variance) in the environment.
Noisy Networks	Noise is injected into the network's weights in a trainable way, making the policy inherently stochastic. This enables effective, state-dependent exploration without relying on an external epsilon-greedy strategy.

Table 3: PPO Functions

Name	Function	Description
Action probability Ratio	$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$	How much current policy has changed from old one.
Generalized Advantage Estimation	$\hat{A}_t = \sum_{l=0}^{\infty} \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$ where $\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)$ $t \in [0, T]$	Compare how good an action is compared to the average expected outcome at that state, with reduced variance.
Clipped Surrogate Objective	$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1 + \epsilon)\hat{A}_t\right)\right]$	Ensures that the policy updates are stable and not too large.
Value Loss	$L_t^{VF} = \left(V_\theta(s_t) - V_t^{target}\right)^2$	Helps the critic learn to estimate how good a state is.
Entropy Bonus	$S[\pi_\theta](s_t)$	Encourages exploration.
Total PPO Loss	$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$	This is the loss used for gradient descent.

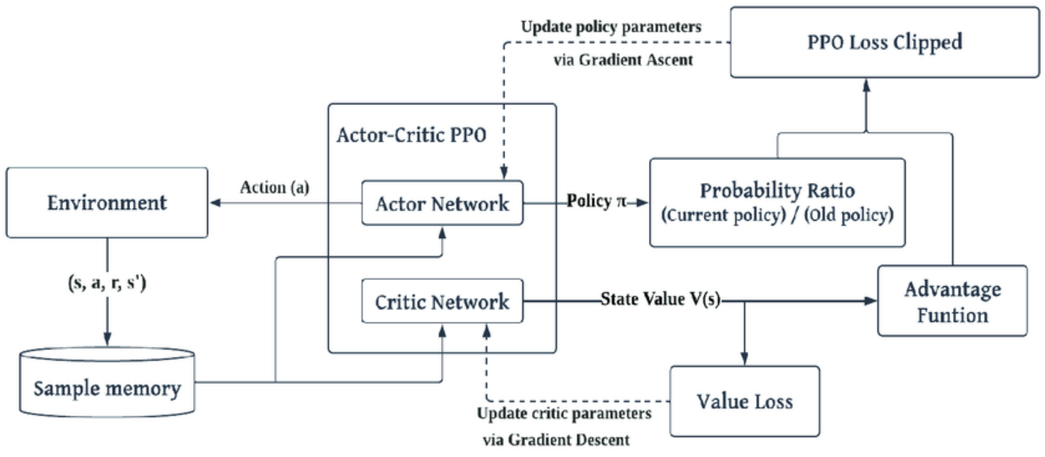

Figure 1: PPO algorithm

Table 4: Best models' hyperparameters

Model	Env	Hyperparameters
Q-learning	Lunar Lander	**Num_iterations**: 100,000; **alpha**: 0.1; **gamma**: 0.9; **epsilon**: 1.0; decay: 0.0008; **bins_per_feature**: 8;
SARSA	Lunar Lander	**Num_iterations**: 100,000; **alpha**: 0.2; **gamma**: 0.8; **epsilon**: 1.0; decay: 0.0011; **bins_per_feature**: 8;
DQN	Lunar Lander	(default from SB3)
Rainbow-DQN	Lunar Lander	**learning_rate**: 0.0001, **gamma**: 0.99, **alpha**: 0.2, **beta**: 0.6, **atom_size** = 51, **n_step** = 3, **buffer_size**: 1,000,000: **target_update**: 1,000,
PPO	Lunar Lander	**learning_rate**: 0.000996; **n_steps**: 1856; **batch size**: 128; **n_epochs**: 10; **gamma**: 0.9925; **gae_lambda**: 0.9998; **clip_range**: 0.3294; **ent_coef**: 0.0101; **vf_coef**: 0.1852; **max_grad_norm**: 0.7174
	Car Racing	**learning_rate**: 0.0003; **n_steps**: 2048; **batch_size**: 64; **n_epochs**: 10; **gamma**: 0.99; **gae_lambda**: 0.95; **clip_range**: 0.2; **ent_coef**: 0.0001; **vf_coef**: 0.5; **max_grad_norm**: 0.5
SAC	Car Racing	**learning_rate**: 0,0001, **batch_size**: 256, **learning_starts**: 50,000, **buffer_size**: 1,000,000, **ent_coef**: “auto_0.01”, **train_freq**: 1, **gradient_steps**: 1

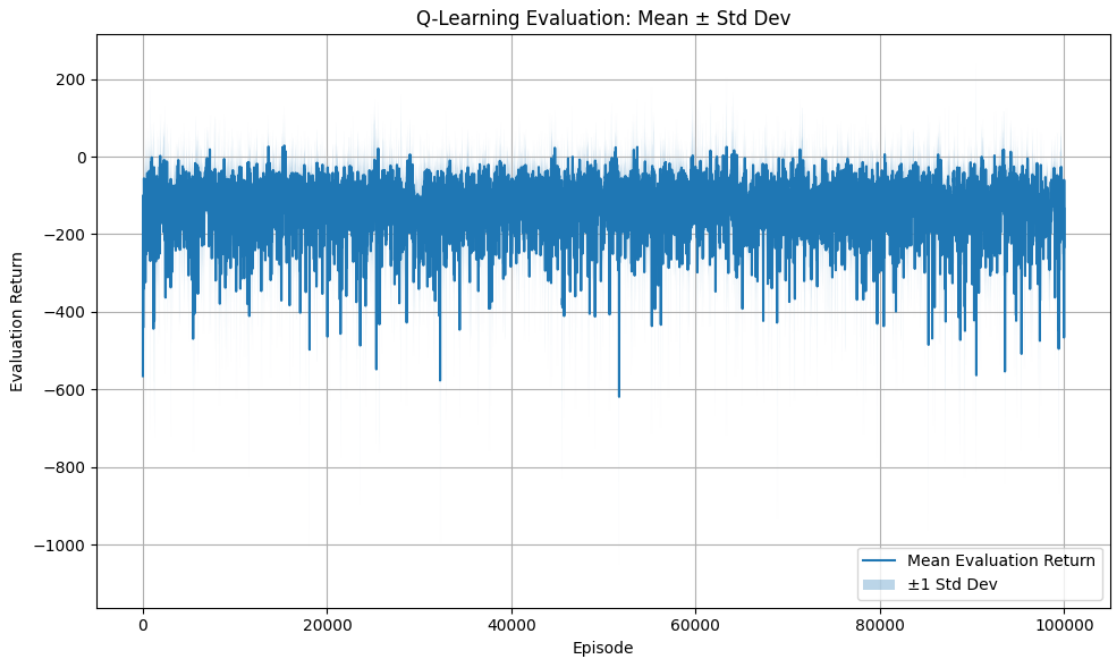

Figure 2: Q-Learning – Evaluation Return During Training (Mean ± Std Dev)

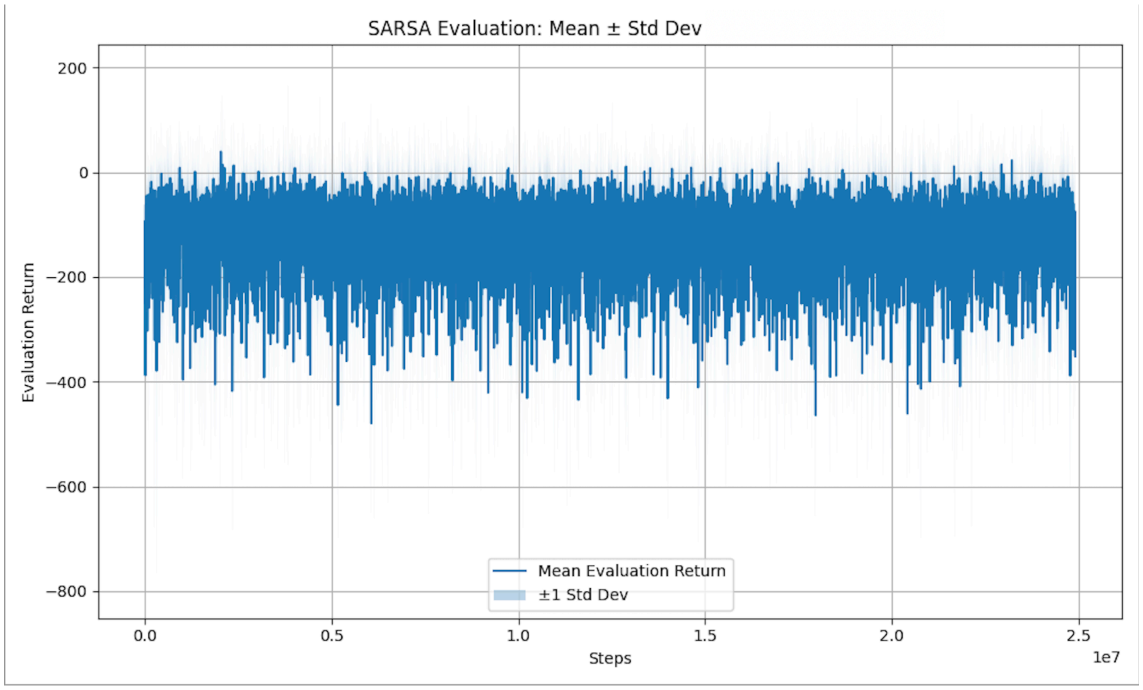

Figure 3: SARSA – Evaluation Return During Training (Mean ± Std Dev)

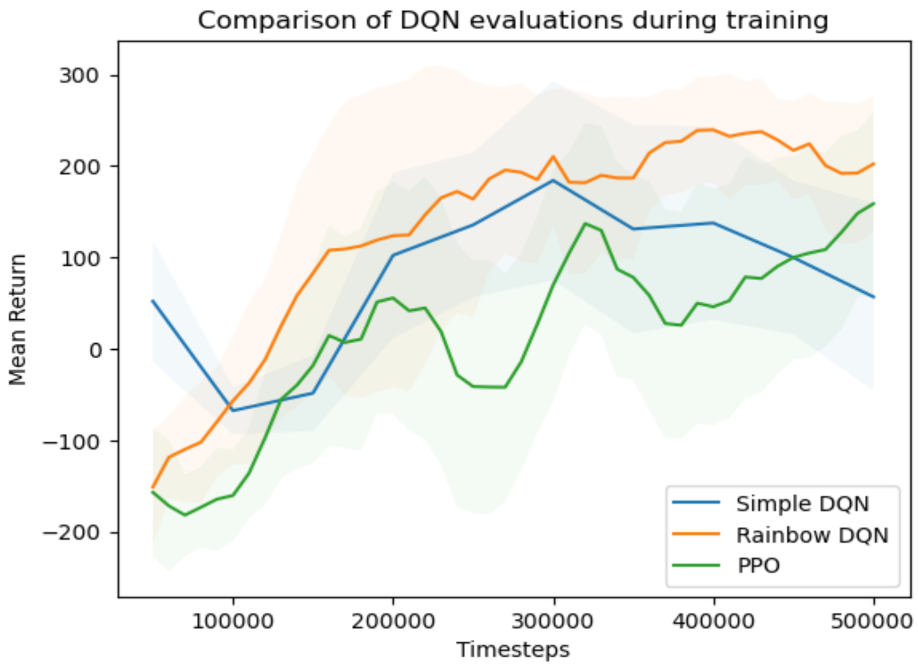

Figure 4: Algorithms evaluation during training - comparison for Lunar Lander

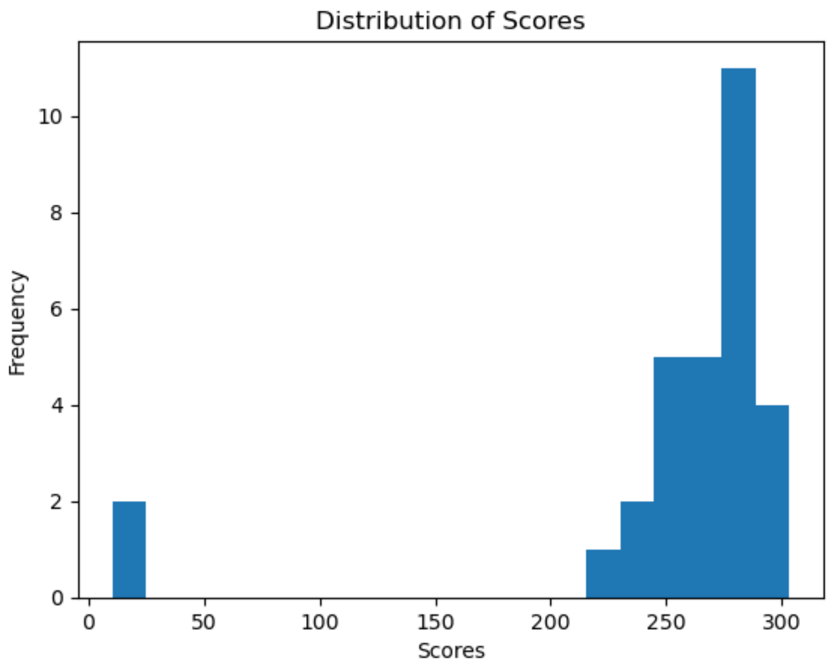

Figure 5: Distribution of return for 30 episodes in Lunar Landing – Rainbow-DQN

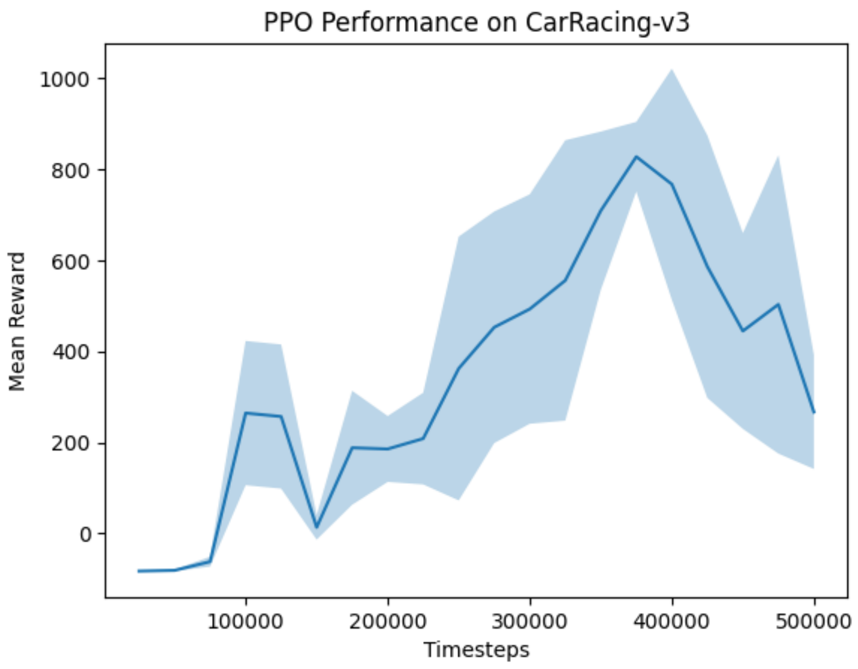

Figure 6: Training evaluation for PPO with default parameters

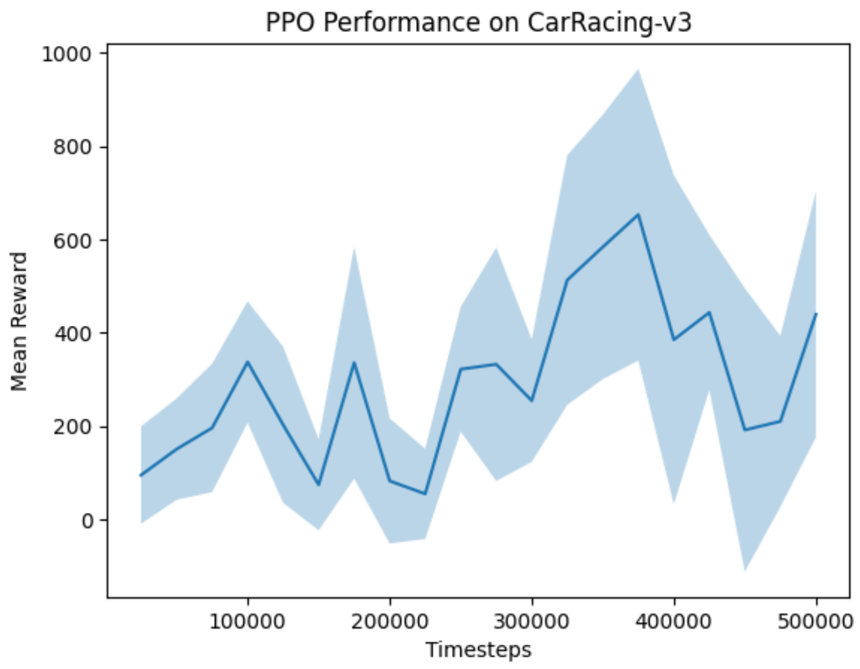

Figure 7: Training evaluation for PPO with ent_coef=0.001

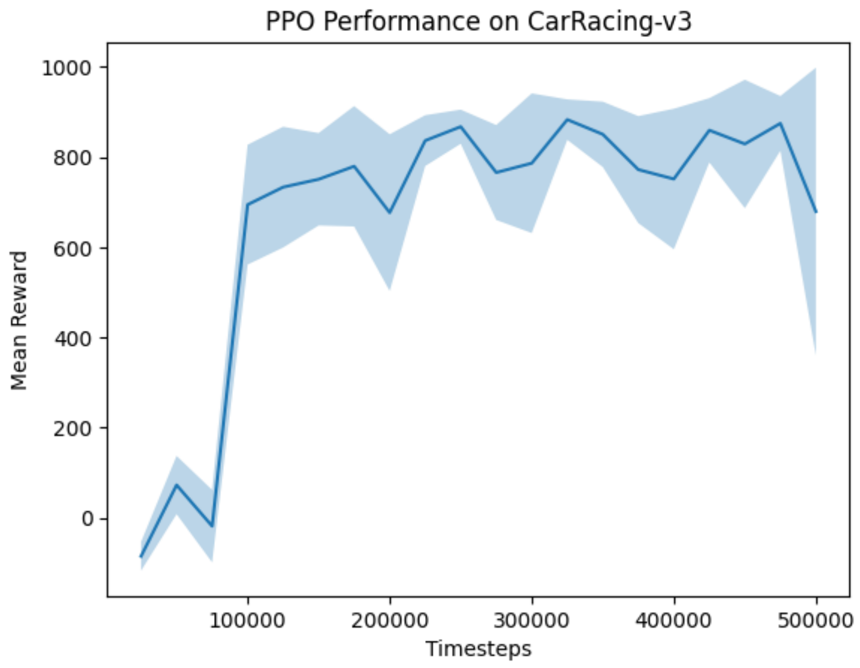

Figure 8: Algorithm evaluation for PPO with ent_coef=0.0001

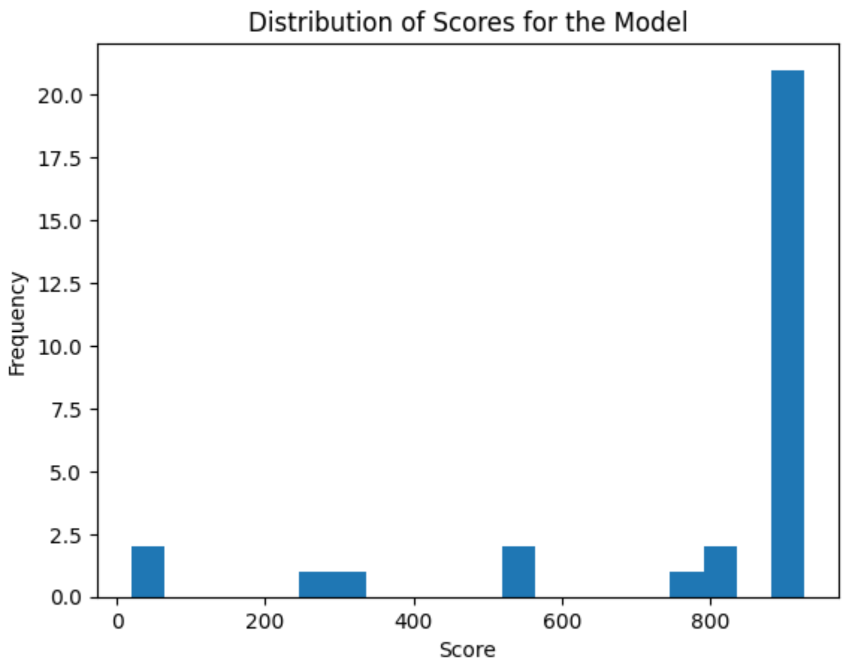

Figure 9: : Distribution of return for 30 episodes CarRacing for best model

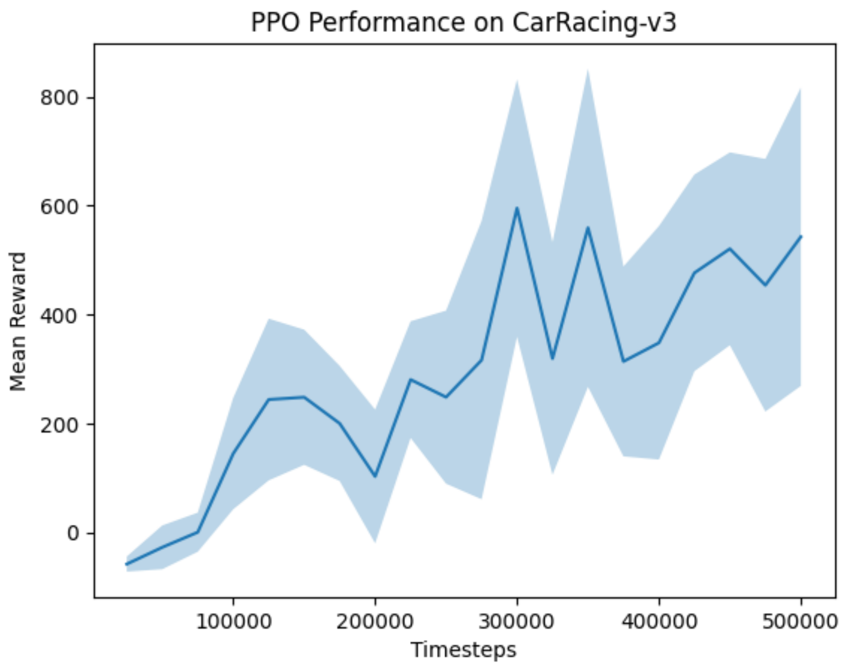

Figure 10: Algorithm evaluation for PPO with higher rollout and batch size, ent_coef=0.0005

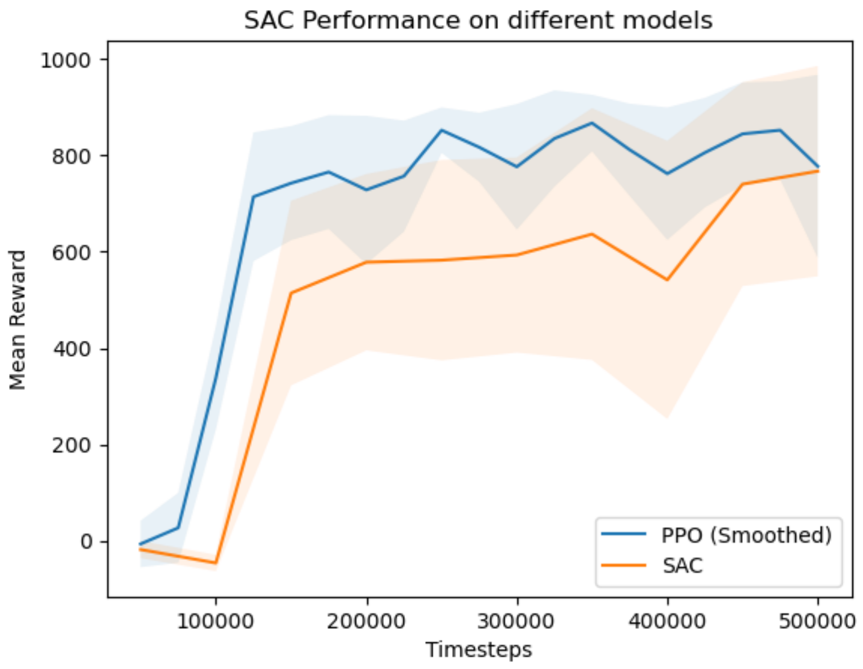

Figure 11: Comparison of evaluations during training for Car Racing