Investigation of Reinforcement Learning Methods for Raceline Optimization (Attempt)

Joseph Hadidjojo, Tung Pham

December 19, 2024

Abstract

This work initially attempts to investigate the application of reinforcement learning (RL) to optimize raceline trajectories. However, due to unexpected complications in the environment setup, the RL agents were tested on the CartPole environment instead. The performance of Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN) were compared. DQN exhibited rapid early learning but suffered from long-term instability due to overestimation bias. In contrast, DDQN demonstrated smoother learning, sustained optimal performance, and improved stability. These results validate the theoretical advantages of DDQN and highlight its suitability for solving complex problems like raceline optimization. However, further work is needed to develop a representative raceline optimization environment to fully evaluate and conclusively confirm its applicability.

1 Introduction

Formula 1 (F1) racing is renowned for the intense precision, skill, and strategy that its drivers bring to the track. In each race, drivers must determine the fastest path around the circuit, or the "racing line," which enables them to maintain optimal speed while negotiating corners and straights. This process is not merely a physical feat; it requires an in-depth understanding of the vehicle's capabilities and a sharp sense of timing. Talented drivers often master the racing line in only a few laps of practice, fine-tuning it to suit their unique driving style and the distinct characteristics of their machinery [1].

Inspired by this mastery, the objective of this work is to explore the potential of reinforcement learning (RL) to approximate an optimal racing line on a simplified racing track. The core idea is to use RL, where an agent learns through experience, to find the fastest path around a track by optimizing two key actions: acceleration and rotation. By framing the problem in this way, we aim to develop a model that could eventually serve as a foundational tool for amateur racers with limited experience.

However, this problem presents unique challenges compared to more conventional RL problems:

- **High-dimensional state and action spaces**: Unlike simpler tasks with discrete or limited action spaces, racing involves continuous control of speed, position, and rotation, all of which must be adjusted dynamically to adapt to the track layout.
- **Balancing competing objectives**: Achieving the optimal racing line requires the agent to balance maximizing speed with avoiding crashes, spinouts, and leaving the track boundaries. High speeds, especially around tight corners, increase the risk of the agent veering off course, demanding precise control over acceleration and rotation to maintain the ideal path within the track's limits.
- **Delayed rewards**: The consequences of an action, like setting up for a corner, may not be immediately evident. The agent must learn to associate actions taken several turns ago with the eventual outcome, making the learning process more complex.
- **Track variability and precision demands**: Unlike simpler environments, the race track's varying curvature and the need for fine-grained control make it harder for the agent to generalize its learned strategy across different segments of the track.

These factors contribute to making the problem of finding an optimal race line particularly challenging, setting it apart from simpler RL applications.

While F1 and professional racing teams likely already utilize advanced tools and systems to estimate the optimal racing line, these applications are typically proprietary and remain restricted to private use. Due to the competitive nature of the sport, insights and strategies derived from such systems are rarely disclosed to the public. Consequently, amateur racers and enthusiasts have limited access to data-driven methods for learning and improving their racing lines. By exploring this problem with reinforcement learning, we aim to create a foundation for a more accessible tool that can help individuals develop their skills and gain insights into racing line optimization without requiring the high costs or exclusivity of professional-grade systems/academies. This work not only contributes to the field of RL in complex control tasks but also has the potential to democratize access to advanced racing analytics for a broader audience.

2 Background and Related Works

Reinforcement learning (RL) has seen significant advancements in recent years, especially in applications requiring continuous control and dynamic decision-making, such as autonomous driving and racing. In racing, RL agents must optimize not only for speed but also for stability, as they navigate complex tracks with tight turns and variable conditions. This section reviews notable works in RL and autonomous driving that have informed the development of driving and racing agents, highlighting the techniques and challenges relevant to this project.

2.1 Deep Q-Network (DQN)

A foundational work in deep reinforcement learning is the introduction of the Deep Q-Network (DQN) by Mnih et al. (2015) [2], which achieved human-level performance on Atari games. Although the DQN algorithm primarily focuses on discrete action spaces, it inspired adaptations for continuous and complex environments, such as autonomous driving and racing, where agents must navigate through tracks while balancing speed and control. This approach laid the groundwork for subsequent RL algorithms that could handle the continuous control demands of racing tasks.

2.2 Double Deep Q-Network (DDQN)

Double Deep Q-Networks (DDQN), proposed by van Hasselt et al. (2016) [3], addressed the overestimation bias in DQN by decoupling action selection and action evaluation. In DDQN, the next action is selected using the current Q-network, while its value is evaluated using the target network, resulting in more accurate target Q-values. This improvement enhances stability and policy performance, particularly in environments with high-dimensional state spaces and delayed rewards. DDQN's robustness makes it especially relevant to racing tasks, where balancing speed and control is critical.

2.3 Learning to Drive in a Day

Kendall et al. (2018) demonstrated the application of deep reinforcement learning to autonomous driving by training a model to perform lane following in a synthetic environment, followed by transferring the learned policy to a real-life driving environment [4]. Their approach utilized a model-free reinforcement learning algorithm, with all exploration and optimization conducted directly on the vehicle. This work highlighted the potential of end-to-end learning systems in autonomous driving, moving away from reliance on predefined rules and maps.

By drawing on these established techniques, this project seeks to develop an RL-based racing agent capable of navigating a simplified F1 track while balancing speed and stability, making it accessible as a foundational tool for aspiring racers.

3 Technical Approach / Methodology / Theoretical Framework

3.1 Problem Formulation

We envision our agent to operate within a discrete racing track environment represented as a 2D space. At each time step t, the agent observes the current state s_t, which includes:

- Car Position on the track (x, y),
- Car Velocity components (v_x, v_y),
- Car Orientation (θ),
- Car Angular momentum (ω),
- Car's Absolute distance to Left and Right track limits

The agent selects an action $a_t = [a_{\text{acc}}, a_{\text{rot}}]$, where:

- a_{acc} controls acceleration, constant (do nothing) or deceleration ($a \in \{-1, 0, 1\}$),
- a_{rot} controls the steering angle of the car from full-left, neutral or full-right ($\Delta\theta \in \{-1, 0, 1\}$)

The environment updates the agent's state based on these actions and the track's physics, such as track boundaries. Most real-life environmental factors such as friction, elevation, and grip condition will be ignored. The agent receives a reward r at the end of each lap, formulated as

$$r = -T + \alpha\Delta v_{max} - \beta O$$

where T is the lap time, Δv_{max} is the maximum speed change over a run, O is the number of track excursions, and α, β are tunable weight parameters. Through this, the agent learns to balance the objectives of speed and control (reducing spin-outs and track excursions), gradually improving its policy over training episodes.

3.1.1 Major Roadblock

Initially, we had selected the Car Racing v2 environment from Gymnasium v1.0 as the prime candidate for our agent's testing environment. However, we discovered that there was a bug in the discrete version of the environment that crashed the environment whenever it was initialized. Committing to this environment would result in an extensive effort to debug and develop a fix, in addition to adapting the environment to our current problem formulation. This resulted in a severe underestimation of the amount of work required to set up our environment as desired.

Due to the time and resource constraints, and upon consultation with the course instructor Yash Shukla, a decision was made to instead test our algorithm implementation on an alternative environment. To this end, Gymnasium's Cart Pole environment was selected due to its similar state-space representation and underlying physics formulation as our envisioned Problem Formulation.

3.1.2 Gymnasium's CartPole Environment

In this environment, the agent operates within a discrete space where at each time step t, the agent observes the current state s_t, which includes the followings:

- Cart Position (p) on the x-axis,
- Cart Velocity (v),
- Pole Angle (θ),
- Pole angle velocity (ω)

The actions that the agent can take at any given time t is to push the cart with a fixed amount of force in either the left or right direction ($a \in \{0, 1\}$).

The environment will then perform the underlying physics calculation needed (such as the velocity impact based on the angle of the pole) and update the state based on the action taken. Most real-life environmental factors, such as friction, are ignored. The agent receives 1 unit of reward for every timestep the pole is kept upright.

3.2 Methodology

To optimize the agent's behavior, we leverage two reinforcement learning techniques: Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN). Each method builds upon fundamental principles of RL, such as the Bellman equation and policy optimization.

3.2.1 Q-Learning and Deep Q-Learning

Q-learning [5] is a value-based reinforcement learning algorithm that seeks to learn the optimal action-value function $Q^*(s, a)$ using the Bellman equation:

$$Q^*(s, a) = r + \gamma \max_a Q^*(s', a),$$

where s' is the next state, and γ is the discount factor.

The algorithm iteratively updates the Q-values using [6]:

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

where α is the learning rate. This simple, table-based approach works well in discrete environments with small state and action spaces. However, Q-learning has significant limitations when applied to complex environments like racing tracks:

- **High-dimensional state spaces**: Representing continuous states (e.g., positions, velocities, and orientations) as discrete states leads to a combinatorial explosion in the size of the Q-table.

- **Inefficient learning**: In large state-action spaces, learning becomes computationally expensive, and generalization across similar states becomes practically impossible.
- **Overestimation bias**: During training, the agent is prone to picking up noises in the environment which causes the estimated value of some actions to be inflated and propagated, leading to overestimation.

These limitations necessitate scalable approaches like **Deep Q-Networks (DQN)** [2] for handling high-dimensional state spaces. DQN extends Q-learning by approximating the Q-function $Q(s, a)$ with a deep neural network, enabling it to handle large, high-dimensional state spaces. Instead of maintaining a Q-table, DQN uses a neural network parameterized by θ to predict $Q(s, a)$ for each possible action. This makes DQN effective for high-dimensional problems, though it remains limited to discrete action spaces.

A key feature introduced by DQN is the **Experience Replay** memory, a mechanism where the agent stores its past experiences (s, a, r, s') in a replay buffer and samples mini-batches of these experiences randomly to train the network. The Experience Replay is a necessary by-product of leveraging deep neural networks, due to their requirements of independent and identically distributed (i.i.d.) data to be trained effectively (which is not naturally the case for many RL methods, such as the Q-table).

To train the Q-network, DQN minimizes the temporal difference (TD) loss:

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

where:

- θ: Represents the parameters (weights) of the current Q-network used to approximate $Q(s, a)$.
- θ^-: Represents the parameters of the target Q-network, which are periodically updated to match θ. The use of θ^- ensures stability by decoupling the target computation from the current network's rapidly changing parameters.

In this work, we leverage the ML framework PyTorch to construct the Deep Q-Network. A 3 fully connected layers network with ReLu activation method was utilized to leverage a simplistic design. Additionally, we incorporate a separate but identical network with delayed weight updates in order to stabilize the training of the Q-Network [7].

While DQN is effective for high-dimensional discrete action problems, it still suffers from the same issue of overestimation bias as Q-Learning. This occurs because DQN uses the maximum estimated Q-value $\max_a Q(s', a)$ of the next state s' for updating the Q-value of the current state-action pair $Q(s, a)$. If the estimated Q-values are noisy, this approach can systematically overestimate the true values, leading to suboptimal policies and slower convergence. To overcome this issue, Double Deep Q-Network is utilized and will be discussed in further detail in the next section.

3.2.2 Double Q-Learning and Double Deep Q-Learning

Double Q-Learning [8] was introduced as a method to combat the overestimation bias problem in Q-Learning. It does this by maintaining an additional Q-table estimates and alternates between using one to select the best action and the other to evaluate it. The following update algorithm allows concurrent learning of different estimates simultaneously [6]:

$$Q_i(S_t, A_t) \leftarrow Q_i(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_j(S_{t+1}, \arg\max_a Q_i(S_{t+1}, a)) - Q_i(S_t, A_t)]$$

In the event that one Q-table is overestimating, it is extremely unlikely for the other Q-table to do the same. Therefore, by using one table to update the other, an unbiased estimate can be produced which avoids overestimation.

This approach works well with small and simple environments with limited state and action space. However, for bigger problems, it faces the same issues as other tabular-based approaches like Q-Learning, as was discussed in §3.2.1.

To combine Double Q-Learning's capabilities in mitigating overestimation bias, and the scalability of Deep Q-Networks, the **Double Deep Q-Networks (DDQN)** [3] was introduced. DDQN extends Double Q-Learning by replacing both Q-tables with deep neural networks (commonly referred to as the "online"(θ) and "target" (θ^-) network), enabling it to handle much larger and higher-dimensional state spaces. This also improves the original DQN implementation by decoupling the action selection and action evaluation processes, which combined results in a scalable method with minimal overestimation bias.

In finer detail: Instead of directly using the maximum Q-value from a single network for the next state s', DDQN splits the computation into two parts:

1. **Action Selection:** The action a^* for the next state s' is chosen using the current (online) Q-network θ:

$$a^* = \arg\max_a Q(s', a; \theta).$$

2. **Action Evaluation:** The value of this action $Q(s', a^*; \theta^-)$ is then estimated using the target network:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a^*; \theta^-) - Q(s, a) \right].$$

By separating the networks used for action selection and evaluation, DDQN avoids the optimistic bias inherent in using a single network for both, thus yielding more accurate Q-value estimates. The reduced bias in Q-value estimates leads to more stable training and faster convergence, particularly in environments with high-dimensional state and action spaces, such as racing tracks. These more accurate Q-value estimates also enable the agent to select more optimal actions, improving overall performance.

In terms of implementation, we once again leverage the PyTorch ML framework and followed the same neural network architectures as our DQN implementation for both of DDQN's Deep Q-Networks (please refer back to §3.2.1 and the attached codebase for further details).

3.2.3 Hyperparameter Search

For both agents, we utilized Python's Ray Tuning package to run a hyperparameter search on two separate machines: an x64-based machine with 10 concurrent threads and 1 RTX 3070 GPU with CUDA 12.4 accelerator, and an ARM64-based MPS accelerated computer with 10 concurrent threads. The hyperparameter search space is defined in Table 1. The number of episodes was

Hyperparameter	**Search Space**
Replay Buffer Memory	{5000, 10000, 20000}
Batch Size	{64, 128, 256}
Starting ϵ	Uniformly selected from [0.9, 1.0]
Minimum ϵ	Uniformly selected from [0.01, 0.1]
ϵ Decay Rate	{500, 1000, 2000}
τ	Uniformly selected from $\log_{10}(1e^{-3})$ to $\log_{10}(1e^{-2})$
γ	Uniformly selected from [0.95, 0.99]
Learning Rate	Uniformly selected from $\log_{10}(1e^{-4})$ to $\log_{10}(1e^{-3})$

Table 1: Hyperparameter Search Space. Here τ is the soft update parameter used to control how quickly the target network's weights are updated to match the main network's weights, γ is the discounted factor, and ϵ is the exploration-exploitation rate used in ϵ-greedy action selection method.

kept fixed at 1000 and we formulate our hyperparameter search objective to be the average reward over the last-100 episodes, as well as an additional stability score measured through tracking the minimum reward acquired by the agent over the same period.

4 Experimental Results / Technical Demonstration

4.1 Hyperparameter Search Result

The resulting hyperparameter configurations from the search are tabulated in Table 2. Both configurations achieved 500 average rewards convergence and a stability score of 500 over the last-100 episodes.

4.2 Performance of DQN and DDQN on the CartPole Environment

The performance of Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) agents on the CartPole environment was evaluated over 1000 episodes by tracking the total rewards achieved. The reward corresponds to the duration

Parameter	**DQN**	**DDQN**
Replay Buffer Memory	5000	20000
Batch Size	256	64
Starting ϵ	0.9548	0.9551
Minimum ϵ	0.03078	0.002246
ϵ Decay Rate	2000	3500
τ	0.003826	0.004221
γ	0.9728	0.97405
Learning Rate	0.0001785	0.0003433

Table 2: Hyperparameter Search Results for DQN and DDQN

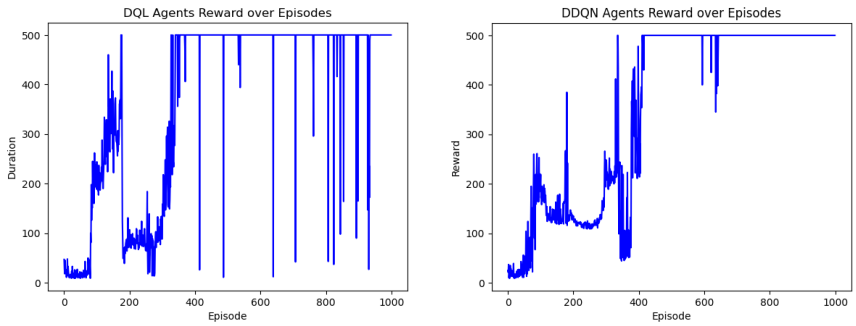

Figure 1: Rewards plot over 1000 episodes: DQN (left) and DDQN (right)

for which the agent successfully balances the pole without failure.[1]

The DQN agent initially demonstrates a rapid increase in performance, achieving near-optimal rewards by approximately episode 300. However, as training progresses, significant instability emerges. After reaching the maximum reward of 500, the agent frequently experiences sharp drops in performance, where rewards occasionally fall to near-zero values. This inconsistency persists throughout the later episodes, particularly beyond episode 500, highlighting DQN's struggle to maintain stable and reliable performance. This behavior is attributed to the overestimation bias limitation as discussed in §3.2.1. The inflated Q-value estimates introduce instability causing the abrupt fluctuations in rewards.

In contrast, the DDQN agent shows a smoother and more gradual learning curve. While it takes slightly longer to achieve high rewards compared to DQN, the DDQN agent steadily improves its performance, converging to the maximum reward of 500 by approximately episode 400. Notably, once this optimal performance is reached, the DDQN agent maintains it with remarkable consistency throughout the remaining episodes, exhibiting only minor fluctuations. This

[1]Recorded demonstrations of each method are available at: `https://drive.google.com/drive/folders/1zNHzl44QasyTJddteCR-xB4u2pGK30rb?usp=sharing`

improved stability of DDQN can be attributed to its ability to mitigate overestimation bias, as was discussed in §3.2.2. By decoupling action selection from value evaluation, DDQN ensures more accurate Q-value estimates, allowing the agent to learn a robust and reliable policy.

5 Conclusion and Future Work

In this work, we initially set out to explore solving the raceline optimization problem using reinforcement learning (RL). However, due to unexpected complications in the candidate environment, we were unable to explicitly implement this solution. Instead, we redirected our focus to testing our implementation on the CartPole environment, which was chosen for its similar state representations and underlying physics. This shift allowed us to validate our implementations and compare the performance of DQN and DDQN agents in a somewhat representative environment.

While we were not able to fully test our agents on a fully-formulated environment as was intended, our test results on the CartPole environment demonstrated that while DQN exhibited rapid early learning, it struggled with long-term stability due to overestimation bias, which led to frequent fluctuations in rewards and inconsistent performance. DDQN, on the other hand, provided a smoother and more gradual learning process. By decoupling the action selection and evaluation steps, DDQN effectively reduced overestimation bias, resulting in more accurate Q-value estimates. As a result, the DDQN agent achieved and sustained optimal performance, exhibiting far fewer fluctuations compared to DQN. These findings align with the theoretical advantages of DDQN and highlight its robustness for environments requiring consistent and stable policies.

Looking ahead, our top priority is to develop and implement a fully functioning and representative environment of our raceline optimization problem. This environment would allow us to properly test our agents on a setup that mirrors the complexities of real-world raceline optimization. Furthermore, we would like to further extend this and explore applying transfer learning techniques to deploy agents trained in this simulated environment to a real-life setup, such as the F1Tenth environment [9].[2] By leveraging transfer learning, we aim to adapt the knowledge gained in simulation to physical systems and possibly bridge the gap between theoretical RL solutions and practical applications by eventually integrating this into driver-in-loop simulators for the benefit of new and young race drivers of the future.

[2]The F1Tenth environment is a platform for autonomous racing that uses scaled-down F1 race cars equipped with sensors and computational hardware to navigate real-world tracks.

References

[1] Adam Brouillard. *The Perfect Corner: A Driver's Step-By-Step Guide to Finding Their Own Optimal Line Through the Physics of Racing*. Paradigm Shift Motorsport Books, Paradigm Shift Driver Development, March 2016.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[4] Alex Kendall, Jamie Hawke, David Janz, Pawel Mazur, Daniele Reda, John-Paul Allen, Victor-Daniel Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *arXiv preprint arXiv:1807.00412*, 2018.

[5] Christopher JCH Watkins. *Learning from Delayed Rewards*. Phd thesis, King's College, Cambridge, 1989.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press Cambridge, 2018.

[7] Jonathan Hui. Reinforcement learning explained visually - part 5: Deep q-networks step by step, 2018. Accessed: 2024-12-18.

[8] Hado van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, volume 23, 2010.

[9] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In *NeurIPS 2019 Competition and Demonstration Track*, pages 77–89. PMLR, 2020.