Deep Reinforcement Learning Agent For Car Racing Game

Qidi Fang, Tian Huang, Jiajin Huo

Abstract

This work explores the application of advanced deep reinforcement learning (DRL) techniques to the domain of racing games, a challenge characterized by high-dimensional and dynamic control requirements. We investigate the effectiveness of Double Deep Q-learning Networks (Double DQN), Actor-Critic (AC), and Proximal Policy Optimization (PPO) within the OpenAI Gym's CarRacing environment. Our findings are predicated on comparative analyses of training durations, reward trends, and specific training challenges, aiming to contribute to the broader understanding of DRL's potential in complex gaming scenarios.

1 Introduction

The exploration of Deep Reinforcement Learning (DRL) in gaming has led to significant advancements, particularly in racing games, which offer a unique blend of complexity and dynamic control challenges. This study investigates the application of DRL to the OpenAI Gym's CarRacing environment, a proxy for real-world driving scenarios. We focus on assessing the performance of notable DRL algorithms - Double Deep Q-learning Networks (Double DQN), Actor-Critic (AC), and Proximal Policy Optimization (PPO). Our investigation is tailored towards enriching the knowledge base concerning the application of DRL in solving complex gaming challenges.

2 Background and Related Work

Deep reinforcement learning (DRL) has carved a niche for itself in the gaming industry, achieving human-like or superior performances in various video games. Pioneering implementations, such as Agent57, DQN, and Implicit Quantile Networks (IQN), have set benchmarks in classic games, mastering strategic gameplay in environments with limited action spaces(Badia et al. [1], Mnih et al. [2], Dabney et al. [3]).

Transitioning to more complex and dynamic environments, DRL approaches have begun to showcase their prowess in racing games. Studies such as those by Wurman et al. illustrate DRL's potential to outclass human expertise, with algorithms rivaling champion drivers in simulations like Gran Turismo[4]. Moreover, research by Barrett et al. has enriched the dialogue on DRL applications by introducing novel racing game scenarios within OpenAI Gym and comparing the performance of different algorithms, such as Sampled Policy Gradient (SPG) and Proximal Policy Optimization (PPO)[5]. The adaptability of DRL to intricate racing scenarios has been further emphasized by Jaritz et al., who employed an Asynchronous Actor-Critic (A3C) framework for end-to-end driving using only RGB images as inputs[6].

Inspired by these developments, our project is driven by a curiosity to delve into the nuances of DRL in the context of high-dimensional continuous game environments. The OpenAI Gym's CarRacing game serves as our testbed, allowing for a comprehensive examination and comparison of algorithms like Double DQN, AC, and PPO. Through this comparative study, we aim to glean insights into the algorithms' learning behaviors, strategic decision-making processes, and adaptability to the complex dynamics of racing games.

3 Gym Box2D - Car Racing

OpenAI Gym Box2D is a toolkit for the development and comparison of reinforcement learning algorithms. It offers a variety of benchmark problems, termed environments, with a standardized interface. A notable environment in this package is the Car Racing game, which tasks an agent with controlling a vehicle on a race track as shown in Figure 1. The objective is to drive as quickly as possible while

staying on the track. This environment is particularly challenging due to the continuous state and action spaces, and the need for the agent to make complex decisions based on real-time visual input.

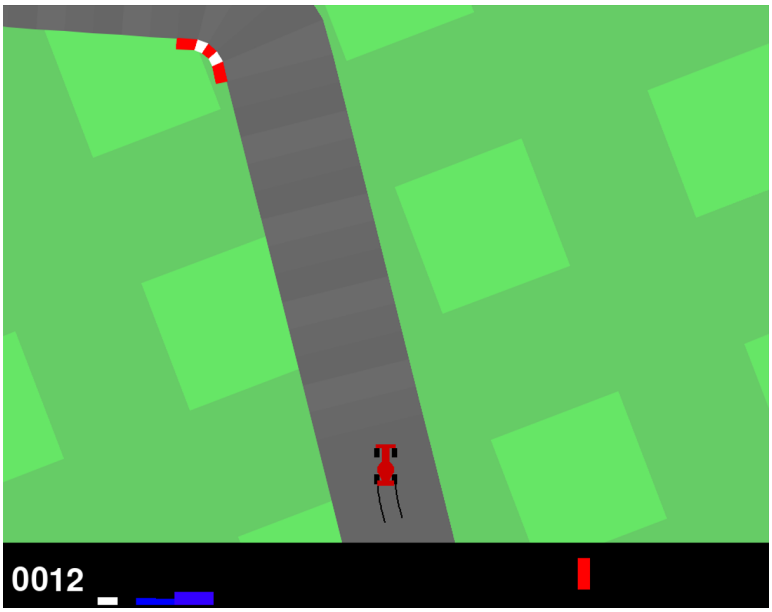

Figure 1: Screen shot of Gym Box2D - Car Racing. As shown in the picture, the track is gray, and the area outside the track is green. On the track, there are alternating dark gray and light gray stripes; these are the tiles into which the track is divided.

3.1 State Space

The state space of the Car Racing environment is represented by a $96 \times 96 \times 3$ RGB image of the game interface. Initially, this visual representation is directly utilized as the state space. This high-dimensional input captures the nuances of the environment's visual aspect, offering a comprehensive view of the car's immediate surroundings for the learning agent.

$$\text{State Space} = \mathbb{R}^{96 \times 96 \times 3} \tag{1}$$

3.2 Action Space

The action space in the environment comprises inputs for steering, gas, and braking. The initial setup offers two configurations for the action space: continuous and discrete.
Continuous Action Space: A three-dimensional array $[x, y, z]$ where $x \in$ [-1, 1], and $y, z \in [0, 1]$, correspond to the steering, gas, and braking actions respectively.

$$\text{Continuous Action Space} = \{[x, y, z] \mid x, y, z \in [0, 1]\} \tag{2}$$

Discrete Action Space: While the environment does not inherently provide a discrete action space, the official documentation offers an example of how to discretize the action space for practical use. This discretization results in a five-dimensional binary vector $[a, b, c, d, e]$, where each element corresponds to a specific action: do nothing, steer left, steer right, accelerate, and brake, respectively.

$$\text{Discrete Action Space} = \{[a, b, c, d, e] \mid a, b, c, d, e \in \{0, 1\}\} \tag{3}$$

Each element in the vector can take a value of 0 or 1, where 1 represents the execution of the corresponding action and 0 represents not executing it. This formalism allows for a straightforward representation of an agent's possible decisions at any given timestep within the environment.

3.3 Reward Function

The reward function is designed to encourage the agent to stay on the track and to do so efficiently. It comprises a constant penalty for each frame and a positive reward for passing track tiles in the correct direction.

$$\text{reward} = \begin{cases} \text{-0.1} & \text{for each frame,} \\ +1.0 & \text{for each tile passed in a counterclockwise direction,} \\ -100.0 & \text{for die.} \end{cases} \tag{4}$$

4 Wrapped Environment

In this study, the encapsulation of the environment was driven by three key considerations: Firstly, to optimize the state space by incorporating temporal dimension information without increasing computational complexity. Secondly, to facilitate a more effective comparative analysis, we discretized the action space, making it compatible with algorithms such as DQN which are not designed for continuous spaces. Lastly, we revised the reward function to introduce penalties for the car running off the track, thereby enhancing the training feedback loop for improved navigation performance.

4.1 State Space

To incorporate spatio-temporal information into the state space without increasing computational complexity, we first converted the original RGB frames to grayscale. We then stacked the three most recent grayscale frames to create a single image stack. This process allowed the agent to make informed decisions based on the immediate past trajectory of the track, effectively capturing the essence of movement and change over time within the environment.

Figure 2: The conversion process from RGB to grayscale frames is depicted. Three consecutive grayscale frames were stacked to form the state space, enriching the agent's perception with temporal information from the recent past.

4.2 Action Space

The transformation of the action space from continuous to discrete was essential for our comparative study. Recognizing the limitations of algorithms like Double DQN when dealing with continuous spaces, we simplified the control scheme. The car was programmed to move forward, turn left, and turn right at either full or half speed. To prevent the car from coming to a complete halt, braking values were set to 0.5 or 0.8. This discretization not only made the algorithms compatible across a standard environment but also allowed for a straightforward comparison of their performance.

$$\text{Action Space} = \{[x, y, z] \mid x \in \{-0.5, -1.0, 0.5, 1.0\}, y \in \{0.5, 1.0\}, z \in \{0.5, 0.8\}\} \tag{5}$$

where x represents steering, y represents gas, and z corresponds to braking. In practice, we utilized numerical representations for all possible action combinations to streamline the execution of these actions within the environment.

Index	Discrete action	Continuous action
0	Accelerate hard	$[0.0, 0.5, 0.0]$
1	Accelerate oft	$[0.0, 1.0, 0.0]$
2	Turn left hard & Accelerate soft	$[-1.0, 0.5, 0.0]$
3	Turn left hard & Brake soft	$[-1.0, 0.0, 0.5]$
4	Turn left soft & Accelerate soft	$[-0.5, 0.5, 0.0]$
5	Turn left soft & Brake soft	$[-0.5, 0.0, 0.5]$
6	Turn right hard & Accelerate soft	$[1.0, 0.5, 0.0]$
7	Turn right hard & Brake soft	$[1.0, 0.0, 0.5]$
8	Turn right soft & Accelerate soft	$[0.5, 0.5, 0.0]$
9	Turn right soft & Brake soft	$[0.5, 0.0, 0.5]$
10	Turn left hard	$[-1.0, 0.0, 0.0]$
11	Turn left soft	$[-0.5, 0.0, 0.0]$
12	Turn right hard	$[1.0, 0.0, 0]$
13	Turn right soft	$[0.5, 0, 0.0]$

Table 1: This table illustrates the mapping from discrete to continuous action space for the CarRacing environment. Each discrete action, corresponding to specific maneuvers such as acceleration and steering, is mapped to a vector representing continuous control inputs.

4.3 Reward Function

To further incentivize proper track navigation, the reward function was adjusted to introduce a penalty for off-track driving. In addition to the existing rewards and penalties of the original environment, an extra penalty is now incurred for any frame in which the car runs off the track.

$$\text{reward} = \begin{cases} -0.1 & \text{per frame,} \\ +1.0 & \text{for each tile passed in a counterclockwise direction,} \\ -0.05 & \text{extra, per frame for running off the track.} \end{cases} \tag{6}$$

5 DRL Agent

In our study, we implemented the AC, PPO, and Double DQN algorithms within the OpenAI Gym Car Racing game environment. The foundational principles of these three algorithms are explained in the subsequent sections.

5.1 Double DQN

To introduce the Double Deep Q-learning Network, it was necessary to revisit Q-learning. The Deep Q-learning Network (DQN), proposed by Mnih et al.[2], modified Q-learning to accommodate continuous states. The Double Deep Q-learning Network (Double DQN), introduced by Hasselt et al.[10], improved upon DQN.

5.1.1 Q-learning

Q-learning algorithms established a matrix-formatted table to store all action values for each state. Action values in the table represented the expected return obtainable by selecting an action in a given state and adhering to a specific policy. The Temporal Difference (TD) control algorithm was defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

The pseudo-code is shown as below:

Algorithm 1 Q-learning (off-policy TD control) for estimating $\pi \approx \pi^*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+$, $a \in A(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
for each episode **do**
    Initialize S
    for each step of episode **do**
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    end for
    Until S is terminal
end for

However, the tabular approach for storing action values inherent to traditional Q-learning is constrained to discrete state spaces. In scenarios where the state spaces are exceedingly vast, this method becomes impractical. The DQN emerged as a solution to address these limitations by approximating action values for high-dimensional and continuous spaces.

5.1.2 DQN

To solve the action-value function within environments characterized by continuous state spaces, researchers typically resorted to function approximation methods. One prominent approach involved representing the action-value function Q with a neural network. The optimization of this network was achieved by minimizing a loss function, expressed as:

$$\omega^* = \arg\min_{\omega} \frac{1}{2N} \sum_{i=1}^{N} \left[Q_\omega(s_i, a_i) - \left(r_i + \gamma \max_{a'} Q_\omega(s_i', a') \right) \right]^2$$

This formulation enabled the extension of Q-learning to accommodate deep neural network structures, giving rise to the DQN. The algorithm was operationalized through the following pseudo-code:

Algorithm 2 Deep Q-learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1 to M **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1$ to T **do**
        With probability ε select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

5.1.3 Double DQN

Traditional DQN tended to overestimate action values, which would lead to unstable training and low-quality policy[10]. The solution was to use the output of the training neural network to select the action with the highest value and use the target neural network to calculate the value of that action.

The pseudo-code is shown as below:

Algorithm 3 Double Q-learning

Initialize primary network Q_θ, target network Q'_θ, replay buffer D, $\tau \ll 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer D
  end for
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim D$
    Compute target Q value:
    $Q^*(s_t, a_t) = r_t + \gamma Q'_\theta(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
    $\theta' \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta'$
  end for
end for

The decision to not utilize Dueling DQN [11] was grounded in its architectural design, which separates the representation of state values and state-dependent action advantages. This architecture is typically more suited to games where it is not necessary to know the value of each action at every timestep.

5.2 Actor-Critic

Actor-critic (AC) methods were proposed by Konda et al. [7], providing a framework for balancing policy evaluation with policy improvement. The initial concept of AC parallels the policy-gradient methodology, introduced by Sutton et al. [8]. Initially, the value function is defined as:

$$V(s;\theta) = \sum_a \pi(a|s;\theta) \cdot Q_\pi(s,a), \quad s \in S, \quad a \in A. \tag{7}$$

In policy-gradient methods, the primary objective is to learn the parameters θ that maximize the expected value:

$$J(\theta) = \mathbb{E}_\pi[V(s;\theta)]. \tag{8}$$

This is achieved by optimizing θ via gradient ascent:

$$\theta \leftarrow \theta + \beta \nabla_\theta V(s;\theta). \tag{9}$$

The central challenge lies in computing the gradient:

$$\nabla_\theta V(s;\theta). \tag{10}$$

Upon derivation, one obtains:

$$\begin{aligned} \nabla_\theta V(s;\theta) &= \sum_a \nabla_\theta \pi(a|s;\theta) \cdot Q_\pi(s,a) \\ &= \sum_a \pi(a|s;\theta) \cdot \nabla_\theta \log \pi(a|s;\theta) \cdot Q_\pi(s,a) \\ &= \mathbb{E}_A \left[\nabla_\theta \log \pi(A|s;\theta) \cdot Q_\pi(s,A) \right]. \end{aligned} \tag{11}$$

This expectation can be estimated using the Monte Carlo method. An action $\hat{a}$ is sampled according to the policy's probability density function $\pi(\cdot|s;\theta)$, and then one computes:

$$g(\hat{a},\theta) = \nabla_\theta \log \pi(\hat{a}|s;\theta) \cdot Q_\pi(s,\hat{a}). \tag{12}$$

However, the value $Q_\pi(s,\hat{a})$ remains unknown and must be estimated. Within the AC framework, a value neural network is utilized to estimate this Q-value. Once estimated, the gradient ascent method is applied to refine the policy neural network.
For updating the value neural network using Temporal Difference (TD), the following steps are taken:

1. Compute the value functions for the current and next state-action pairs:

$$q(s_t, a_t; w_t) \quad \text{and} \quad q(s_{t+1}, a_{t+1}; w_t), \quad s \in S, \quad a \in A, \quad t \in 1 \to T.$$

2. Calculate the Temporal Difference (TD) target:

$$y_t = r_t + \gamma \cdot q(s_{t+1}, a_{t+1}; w_t), \quad \gamma = \text{Discount Rate}.$$

3. Define the loss function based on the squared difference between the TD target and the estimated value:

$$L(w) = \frac{1}{2}\left[q(s_t, a_t; w) - y_t\right]^2.$$

4. Update the weights via gradient descent:

$$w_{t+1} = w_t - \alpha \nabla_w L(w)\Big|_{w=w_t}.$$

Algorithm 4 Actor-Critic Algorithm

for episode $e = 1$ to E **do**
  Observe initial state s_t
  Randomly sample action $a_t \sim \pi(\cdot|s_t; \theta_t)$
  for $t = 1$ to T **do**
    Perform action a_t, then environment gives new state s_{t+1} and reward r_t
    Randomly sample action $a_{t+1} \sim \pi(\cdot|s_{t+1}; \theta_t)$
    Evaluate value network: $q_t = q(s_t, a_t; w_t)$ and $q_{t+1} = q(s_{t+1}, a_{t+1}; w_t)$
    Compute TD error: $\delta_t = r_t + \gamma q_{t+1} - q_t$
    Differentiate value network: $dw_t = \frac{\partial q(s_t, a_t; w_t)}{\partial w}\Big|_{w=w_t}$
    Update value network: $w_{t+1} = w_t - \alpha \delta_t dw_t$
    Differentiate policy network: $d\theta_t = \frac{\partial \log \pi(a_t|s_t;\theta_t)}{\partial \theta}\Big|_{\theta=\theta_t}$
  end for
end for

5.3 PPO

Proximal Policy Optimization (PPO) was purposed by John Schulman et al.[13]

5.3.1 Limitation of Other Policy-based Methods

Although policy-based methods like Actor-Critic algorithm are simple and intuitive, they can encounter instability during training in practical applications. The goal of policy-based methods is to parameterize the agent's policy and design an objective function that measures the quality of the policy. The policy is optimized by maximizing this objective function through gradient ascent, leading to an optimal policy. These algorithms have a clear drawback: when the policy network is a deep model, updating the parameters along the policy gradient can often lead to a significantly worse policy due to too large a step size, which in turn negatively affects the training outcome.

Therefore, the TRPO algorithm (John Schulman et al. 2015) is developed to ensure some level of safety in terms of policy performance when updating policies [12]. However, the computational process of TRPO is very complex, and the amount of computation for each update is substantial. Meanwhile, PPO algorithm (John Schulman et al. 2017) is based on the principles of TRPO, but its algorithm implementation is much simpler[13]. Moreover, a large amount of experimental results show that PPO can learn just as well or even faster compared to TRPO, making PPO a very popular reinforcement learning algorithm.

5.3.2 PPO-Clip

PPO has two forms, one is PPO-Penalty and the other is PPO-Clip. In our final project, we implement the PPO-Clip (John Schulman et al. 2017)[13]. The objective function $J(\theta)$ is defined as the expected value of the total discounted rewards starting from the initial state distribution s_0 and following policy π_θ:

$$J(\theta) = \mathbb{E}_{s_0}\left[V^{\pi_\theta}(s_0)\right]$$
$$J(\theta) = -\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\left(\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right]$$

The difference in the objective function when updating policy parameters from θ to θ' is:

$$J(\theta') - J(\theta) = \mathbb{E}_{s_0}\left[V^{\pi_{\theta'}}(s_0)\right] - \mathbb{E}_{s_0}\left[V^{\pi_\theta}(s_0)\right]$$
$$J(\theta') - J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t) + \gamma V^{\pi_{\theta'}}(s_{t+1}) - V^{\pi_\theta}(s_t)\right]$$

Defining the advantage function A:

$$J(\theta') - J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]$$
$$J(\theta') - J(\theta) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim\pi_{\theta'},a\sim\pi_{\theta'}}\left[A^{\pi_\theta}(s,a)\right]$$

Thus, as long as the expected advantage is non-negative, the policy update is guaranteed to be safe, meaning that $J(\theta') \geq J(\theta)$.

Take an approximation step. Ignore the changes in the state visitation distribution between two policies, and adopt the state distribution of the old policy directly. The optimization objective is substituted to:

$$L_\theta(\theta') = J(\theta) + \frac{1}{1-\gamma}\mathbb{E}_{s\sim\nu^\theta}\mathbb{E}_{a\sim\pi_{\theta'}(\cdot|s)}[A^{\pi_\theta}(s,a)]$$

Use importance sampling to process the action distribution in order to estimate and optimize the new policy based on the data already sampled from the old policy:

$$L_\theta(\theta') = J(\theta) + \mathbb{E}_{s\sim\nu^\theta}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A^{\pi_\theta}(s,a)\right]$$

PPO-Clip doesn't use the KL divergence term or constraints like TRPO. Instead, it uses a special clipping technique in the objective function to eliminate the motivation for the new policy to deviate significantly from the old policy. The objective is given by:

$$\arg\max_\theta \mathbb{E}_{s\sim\nu^{\pi_{\theta_k}}}\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_k}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_k}(s,a)\right)$$

where $\text{clip}(x,l,r) := \max(\min(x,r),l)$, ensures that x is clipped within the range $[l,r]$. This is a form of conservative policy update, which is also known as the clip function.
If $A^{\pi_k}(s,a) > 0$, it limits the ratio to not increase by more than $1+\epsilon$; if $A^{\pi_k}(s,a) < 0$, it limits the ratio to not decrease by more than $1-\epsilon$. In both cases, the term $A^{\pi_k}(s,a)$ ensures that the updates are bounded.

6 Implementation

In this project, we have developed on-policy AC, Double DQN, on-policy PPO algorithm and off-policy PPO algorithms in Python, adhering to an object-oriented programming paradigm. The source code is openly accessible on GitHub at: Project Repository.

Algorithm 5 PPO with Clipped Objective

Initialize policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}^{\pi_k}$ using any advantage estimation algorithm
    Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} L^{\text{CLIP}}_{\theta_k}(\theta)$$

    by taking K steps of minibatch SGD (via Adam), where

$$L^{\text{CLIP}}_{\theta_k}(\theta) = \mathbb{E}_{\tau \sim \pi_k}\left[\sum_{t=0}^{T} \min\left(r_t(\theta)\hat{A}^{\pi_k}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_k}_t\right)\right]$$

end for

DQN is inherently an off-policy algorithm, which is why we consider 'off-policy' as one of its defining characteristics. On the other hand, AC and PPO are more flexible, capable of functioning both on-policy and off-policy. Generally speaking, off-policy versions tend to outperform their on-policy counterparts, making the off-policy variant an enhanced version of the same algorithm. The original form of any algorithm is definitely on-policy. Therefore, to make a fair comparison of the strengths and weaknesses of these three algorithms, we have implemented DQN in its off-policy form and both PPO and AC in their on-policy forms.

In additon, since DQN is intrinsically limited to discrete action spaces, we also decided to maintain fairness in our experiments by adapting both AC and PPO to also work within discrete action spaces. However, after implementing the aforementioned algorithms, we observed that their performance fell short of our expectations. This discrepancy could likely be attributed to the fact that the 'car racing' environment inherently requires continuous actions, and our modification merely involved wrapping the environment to accept discrete inputs. Consequently, to truly tap into the potential of policy-based algorithms, we went a step further and implemented an off-policy version of PPO designed for continuous action spaces.

Our experimentation commenced with a comparative analysis between an on-policy AC with discretized action space as delineated in Section 4, an off-policy Double DQN with discretized action space and an on-policy PPO identical action space discretization. Additionally, Further, we implemented an off-policy PPO with continuous action spaces. The detailed configurations for our experimental setup are enumerated in the Appendix.

7 Results and Analysis

In the car racing test environment, we compared four algorithms in terms of training time and total reward per episode. We then analyzed the experimental results to evaluate their performance.

7.1 Training Time Comparison

First, we compare the training times which is the time taken for each algorithm to complete its training process among different agent.

In figure 3, we can notice that AC has the lowest training time among the algorithms. PPO Discrete has a much lower training time compared to PPO Continuous, but substantially higher than AC. This is due to the fact that, compared to AC, PPO Discrete incorporates a clipping mechanism during each policy update, which necessitates longer runtime. Furthermore, PPO Continuous deals with a continuous action space, which is inherently more complex, thereby requiring additional computation time. Finally, Double DQN has the highest training time, indicating it is the slowest algorithm among those tested. Its bar is the tallest, reaching close to 16,000 seconds. After replaying the Double DQN algorithm, we noticed that the car had a tendency to spin in place on the track — it kept running, but without fulfilling the termination conditions. As a result, within a single episode, the car often lingered in what should

have been a concluded training phase, thus unnecessarily prolonging the training time.

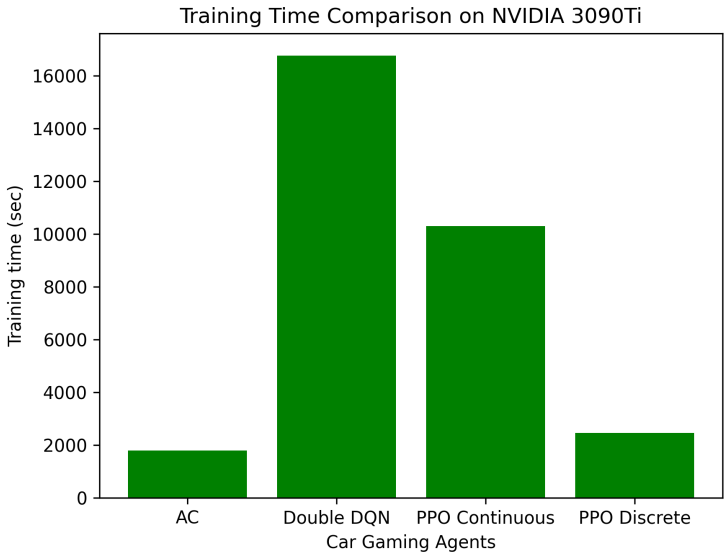

Figure 3: Training time comparison on NVIDIA 3090Ti. The y-axis represents the training time in seconds. The x-axis shows different algorithms used for training car gaming agents.

7.2 Average Reward Comparison

Second, we compared the average reward per episode during the training process, which reflected the performance of four different agents.

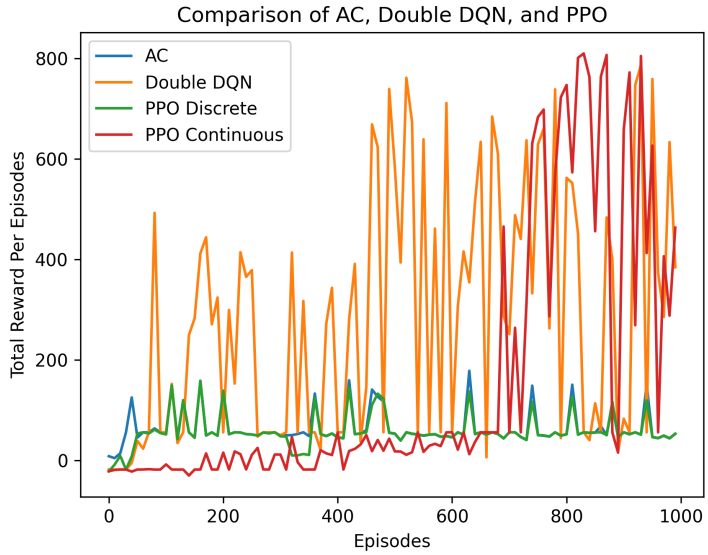

Figure 4: Reward comparison of AC, Double DQN, and PPO. The y-axis represents the total reward per episode. The x-axis shows the episodes, which are the number of attempts the agent has taken to learn the task. AC is represented by the blue line. Double DQN is represented by the orange line. PPO Discrete is represented by the green line. PPO Continuous is represented by the red line

In figure 4, we can observe that in the car racing environment, both PPO Discrete and AC struggled throughout the training process, seldom achieving rewards over 200. This indicates their difficulty in developing an effective strategy. It suggests that policy-based algorithms underperform when the action space is discrete. Additionally, these two algorithms are on-policy, requiring frequent model updates. In the early stages of training, there are many actions with low rewards, and the model does not specifically address these suboptimal actions, which hinders the on-policy algorithms from converging to a robust strategy.

As for DQN, it initially experienced a rapid increase in performance. This could be because Double DQN, by using the target network to estimate Q-values, avoids overly optimistic value estimations. It effectively decouples the action selection process from the generation of target Q-values, which is particularly beneficial for off-policy datasets. Eventually, it can achieve fairly good rewards.

PPO Continuous starts with low rewards as it's an off-policy learning method that does not necessarily update every round, so initially, it's just making random moves. After several hundred episodes, with only a few updates, its performance is poor. However, with enough training iterations, it can reach peak performance, making PPO Continuous the strongest in terms of capability among the four algorithms.

Both DQN and PPO Continuous show significant fluctuations after reaching a relatively good strategy, due to the limited adaptability of off-policy learning to new data. In the car racing environment, with three maps alternating, these two off-policy learning algorithms may have low adaptability to certain maps.

7.3 Game Performance Comparison

Finally, we compared the gaming performance of the four trained algorithms in car racing, with specific videos available at the link Open AI Gym - Car Racing Game DRL Agent.

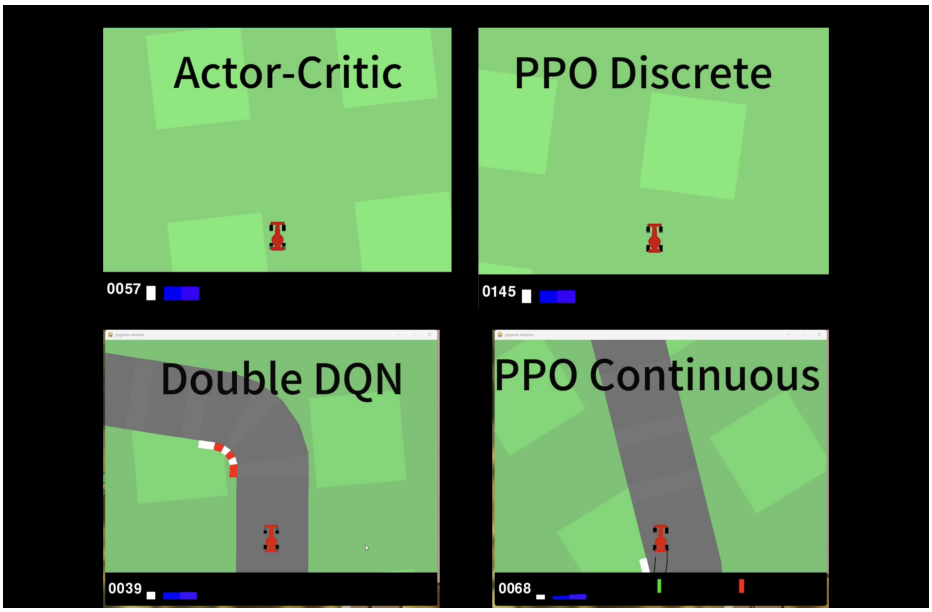

Figure 5: Game performance comparison on car racing. A screenshot of a comparison video showcasing gaming performance, where the video features four different panels, each displaying the gameplay of a different agent.

8 Discussion

Contributions: This study's primary contribution lies in the empirical analysis of the performance of various DRL algorithms within the CarRacing environment. Our results confirm that while Actor-Critic

and PPO Discrete algorithms show promise in discretized action spaces, they underperform relative to Double DQN and PPO Continuous. Specifically, our hypotheses regarding the efficiency of Double DQN in training time and the superior performance of PPO in continuous action spaces were supported.

Observations: Throughout the experimental process, we made several observations worth noting. For example, the tendency of Double DQN to allow the car to spin in place suggests a potential gap in the algorithm's ability to handle certain states, offering an avenue for improvement. Additionally, the fluctuating performance of off-policy algorithms across different maps indicates a need for better generalization within the learning process.

Future Work: Our study acknowledges limitations, particularly the algorithms' adaptability to new or unseen environments within the CarRacing game. The exploration of these DRL algorithms sets the stage for future research, including enhancing generalization and efficiency. The insights garnered have potential applications in autonomous system design and offer a stepping stone for more nuanced DRL implementations. Moving forward, extending our work to more complex scenarios and incorporating real-world dynamics could prove invaluable.

9 Conclusion

In conclusion, our investigation has provided a detailed comparative analysis of DRL algorithms in a simulated high-dimensional continuous control task. We have shown that PPO Continuous holds the most promise for such tasks, given sufficient training iterations. The implications of our work extend to the broader field of AI, offering insights that could influence future designs of autonomous systems and the development of more sophisticated DRL strategies. Ultimately, this study serves as a testament to the evolving landscape of DRL and its potential to transform complex task resolution in dynamic environments.

Appendix

Hyperparameters

General

Number of episodes	1000
# EXP buffer size (state transitions)	10000
# Minimal buffer size before learning	500
# Batch size	64
# Target network update frequency	10
Discount factor γ	0.98
Action Frame	8
State Frame	3
Random Seeds	0
Optimizer	Adam

Network Setup (All Algorithms)

Convolutional Layer 1	8 filters, 4x4 kernel, stride 2, ReLU activation
Convolutional Layer 2	16 filters, 3x3 kernel, stride 2, ReLU activation
Convolutional Layer 3	32 filters, 3x3 kernel, stride 2, ReLU activation
Convolutional Layer 4	64 filters, 3x3 kernel, stride 2, ReLU activation
Convolutional Layer 5	128 filters, 3x3 kernel, stride 1, ReLU activation
Convolutional Layer 6	256 filters, 3x3 kernel, stride 1, ReLU activation
Fully Connected Layer 1	Linear, from 256 to hidden dimension, ReLU activation
Output Layer:	
Action Branch	Softplus activation
Value Branch	Softplus activation

Discretized Action Space On-Policy AC

Actor learning rate	0.001
Critic learning rate	0.001

Discretized Action Space Off-Policy Double DQN

Learning rate	0.001
Epsilon ϵ	0.1

PPO, both versions

Critic learning rate	0.001
Actor learning rate	0.001
Entropy factor β	0.02
Epsilon ϵ	0.2
# Target network update frequency	10

References

[1] Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., & Blundell, C. (2020). Agent57: Outperforming the Atari Human Benchmark. *Proceedings of the 37th International Conference on Machine Learning*, 507–517.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning (arXiv:1312.5602). *arXiv*. `http://arxiv.org/abs/1312.5602`

[3] Dabney, W., Ostrovski, G., Silver, D., & Munos, R. (2018). Implicit Quantile Networks for Distributional Reinforcement Learning (arXiv:1806.06923; Version 1). *arXiv*. `http://arxiv.org/abs/1806.06923`

[4] Wurman, P. R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T. J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., Gilpin, L., Khandelwal, P., Lin, H., MacAlpine, P., Oller, D., Seno, T., Sherstan, C., Thomure, M. D., … Kitano, H. (2022). Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896), Article 7896. `https://doi.org/10.1038/s41586-021-04357-7`

[5] Holubar, M. S., & Wiering, M. A. (2020). Continuous-action Reinforcement Learning for Playing Racing Games: Comparing SPG to PPO (arXiv:2001.05270; Version 1). *arXiv*. `http://arxiv.org/abs/2001.05270`

[6] Jaritz, de Charette, R., Toromanoff, M., Perot, E., & Nashashibi, F. (2018). End-to-End Race Driving with Deep Reinforcement Learning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2070–2075. `https://doi.org/10.1109/ICRA.2018.8460934`

[7] Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, 12.

[8] Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12.

[9] Hasselt, H. (2010). Double Q-learning. *Advances in neural information processing systems*, 23.

[10] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. *In Proceedings of the AAAI conference on artificial intelligence* (Vol. 30, No. 1).

[11] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. *In International conference on machine learning* (pp. 1995-2003). PMLR.

[12] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015, Feb). Trust Region Policy Optimization. *arXiv:1502.05477*.

[13] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, Jul). Proximal Policy Optimization Algorithms. *arXiv:1707.06347*