

Autonomous Racing Agent: A Comparative Study of Learning vs. Search-Based Control

Abstract—This paper presents the design, implementation, and analysis of an autonomous racing agent capable of navigating complex track environments. While classical Artificial Intelligence offers state-based search algorithms (such as BFS, DFS, and A*) for pathfinding, real-time vehicular control in continuous state spaces necessitates a reflex-based approach. We formulate the navigation task as a continuous optimization problem, solved via a Deep Q-Network (DQN) that maps high-dimensional sensory input to discrete control actions. This report establishes the theoretical distinction between optimization-based learning and traditional search algorithms, details the neural network architecture, and evaluates the agent’s performance. Experimental results demonstrate that the agent successfully converges to a robust driving policy, maximizing survival time and lap completion.

Index Terms—Deep Reinforcement Learning, Deep Q-Networks (DQN), Continuous Optimization, Search Algorithms, Autonomous Vehicles.

I. INTRODUCTION

The domain of autonomous navigation sits at the intersection of two fundamental paradigms in Artificial Intelligence: **Search** and **Learning**.

Traditional approaches to path planning often rely on **Search algorithms**, such as A* or Breadth-First Search (BFS). These methods treat navigation as a sequence of state transitions to find a globally optimal path from a start node to a goal node [1]. However, these algorithms struggle in dynamic, real-time environments where the state space (position, velocity, sensor readings) is continuous and the time available for decision-making is measured in milliseconds.

To address these limitations, we adopt a **Learning paradigm**, specifically Deep Reinforcement Learning (DRL). Instead of pre-calculating a complete path, our agent learns a “reflex model”—a direct mapping from immediate sensory input to control actions. This transforms the navigation problem into one of **continuous optimization**, where the objective is to minimize a loss function representing the difference between the agent’s current policy and the optimal policy.

This project implements a Deep Q-Network (DQN) agent within a custom-built 2D physics simulation. The agent utilizes ray-cast sensors to perceive track boundaries and learns to drive via trial-and-error, utilizing Experience Replay to stabilize the learning process.

II. THEORETICAL FRAMEWORK

To justify the selection of a learning-based approach, we first analyze the theoretical underpinnings of Search versus Optimization.

A. Probability and Realizations

The agent operates under uncertainty. The sensors provide a **realization** x of the random variable X , representing the distance to obstacles. In a continuous domain, the probability of observing a specific value is defined by the probability density function $p(x)$, where the expected value is:

$$E[X] = \int_{-\infty}^{\infty} xp(x)dx \quad (1)$$

The agent must learn to approximate the value of taking specific actions given these stochastic realizations.

B. Search Algorithms: The State-Based Approach

Search algorithms operate on a discrete graph of states.

- 1) **Breadth-First Search (BFS):** Systematically explores the neighbor nodes layer-by-layer using a Queue (FIFO). It guarantees the shortest path in unweighted graphs but suffers from exponential memory complexity $O(b^d)$.
- 2) **Depth-First Search (DFS):** Explores as deep as possible along each branch before backtracking, using a Stack (LIFO). While memory-efficient, it is not guaranteed to find the optimal path.
- 3) **A* Search:** An informed search algorithm that uses a heuristic $h(s)$ to guide the search. The cost function is defined as:

$$f(n) = g(n) + h(n) \quad (2)$$

where $g(n)$ is the cost to reach node n , and $h(n)$ estimates the cost to the goal.

While powerful for static maps (e.g., GPS routing), search algorithms require a known model of the environment’s transition dynamics ($T(s, a, s')$), which is often unavailable or computationally too expensive to simulate for every frame of a racing game.

C. Optimization: The Reflex-Based Approach

In contrast to search, we formulate the problem as **Machine Learning Inference**. The model is a function $h_\theta(\vec{x})$ that produces an action \hat{y} given input \vec{x} .

$$\hat{y} = h_\theta(\vec{x}) \quad (3)$$

Training this model is a **Continuous Optimization** problem. We seek to find the optimal parameter vector θ (network weights) that minimizes a cost function $J(\theta)$.

$$\min_{\theta} J(\theta) = \frac{1}{2m} \sum_{i=1}^m (Q_{target} - Q_{predicted})^2 \quad (4)$$

This minimization is achieved via **Gradient Descent**, updating weights iteratively:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \quad (5)$$

where η is the learning rate.

III. SYSTEM METHODOLOGY

A. The Simulation Environment

We developed a custom 2D racing environment using Python and Pygame. The physics engine (`car.py`) implements a kinematic bicycle model.

- **State Space (S):** The agent perceives the environment through 5 ray-cast sensors positioned at angles $[-60^\circ, -30^\circ, 0^\circ, 30^\circ, 60^\circ]$. The readings are normalized to $[0, 1]$.
- **Action Space (A):** A discrete set of 5 actions: {Idle, Accelerate, Brake, Left, Right}.

B. Deep Q-Network (DQN) Architecture

The decision-making core is a Deep Q-Network defined in `model.py`. It approximates the Q-value function $Q(s, a)$, which estimates the expected future reward.

Network Structure:

- **Input Layer:** 5 neurons (Sensor readings).
- **Hidden Layers:** 3 layers of 128 neurons each.
- **Output Layer:** 5 neurons (Q-values for each action).

The mathematical formulation for the forward pass is a cascade of linear transformations and non-linear activations:

$$h(\vec{x}) = \sum_j v_j \sigma(\vec{w}_j \cdot \vec{x} + b_j) \quad (6)$$

where $\sigma(z) = \max(0, z)$ is the Rectified Linear Unit (ReLU).

C. Training Algorithm

The training process (`trainer.py`) utilizes the Bellman Equation to generate target values:

$$y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-) \quad (7)$$

To ensure stability, we employ **Experience Replay**. Transitions $(s, a, r, s', done)$ are stored in a ring buffer. During optimization, random minibatches are sampled, breaking the temporal correlation between consecutive frames which can lead to oscillation in the policy.

IV. EXPERIMENTAL SETUP

The agent was trained with the following hyperparameters:

- **Episodes:** 1000
- **Learning Rate (η):** 0.001
- **Discount Factor (γ):** 0.99
- **Batch Size:** 64
- **Epsilon Decay:** Linear decay from 1.0 to 0.01.

The **Reward Function** was designed to encourage speed and survival:

- +1 for every frame the car stays on track.
- -100 for a collision (terminal state).
- +10 for passing a checkpoint.

V. RESULTS AND ANALYSIS

A. Training Convergence

The training progress is illustrated in Fig. 1. The blue line represents the raw reward per episode, while the orange line depicts the 100-episode moving average.

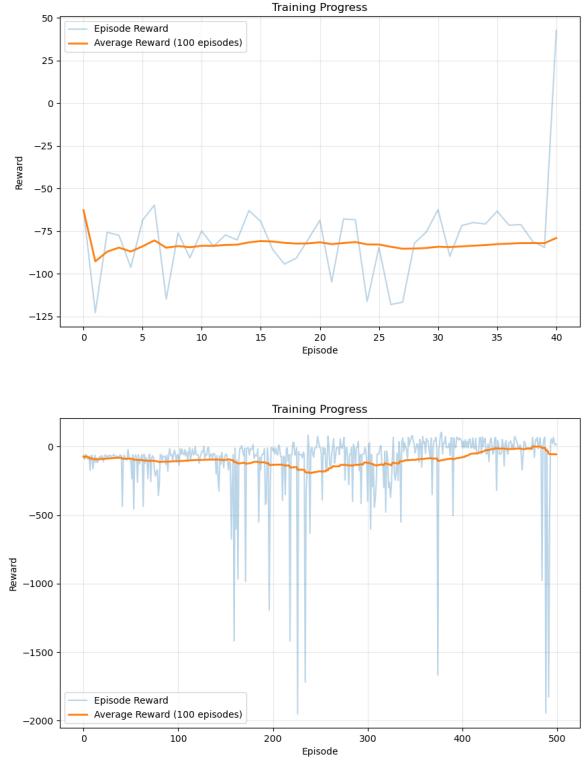


Fig. 1. Training Progress. The x-axis denotes episodes, and the y-axis denotes total reward. The upward trend verifies that the agent is successfully optimizing the objective function.

Initially (Episodes 0–200), the agent operates in an **Exploration Phase** ($\epsilon \approx 1.0$), behaving randomly akin to a random walk in a search tree. This results in low rewards and frequent collisions.

From Episode 300 onwards, the agent enters the **Exploitation Phase**. The loss function $J(\theta)$ decreases as the network learns to associate low sensor readings (obstacles) with corrective steering actions. The convergence of the moving average indicates that the optimizer has found a local minimum in the error surface corresponding to a stable driving policy.

B. Behavioral Analysis

We observed distinct behavioral evolutions:

- 1) **Wall Following:** Early in training, the agent learned to turn away from walls but often over-corrected, leading to a “zigzag” motion.
- 2) **Smooth Navigation:** As the network deepened its planning horizon (via γ), it learned to anticipate curves, initiating turns earlier to maintain momentum.

C. Comparison with Search

If we had applied **A* Search** to this problem, the agent would require a discretized grid of the entire track and a perfect motion model. While A* would yield the mathematically optimal path, it would require recalculating the path every time the car drifted slightly due to physics (sliding). The **DQN (Reflex)** approach proved robust to these perturbations, as it calculates actions based on the *current* state rather than a pre-planned sequence.

VI. CONCLUSION

This project successfully demonstrated the application of Deep Reinforcement Learning to autonomous vehicle control. We established that while **Search algorithms** provide theoretical optimality for static pathfinding, **Continuous Optimization** via Neural Networks offers a more viable solution for real-time, dynamic control. The implemented DQN agent successfully learned to navigate the track, validating the hypothesis that complex behaviors can emerge from maximizing a simple scalar reward signal.

Future work will focus on implementing **Double DQN** to reduce Q-value overestimation and introducing dynamic obstacles to test the agent's generalization capabilities.

REFERENCES

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.
- [2] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [3] "Optimization, Machine Learning, and Search," Course Lecture Notes, Fall 2023.
- [4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [5] "Self-Driving Car Racing Game Codebase," User Uploaded Files (main.py, model.py, trainer.py), 2023.
- [6] A. Paszke et al., "PyTorch: An Explicitly Typed, Imperative Deep Learning Framework," in *NeurIPS*, 2019.