CSC_52081_EP, Reinforcement Learning Project

Anonymous Authors

***Abstract*—Reinforcement Learning (RL) trains autonomous agents to interact with complex environments. This study examines the CarRacing-v3 environment from Gymnasium, featuring high-dimensional observations and both discrete and continuous actions. We compare RL algorithms: DQN, SARSA, CEM, SAES, PPO, and SAC. Our experiments evaluate the effects of visual variability, action-space design, and hyperparameter tuning on performance. A baseline is established with a default agent. This work analyzes algorithmic trade-offs, offering insights into RL strategies for continuous control in visually complex settings.**

I. Introduction

Reinforcement Learning (RL) has become a powerful paradigm for developing autonomous agents that learn optimal behaviors through interactions with their environments. In this study, we employ the CarRacing-v3 environment provided by Gymnasium [1], which presents a challenging control task in a racing scenario. The environment is characterized by a high-dimensional observation space and two distinct modes for the action space. Specifically, the observation space consists of a top-down 96×96 RGB image capturing both the car and the racetrack, thus requiring the use of deep convolutional neural networks (CNNs) for effective feature extraction.

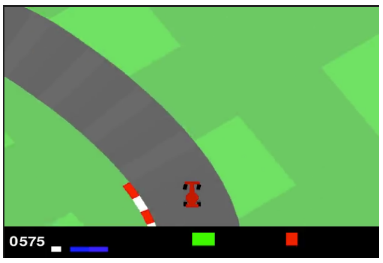

Fig. 1. A top-down 96x96 RGB image of the car and racetrack.

Regarding the action space, CarRacing-v3 supports both continuous and discrete control modalities. In the continuous mode, the agent outputs three real-valued commands: steering, where values range from -1 (full left) to $+1$ (full right); gas; and braking. Conversely, in the discrete mode, the action space is reduced to five actions: do nothing, steer left, steer right, gas, and brake. This duality in action representation allows for a comprehensive evaluation of various RL algorithms under different control settings.

The reward structure of the environment underscores the challenge by combining two components: a penalty of -0.1 per frame and a reward of $+\frac{1000}{N}$ for each new track tile visited, where N represents the total number of track tiles. For example, completing the race after visiting all N tiles in 732 frames, results in a reward of $1000 - 0.1 \times 732 = 926.8$ points, as shown in [1]. This scheme incentivize the agent to balance exploration (visiting tiles) with efficiency (minimizing frame usage), aligning its learning objectives with the task's overarching goal.

The primary objective of this project is to investigate and compare different RL policies across both discrete and continuous action modalities. For discrete action control, we implement methods such as Deep Q-Network (DQN) and SARSA. In contrast, for continuous action control, we explore approaches like the Cross-Entropy Method (CEM) and Self-Adaptive Evolution Strategy (SA-ES), and we also consider incorporating policy gradient techniques (e.g., Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC)). This comparative analysis is driven by our interest in understanding the strengths and limitations of each method in handling complex decision spaces.

The dual nature of the action space in CarRacing-v3 presents a significant challenge. When dealing with high-dimensional visual inputs, the necessity of effective feature extraction becomes paramount. To address this, our approach includes the development of a convolutional neural network architecture tailored to process the 96×96 RGB images, reducing their dimensionality while preserving essential spatial features required for decision making. Additionally, transitioning between discrete and continuous representations of actions requires careful algorithmic design and parameter tuning to ensure stable learning and convergence.

While previous studies have applied various RL techniques in simulated environments, many have tended to focus on either discrete or continuous action spaces separately. In our work, we adopt a comparative approach by evaluating different agents within the same CarRacing-v3 environment. This allows us to assess the performance of each method under similar conditions, examining aspects such as learning stability, computational complexity, and overall policy effectiveness.

At this initial stage, this work primarily outlines the methodology and anticipated challenges, rather than presenting final empirical results. Our approach involves designing the CNN-based feature extractor, implementing the chosen RL algorithms, and setting up a robust framework for comparing their performances. While our preliminary findings are yet to be finalized, we expect that this study will offer valuable insights into the practical implications of employing RL in high-dimensional, real-time control tasks.

Specific limitations of the current work include the preliminary nature of our experiments and the need for further tuning and validation. Future work will focus on extensive empirical evaluations, exploring additional policy gradient methods, and refining the network architecture to better handle the complexities of the CarRacing-v3 environment.

The code for this project is available at GitHub, providing a reproducible framework for future investigations and extensions of this work.

II. Background

A. Discrete Action Space

DEEP Q-Network (DQN) and SARSA are powerful reinforcement learning algorithms to solve discrete action control problems, suitable to our first approach to the Car Racing environment. Both approaches are based on the Q-learning algorithm, which is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function $Q(s, a)$, where s is the state and a is the action. However, DQN uses a deep neural network to approximate the Q-function, while SARSA uses a table to store the Q-values [4]. To account for the limitations of the SARSA approach in limited high-dimensional state spaces (as the Car Racing environment with an observation space of 96x96x3), we will explore a modern approach called Deep SARSA. [5]

Deep SARSA combines the on-policy nature of traditional SARSA with neural network architectures to handle large state spaces effectively. Unlike DQN's off-policy approach, Deep SARSA maintains SARSA's fundamental characteristics while scaling to complex environments.

On-policy learning methods, such as SARSA, updates the Q-values using actions selected by the current policy. (usually, the ϵ-greedy). The ϵ-greedy policy selects a random action (explore) with probability ϵ and the best action (exploit) with probability 1-ϵ.

On the other hand, off-policy learning methods, such as DQN, updates the Q-values using actions selected by a different policy. (often by greedy selection).

On-policy methods tend to be more stable, but they can be less sample-efficient and must actively explore during training. In contrast, off-policy methods can learn from experiences, which can lead to better exploration and exploitation of the environment, but may require careful tuning to ensure stability.

B. Continuous Action Space

For the second approach to the Car Racing environment, we will explore algorithms that can handle continuous action spaces. Starting from an evolutionary approach, we will test the performance of the Cross-Entropy Method (CEM) in the Car Racing environment.

The Cross-Entropy Method is a simple optimization algorithm that iteratively samples policies from a Gaussian distribution and updates the distribution parameters to maximize the expected return. It generates multiple candidate solutions (policies), ranks them based on performance, and updates the policy distribution using the best-performing candidates. It is efficient in high-dimensional spaces and can discover complex polices through population diversity. It follows the steps:

1) Sample N policies from a Gaussian distribution.
2) Evaluate the policies in the environment.
3) Select the top M policies.
4) Update the distribution parameters to fit the selected policies.
5) Repeat until convergence.

Secondly, we will explore and compare two policy-based methods designed for continuous control: Proximal Policy Optimization (PPO) [6] and Soft Actor-Critic (SAC) [7] in the Car Racing environment.

PPO is primarily an on-policy gradient method because it collects trajectories using the current policy, and updates it using only the data from the most recent rollout (episode or batch). Once data is used for training, it is discarded (unlike fully off-policy algorithms like DQN that store and reuse old experiences). It uses an actor-critic architecture to learn a parameterized policy and, even though PPO directly updates the policy, it still needs a value function (critic network) for advantage estimation (GAE - Generalized Advantage Estimation). The advantage function helps reduce variance in the policy updates.

The classic PPO algorithm follows the steps:

1) Collect trajectories using the current policy in a buffer.
2) Compute the advantage function using the critic network.
3) Update the policy using the advantage function and the policy gradient.
4) Repeat until convergence.

SAC is an off-policy (meaning it reuses past experiences for learning) actor-critic method that uses the maximum entropy framework to encourage exploration and improve sample efficiency. It learns a stochastic policy that maximizes the expected return while maximizing the entropy of the policy (avoiding premature convergence to suboptimal policies). It follows the steps:

1) Collect trajectories using the current policy.
2) Compute the advantage function using the critic network.
3) Update the policy using the advantage function and the policy gradient.
4) Update the critic network using the temporal difference error.
5) Repeat until convergence.

C. Objective

In this study, we will compare the performance of Deep SARSA and DQN in the Car Racing environment to understand the trade-offs between on-policy and off-policy learning methods in the context of discrete action space. Conversely, we also aim to compare the performance of CEM, PPO, and SAC to understand the trade-offs between evolutionary algorithm and policy-based methods in continuous action spaces.

III. Methodology / Approach

A. Environment and Agent Implementation

This study employs the CarRacing-v3 environment from Gymnasium [1], a challenging benchmark characterized by high-dimensional visual observations and multiple action modalities. In order to transform the observation space (a 3x96x96 picture frame) in the action space supported by the environment, we implemented a Convolutional Neural Network (CNN) inspired by [8] with the following characteristics:

- **Image**: the frames are transformed from an RGB state representation to grayscale, and the resolution is downscaled to 84x84 pixels. These steps facilitate the network computations, making them less complex. Additionally, to allow the network to accurately track the car's dynamics and improve the agent's ability to adjust actions based on the current state, each frame is stacked with the previous three frames. As a result, every state is represented by a sequence of four consecutive grayscale frames at 84x84 resolution (see Figure 2).

- **Input Layer**: 4-channel 84x84 images.
- **First Convolutional Layer**: 16 filters, 8x8 kernel, stride 4, ReLU activation. After applying the convolution, the output have dimensions 20x20.
- **Second Convolutional Layer**: 32 filters, 4x4 kernel, stride 2, ReLU activation. After applying the convolution, the output have dimensions 9x9.
- **Flatten Layer**: The output from the second convolutional layer is flattened into a 1D vector. The size of this vector is 32 * 9 * 9 = 2592.
- **First Fully Connected (FC) Layer**: 256 neurons, ReLU activation.
- **Second Fully Connected (FC) Layer**: The last fully connected layer defines the action space of the environment. For discrete action-space trials (DQN and Deep SARSA), this layer consists of five neurons with a linear activation function. In contrast, for continuous action-space tests (PPO, SAC, and CEM), the output layer comprises three neurons with different activation functions: *tanh* for the steering action (ranging from [-1, 1]) and *sigmoid* for the gas and brake actions (both ranging from [0, 1]).

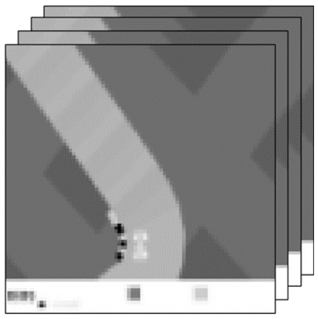

Fig. 2. Input observation space for the CNN, composed by four grayscale consecutive frames from the car racing environment. Reference: [8]

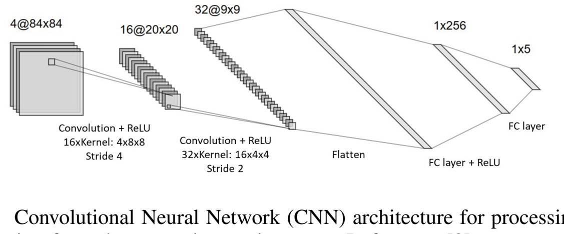

Fig. 3. Convolutional Neural Network (CNN) architecture for processing the observation from the car racing environment. Reference: [8]

B. Replay Buffer

A key innovation introduced in the original DQN paper is the concept of experience replay. This technique involves storing experiences in a replay memory buffer, allowing the agent to break the temporal dependencies between consecutive experiences. During training, random minibatches are sampled from this buffer, which improves the stability of the learning process.

C. Algorithms Design

1) DEEP Q-Network (DQN): The model is trained by optimizing a loss function based on the temporal difference error between predicted and target Q-values.

For the DQN, the target network consists in maintaining two separate networks: the main (or online) network, which is used for learning and selecting actions, and the target network, which is updated less frequently. The target network is a copy of the online network, and its parameters are periodically updated by copying the parameters of the online network to it. This approach helps stabilize the learning process by providing a fixed target for the updates, preventing oscillations and divergence in the Q-value estimates.

Training is conducted on minibatches of state-action-reward-next state sequences sampled from the replay buffer. The update rule of the algorithm is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \quad (1)$$

where $\max_{a'} Q(s',a')$ is the maximum Q-value at the next state s' (greedy selection).

2) DEEP SARSA: For the SARSA algorithm, learning is performed using a single Q-network instead of maintaining a separate target network. The algorithm follows an on-policy approach, where the agent updates its Q-values based on the actions it actually takes, rather than using a target derived from the maximum possible future reward. This ensures that updates remain consistent with the agent's current policy.

Training is conducted on minibatches of state-action-reward-next state-next action sequences sampled from the replay buffer. The update rule of the algorithm is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s', \pi(s')) - Q(s,a) \right] \quad (2)$$

where $Q(s', \pi(s'))$ corresponds to the Q-value of the next state-action pair, following the agent's current policy (ϵ-greedy). This approach ensures that the learning process accounts for the agent's actual behavior, rather than assuming a greedy action selection at every step.

3) Proximal Policy Optimization (PPO): The training for the PPO is described as follows:

- First, we collect trajectories from the environment:
$$\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=0}^{T}$$
where: s_t is the state at time t, a_t is the action taken, r_t is the reward received, and s_{t+1} is the next state.
- Compute Advantage Estimates Using GAE. The temporal difference (TD) residual is given by:
$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

The Generalized Advantage Estimation (GAE) is computed recursively:

$$A_t^{GAE} = \delta_t + (\gamma\lambda) A_{t+1}^{GAE}$$

where γ is the discount factor and λ is the GAE smoothing parameter.
The estimated return is:

$$\hat{R}_t = A_t^{GAE} + V(s_t)$$

- Compute the Probability Ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

where π_θ is the current policy, and $\pi_{\theta_{\text{old}}}$ is the old policy before the update.
- Clipped Surrogate Objective, to prevent large policy updates:

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

where ϵ is a small clipping parameter (0.2 in our code).
- Compute Value Function Loss:

$$L^{\text{VF}}(\theta) = \mathbb{E}\left[(V_\theta(s_t) - \hat{R}_t)^2\right]$$

- To encourage exploration, an entropy bonus is added:

$$L^{\text{ENTROPY}}(\theta) = \mathbb{E}\left[-\pi_\theta(a_t|s_t)\log \pi_\theta(a_t|s_t)\right]$$

- The total loss function combines policy loss, value loss, and entropy:

$$L(\theta) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 L^{\text{ENTROPY}}(\theta)$$

where c_1 and c_2 are coefficients.
- The policy parameters are updated via gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla_\theta L(\theta)$$

where α is the learning rate.

Unlike other algorithms, we used two Neural Networks (NNs): one for the policy and another for the value function. The Policy Network follows the CNN architecture described in Section III-A, with its output representing the mean and standard deviation of the action distribution. Within the PPO class, the update function constructs a normal distribution based on these parameters, from which an action is sampled. The action is then clipped to ensure it stays within the boundaries of the continuous action space.

The Value Network, in contrast, uses a simplified version of the CNN, as it does not require extensive feature extraction. Its output is a single scalar representing the estimated state value.

4) CEM:

5) SAC:

D. Comparison and Evaluation Methods

1) Action-Space Design: We compare the efficiency of discrete versus continuous action spaces by analyzing convergence rates, stability, and final policy performance across different action representations.

2) Hyperparameter Sensitivity Analysis: We investigate the stability and convergence dynamics under different step-size configurations by adapting the learning rate.

3) Performance Metrics: Our evaluation framework incorporates several key indicators to comprehensively assess the performance of the implemented methodologies. Firstly, cumulative rewards are used to quantify the overall policy efficiency and long-term reward accumulation. Secondly, convergence speed is measured by the number of training episodes required to reach a stable performance threshold. Thirdly, sample efficiency is evaluated by assessing the learning progress per unit of environmental interaction, which helps determine the algorithmic effectiveness. Lastly, robustness evaluation is conducted through perturbation tests under varying environmental conditions to gauge the model's adaptability and resilience.

E. Visual Analysis and Interpretation

To further refine our understanding of agent behavior and policy effectiveness, we employ several visualization techniques:

1) Learning Curves: We analyze episodic reward trends over the course of training to identify key inflection points and phases in the learning process. This involves plotting the cumulative rewards obtained in each episode, which helps in diagnosing the learning stability, convergence behavior, and potential over-fitting or under-fitting issues. By examining these curves, we can infer the efficiency of the learning algorithm and the impact of different hyperparameter settings.

2) Final Policy Comparisons: We conduct a comparative analysis of the final policies learned under different training configurations. This involves contrasting the strategies and behaviors exhibited by the agent after the training process is complete. In order to compare algorithm performance under the same conditions of an environment (which is randomly generated), we fixed a seed variable upon its generation to maintain the main caracteristics.

By systematically comparing these policies, we can infer the impact of various algorithmic choices, such as different exploration strategies, reward structures, or network architectures, on the overall performance and robustness of the learned policies. This comparative analysis provides a deeper understanding of the strengths and limitations of each approach, guiding future improvements and refinements.

F. Reproducibility

All implementations utilie OpenAI Gymnasium, PyTorch/TensorFlow for training, and Stable-Baselines3 for baseline comparisons. The full codebase is available at GitHub to facilitate reproducibility.

IV. Results and Discussion

A. Planned Experiments

To rigorously evaluate reinforcement learning methodologies for high-dimensional continuous control, we design a series of structured experiments that investigate different aspects of agent performance and adaptability. Our experiments encompass the following key components:

1) Baseline Performance Assessment: We begin by assessing the performance of a default agent, utilizing an unmodified algorithm with standard hyperparameter. This establishes a reference point against which subsequent optimizations and modifications will be compared. The baseline agent undergoes training in the CarRacing-v3 environment, where its cumulative reward progression, stability, and convergence rate are monitored.

2) Impact of Visual Perturbations: To evaluate robustness to varying environmental conditions, we systematically introduce color shifts during training and testing. The goal is to measure the degradation in policy effectiveness and adaptation capabilities under perturbed visual inputs.

3) Action Representation and Control Granularity: We experiment with both discrete and continuous action representations to analyze differences in learning efficiency, convergence dynamics, and policy smoothness. Additionally, varying the control resolution (i.e., finer vs. coarser action discretization) helps determine the impact of granularity on training stability and generalization.

4) Hyperparameter Optimization: To improve learning dynamics, we conduct a hyperparameter sensitivity analysis, varying key parameters such as learning rate, discount factor, batch size, and replay buffer configuration. This study aims to optimize agent performance while ensuring stability across multiple training runs.

5) Final Policy Comparison: Upon completing the experimental phase, we conduct a comparative analysis of the trained policies, evaluating their decision-making tendencies, behavioral consistency, and generalization to unseen conditions. This is complemented by qualitative visualizations such as action heatmaps and policy trajectories.

B. Preliminary Experiment with a Default Agent

As an initial baseline, we implement a standard reinforcement learning agent using the Stable-Baselines3 PPO algorithm with default hyperparameter. The agent undergoes training in the CarRacing-v3 environment for a fixed number of episodes, providing insight into:

- Initial convergence behavior and training stability.
- Sample efficiency and learning rate under default conditions.
- Performance plateau and limitations of the unoptimized agent.

To further analyze agent behavior, we generate initial learning curves and action heatmaps, highlighting policy inefficiencies and potential areas for improvement. These insights guide subsequent modifications, including fine-tuned hyperparameter, improved feature extraction architectures, and alternative training strategies.

By structuring our experiments in this manner, we establish a systematic framework for iterative enhancements, ultimately leading to a more robust and efficient reinforcement learning agent for complex continuous control tasks.

V. Conclusions

References

[1] Gymnasium. *Car Racing environment*, https://gymnasium.farama.org/environments/box2d/car_racing/, accessed 04 March 2025.

[2] Sutton and Barto. Reinforcement Learning, *MIT Press*, 2020.

[3] Read. Lecture IV - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.

[4] AI Mind. Popular Reinforcement Learning algorithms and their implementation, 2023 https://pub.aimind.so/popular-reinforcement-learning-algorithms-and-their-implementation-7adf0e092464

[5] van Seijen et al. Deep SARSA: A Novel Approach to Deep Reinforcement Learning, 2014.

[6] Schulman et al. Proximal Policy Optimization Algorithms, 2017.

[7] Haarnoja et al. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, 2018.

[8] GitHub - wiitt/DQN-Car-Racing: Implementation of DQN and DDQN algorithms for Playing Car Racing Game https://github.com/wiitt/DQN-Car-Racing GitHub. 2024.

Appendix