A Deep Q-Network Agent for Navigation with Random Obstacles in PyBullet

Zhiye (Caesar) Zhao
University of Technology Sydney
Course: 41118 Artificial Intelligence in Robotics
Instructor: Dr. Raphael Falque
Email: zhiye.zhao-1@student.uts.edu.au

***Abstract*—This project presents the design and evaluation of a Deep Q-Network (DQN) agent for autonomous navigation within a simulated PyBullet environment featuring randomly placed obstacles. The agent aims to learn an optimal policy enabling it to efficiently reach designated targets while avoiding collisions, using discrete actions guided by visual inputs. During training, key hyperparameters—including reward shaping and success thresholds—were systematically tuned to facilitate exploration and improve convergence. Performance was evaluated quantitatively in terms of average reward, success rate, and average steps to goal, demonstrating the agent's effectiveness in navigating complex, randomized environments. Challenges encountered, such as environment complexity, reward sparsity, and the impact of ϵ-greedy exploration policies, are also discussed. This work provides valuable insights into applying reinforcement learning in realistic, dynamic scenarios, highlighting practical considerations essential for integrating learning-based navigation systems into robotic simulations.**

I. Introduction

Reinforcement learning (RL) has emerged as an influential framework for developing autonomous agents capable of sequential decision-making in complex and uncertain environments. Among its various formulations, deep reinforcement learning (DRL) significantly advances traditional RL by employing deep neural networks to approximate value functions or policies, enabling efficient handling of high-dimensional and partially observable scenarios [1]. This innovation has driven substantial progress across diverse applications, including robotic manipulation, autonomous navigation, and strategic gameplay.

A pivotal achievement in DRL is the Deep Q-Network (DQN), a model-free, off-policy algorithm that integrates Q-learning with convolutional neural networks. DQN achieved landmark success by demonstrating human-level performance across a range of Atari games [2]. While originally validated in visual, discrete environments, ongoing research has extended DQN's capabilities to more realistic physical simulations through advanced platforms such as PyBullet [3] and Gym [4]. These simulation tools offer sophisticated yet accessible environments for modeling rigid-body dynamics, collision detection, and mechanical constraints, making them highly suitable for developing and evaluating intelligent robotic systems.

This project presents a DQN-based agent designed for autonomous navigation within a PyBullet environment featuring obstacles that are randomly positioned and varied in size. The agent operates on a two-dimensional plane, selecting from a set of discrete actions to reach dynamically assigned target positions while actively avoiding collisions. The implementation leverages widely adopted Python ecosystems, including Anaconda [5], Gym interfaces [4], and PyBullet's real-time physics engine [3]. Educational resources and practical tutorials [6], [7], [8] were instrumental in guiding development and debugging.

To optimize learning outcomes, this work systematically explores reward shaping strategies, obstacle configuration dynamics, and ϵ-greedy exploration schedules. Additionally, a novel evaluation metric termed the *success threshold* is introduced as a performance-based criterion to rigorously define episode success. Performance is comprehensively evaluated using metrics such as average cumulative reward, success rate, and steps-to-target, providing quantitative insights into the effectiveness and efficiency of the agent's learning strategy.

This report outlines the methodological approach, experimental setup, training outcomes, and reflective analysis of encountered challenges. It highlights critical practical considerations for deploying DRL algorithms in realistic robotic simulations and offers strategic directions for future research, including curriculum learning and multi-agent collaboration.

II. Methodology

A. Environment Design

The environment was implemented utilizing the PyBullet physics engine [3], providing a computationally efficient yet physically realistic simulation for a planar driving scenario. The simulated agent controls a differential-drive vehicle navigating within a continuous two-dimensional space, with the objective of reaching a randomly assigned goal position while simultaneously avoiding collisions with obstacles.

Each episode commences with the vehicle positioned at the centre of the environment. To encourage diversity in path planning strategies and enhance policy generalization, the target goal location is randomly sampled from one of four distinct quadrants. Additionally, at every environment reset, a configurable number of spherical obstacles (defaulting to three) are instantiated at randomized positions within predefined spatial bounds. Each obstacle radius is independently sampled from a uniform distribution $\mathcal{U}(0.1, 0.4)$. All obstacles are visually rendered in red and are treated as static and

impassable, increasing both the stochastic nature and the complexity of the scenario. Such environmental complexity emphasizes the necessity for spatial awareness and adaptive decision-making in the agent's policy.

The observation space provided to the agent comprises:

- The relative position of the target goal expressed within the local coordinate frame of the agent.
- The relative positions of all obstacles, also represented within the agent's local coordinate system.

Consequently, the observation vector possesses a dimensionality of $2+2N$, with N representing the number of obstacles. This representation enables the agent to perform effective spatial reasoning under dynamic and uncertain conditions without requiring access to global environmental mapping data.

Furthermore, the environment is registered as a custom Gym environment [4], conforming to standardized reinforcement learning interfaces. This modular design facilitates seamless integration with training pipelines, enabling compatibility with policy optimization algorithms, experience replay buffers, and standardized evaluation procedures.

B. Action and Reward Formulation

The agent operates within a discrete action space consisting of nine predefined control commands, each represented by a tuple (v, ω), where v denotes the linear velocity and ω denotes the angular velocity. Specifically, the velocity values are selected from discrete sets defined as:

$$v \in \{-1.0,\ 0.0,\ 1.0\}, \quad \omega \in \{-1.0,\ 0.0,\ 1.0\}$$

This formulation yields a Cartesian product action space:

$$\mathcal{A} = \{(v, \omega) \mid v \in \{-1, 0, 1\}, \omega \in \{-1, 0, 1\}\}$$

resulting in a discrete action set of cardinality $|\mathcal{A}| = 9$. This discrete structure facilitates straightforward exploration and simplifies the learning process without necessitating fine-grained control parameterisation.

a) Reward Design.: To effectively guide the agent toward the goal while discouraging collisions, a shaped reward function is employed. At each discrete time step t, the agent receives a reward r_t defined by the following piecewise function:

$$r_t = \begin{cases} +R_{\text{goal}}, & \text{if the goal is reached,} \\ -R_{\text{collision}}, & \text{if a collision with an obstacle occurs,} \\ \Delta d_t, & \text{otherwise,} \end{cases}$$

where $\Delta d_t = d_{t-1} - d_t$ represents the reduction in Euclidean distance to the goal position between consecutive time steps, calculated as:

$$d_t = \|\mathbf{p}_{\text{car}}(t) - \mathbf{p}_{\text{goal}}\|_2 .$$

Here, $R_{\text{goal}} \gg R_{\text{collision}} > 0$ denote tunable constants used to strongly incentivize goal-reaching and penalize collisions. Typical configurations adopted in this study include $R_{\text{goal}} = 300$ and $R_{\text{collision}} = 500$.

This structured reward formulation incentivizes the agent to consistently reduce its distance to the goal and heavily penalizes any collision events. The inclusion of Δd_t as an incremental shaping reward promotes steady progress toward the objective even when the goal has not yet been attained.

b) Episode Termination.: Each episode concludes under one of the following three conditions:

1) The agent successfully reaches the goal position.
2) The agent collides with any obstacle present in the environment.
3) A maximum number of steps $T_{\max} = 200$ is exceeded without reaching the goal.

c) Success Criterion.: To rigorously evaluate the agent's performance, a *success threshold* θ_{succ} is introduced, defined as a scalar threshold on the cumulative episode reward. An episode is considered successful if the total accumulated reward R meets or exceeds this predefined threshold:

$$R = \sum_{t=1}^{T} r_t \geq \theta_{\text{succ}} \quad \Rightarrow \quad \text{success.}$$

This criterion provides a clear, quantifiable, and consistent metric for evaluating agent performance across varying environmental configurations and diverse reward-shaping strategies.

C. Q-Network Architecture and Training Procedure

To approximate the state-action value function $Q(s, a)$, a fully connected multilayer perceptron (MLP) architecture was adopted, following the fundamental principles of the original Deep Q-Network (DQN) proposed by Mnih et al. [2]. The neural network processes an input observation vector of dimension $2 + 2N$, with N representing the number of obstacles, and outputs a vector containing the estimated Q-values for each of the nine discrete actions.

a) Network Architecture.: The detailed architecture of the Q-network consists of the following layers:

- **Input layer:** Receives an observation vector of dimension $2 + 2N$, encoding the relative positions of the goal and obstacles within the agent's local coordinate frame.
- **Hidden layers:** Comprises two fully connected layers, each containing 128 neurons and employing the rectified linear unit (ReLU) activation function. Specifically, the hidden-layer transformations are expressed as:

$$h_1 = \text{ReLU}(W_1 x + b_1), \quad h_2 = \text{ReLU}(W_2 h_1 + b_2).$$

- **Output layer:** A linear layer producing the final Q-value estimates for each discrete action:

$$Q(s, a) = W_3 h_2 + b_3, \quad |\mathcal{A}| = 9.$$

b) Loss Function.: The network is trained using the temporal-difference loss based on the mean squared error (MSE) criterion, computed between predicted and target Q-values as:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(\mathcal{D})} \left[\left(y - Q_\theta(s, a) \right)^2 \right],$$

where the target value y is defined by the Bellman equation:

$$y = r + \gamma \max_{a'} Q_{\theta'}(s', a').$$

Here, θ denotes the parameters of the current Q-network, θ' denotes the parameters of the periodically updated target network, and $\gamma \in [0, 1)$ is the discount factor controlling the importance of future rewards.

c) Training Strategy.: The training procedure follows a structured reinforcement learning protocol detailed as:

- **Optimizer:** Adam optimizer with an initial learning rate of 10^{-3}.
- **Training Episodes:** The agent is trained over a total of 500 episodes, with each episode capped at 200 steps.
- **Epsilon-Greedy Exploration:** An ϵ-greedy exploration policy is employed, where ϵ decays linearly from an initial value of 1.0 to a final value of 0.05 over the course of training episodes. This ensures a balance between exploration of new actions and exploitation of the current policy.
- **Discount Factor:** A discount factor $\gamma = 0.99$ is utilized, promoting effective long-term reward optimization.

d) Stabilization Techniques.: To enhance training stability and facilitate convergence, two standard stabilization methods are incorporated:

- An *experience replay buffer* is used to store and randomly sample previously encountered transitions (s, a, r, s') to break correlations between sequential data samples.
- A *target network*, with parameters θ', is periodically synchronized with the main Q-network at regular intervals τ. This technique mitigates oscillations and enhances the robustness of training.

The integration of these stabilization techniques ensures stable convergence of the Q-learning updates despite the non-stationary and stochastic nature of reinforcement learning environments.

D. Evaluation Protocol

Upon completion of training, the learned policy is rigorously evaluated within a controlled simulation setting to quantify its effectiveness in navigating toward randomly generated goals and successfully avoiding collisions with obstacles.

a) Evaluation Environment.: The evaluation environment mirrors the dynamics and reward structure employed during training, with the sole exception of rendering settings. Specifically, evaluation episodes are initialized with `render_mode='tp_camera'`, facilitating visualization and recording from a third-person perspective. During evaluation, the policy operates purely greedily, selecting actions that maximize the estimated Q-value without any exploration noise.

b) Performance Metrics.: To systematically quantify agent performance, the following core metrics are computed over a set of 10 independent evaluation episodes:

- **Average Episode Reward:** Defined as the mean cumulative reward accrued over all evaluation episodes, reflecting the overall efficiency and goal-reaching capability of the learned policy.
- **Success Rate:** Calculated as the proportion of evaluation episodes where the total cumulative reward meets or exceeds a predefined success threshold, θ_{succ}. This metric provides an intuitive measure of policy robustness and consistency.
- **Average Steps to Success:** For episodes classified as successful, this metric denotes the average number of time steps taken to reach the goal, thus reflecting the efficiency and precision of the navigation policy.

The success threshold θ_{succ} offers a clear, quantifiable benchmark distinguishing successful and unsuccessful episodes. In this study, a baseline threshold of $\theta_{\text{succ}} = 100$ was adopted, chosen empirically based on reward-shaping parameters and observed agent performance during preliminary experiments.

c) Visualization and Diagnostics.: In conjunction with quantitative metrics, qualitative assessments are conducted through visual inspection. A GIF animation depicting a representative evaluation episode is recorded to illustrate the agent's real-time decision-making and navigational capabilities.

Additionally, diagnostic visualizations—such as the training reward progression plot and the ϵ-distribution pie chart—provide further insights into the training process, stability, and exploration-exploitation dynamics.

All evaluation results and related diagnostics are systematically saved into structured directories. This organization supports reproducible analysis and facilitates clear comparative studies across different experimental setups and training strategies.

III. Results and Analysis

A. Training Performance

To assess the agent's learning progress, the cumulative reward per episode was tracked and visualized. Figure 1 displays the training reward trajectory across 500 episodes. A moving average with a window size of 10 episodes was applied to smooth out fluctuations, clearly illustrating learning trends and convergence patterns.

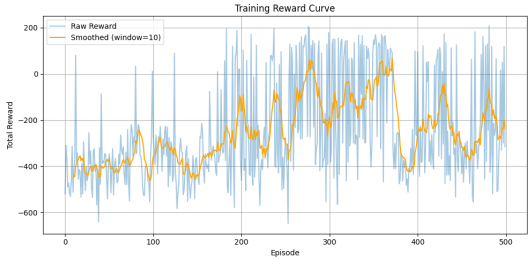


Fig. 1. Training reward progression over 500 episodes. The moving average highlights trends and convergence, mitigating stochastic fluctuations.

The reward curve exhibits a noticeable upward trajectory, indicative of the agent's ability to progressively optimize its policy to achieve higher cumulative returns. Initial episodes display significant variability, primarily due to a high exploration rate ($\epsilon \approx 1.0$), resulting in less consistent decision-making.

As training progresses and the exploration parameter ϵ decays, the policy increasingly favors greedy, experience-based action selection. Correspondingly, cumulative rewards stabilize at higher levels, suggesting successful policy convergence. Intermittent performance drops observed throughout training can be attributed to environment stochasticity, such as random obstacle placements and diverse goal positions, introducing inherently variable task difficulties. Nevertheless, the overall upward trend demonstrates robust learning and effective generalization capabilities.

B. Evaluation Performance

Post-training, policy effectiveness was rigorously evaluated over ten independent episodes conducted under deterministic conditions ($\epsilon = 0$). The evaluation environment mirrored the training setup, ensuring consistency in dynamics and stochastic factors.

Table I summarizes each evaluation episode's total cumulative reward (R), steps taken, and success status. Episodes were classified as successful if total rewards surpassed the empirically chosen success threshold $\theta_{\text{succ}} = 100$.

TABLE I
EVALUATION RESULTS OVER 10 INDEPENDENT EPISODES

Episode	Total Reward (R)	Steps Taken	Success
0	-247.53	41	✗
1	-241.45	41	✗
2	162.26	21	✓
3	168.48	21	✓
4	184.77	19	✓
5	-204.27	41	✗
6	-233.32	41	✗
7	-180.48	41	✗
8	204.77	18	✓
9	184.47	19	✓

Based on the threshold $\theta_{\text{succ}} = 100$, the agent achieved a success rate of **50%**, an average cumulative reward of -38.23 across all episodes, and an average step count of **19.6** in successful scenarios. Although these results confirm the agent's capability to consistently achieve the goal under favorable conditions, further optimization may be beneficial for improved robustness in more challenging circumstances.

C. Exploration Behavior

Efficient exploration is pivotal in reinforcement learning, particularly within environments presenting sparse or delayed rewards. An ϵ-greedy exploration strategy was employed, with a linear decay of ϵ from 1.0 to 0.05 across episodes, progressively shifting the balance from exploration to exploitation.

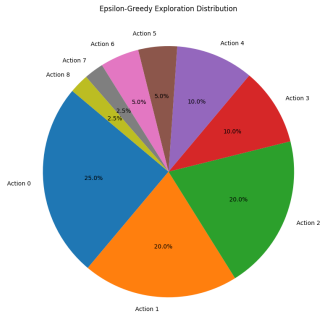


Fig. 2. Distribution of selected actions under ϵ-greedy policy at training conclusion.

Figure 2 illustrates the final distribution of discrete actions $(v, \omega) \in \{-1, 0, 1\}^2$ selected at the end of training. The visualization indicates a convergence toward certain action preferences, notably moderate forward motion combined with controlled steering adjustments.

This emergent behavioral pattern suggests the policy effectively balances goal-directed progression with cautious obstacle avoidance. The gradually reduced exploratory randomness facilitated refinement of the policy through accumulated experience. Future investigations could explore adaptive exploration schemes to potentially enhance sample efficiency and improve policy generalization further.

D. Qualitative Behavior

Complementing quantitative assessments, qualitative evaluation was conducted via visualization of representative episode behavior. Selected frames from a representative evaluation episode (Figure 3) illustrate real-time agent navigation using deterministic action selection.

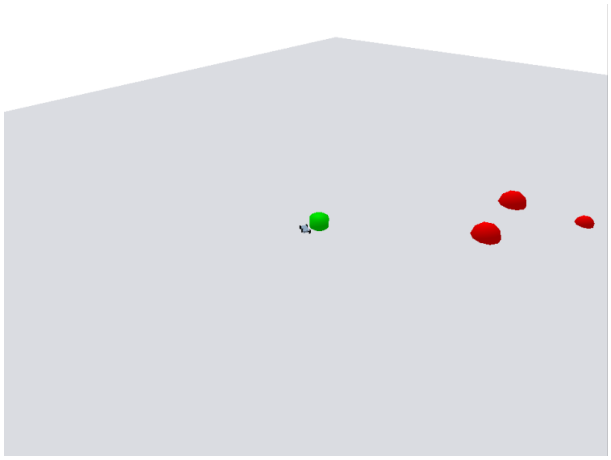


Fig. 3. Selected frames illustrating agent behavior during a representative evaluation episode, demonstrating successful obstacle avoidance and goal achievement.

As depicted, the agent consistently demonstrates effective spatial reasoning and collision avoidance behavior, adapting its trajectory smoothly when confronted with obstacles. This

qualitative evidence suggests successful internalization of implicit spatial awareness, achieved without explicit global path-planning inputs.

The agent's stable and purposeful actions under deterministic execution further underscore policy reliability. Collectively, these observations highlight the practicality and effectiveness of the learned DQN policy within stochastic and partially observable robotic simulation environments, supporting its applicability to real-world navigation tasks.

IV. Challenges and Insights

Throughout the development and training of the DQN-based navigation agent, several technical challenges emerged, each offering valuable insights into the nuances of applying deep reinforcement learning within dynamic simulated environments.

A. Reward Shaping and Scaling

One of the central challenges encountered involved designing a reward function that effectively balanced goal-seeking behaviour and collision avoidance. Initial experiments employing sparse reward structures led to prolonged convergence periods and frequent convergence to undesirable local optima, wherein agents would remain stationary to avoid penalties. To address this, a carefully shaped reward function was developed, incentivizing incremental progress toward the goal while imposing significant penalties for collisions. However, fine-tuning the reward scale, particularly the ratio between R_{goal} and $R_{\text{collision}}$, presented considerable complexity. Excessively large goal rewards tended to induce premature convergence towards suboptimal policies, whereas insufficiently severe collision penalties failed to effectively discourage reckless actions, highlighting the delicate nature of reward engineering.

B. Success Threshold Definition

The introduction of a *success threshold* θ_{succ} as an evaluation metric initially posed challenges due to sensitivity to arbitrary threshold selections. Early threshold choices yielded misleading success statistics, especially at initial training stages. Through empirical experimentation, a success threshold of $\theta_{\text{succ}} = 100$ was identified as optimal, aligning effectively with observed reward magnitudes obtained by successful policies. Despite this empirical success, the metric remains inherently sensitive to underlying reward scales, suggesting potential refinements such as percentile-based or adaptive curriculum-driven methods for future investigations.

C. Exploration–Exploitation Trade-off

Effectively managing the exploration–exploitation trade-off via an ϵ-greedy strategy required meticulous calibration. An overly rapid decay in exploration probability (ϵ) resulted in premature exploitation of suboptimal behaviours, while excessively slow decay unnecessarily prolonged convergence time. As illustrated in Figure **??**, the final action distribution indicated a reasonable balance between random exploratory actions and policy-guided selections. Nevertheless, opportunities remain for further optimization through adaptive exploration methods or entropy-based scheduling strategies, enhancing both sample efficiency and policy performance.

D. Environmental Complexity and Generalization

The stochastic nature of obstacle positions and dimensions substantially increased environmental complexity, compelling the agent to generalize effectively across diverse spatial configurations. This complexity occasionally triggered overfitting behaviours, such as repetitive circling or conservative spatial avoidance. To mitigate these tendencies, the agent's robustness and generalization abilities were systematically assessed across multiple random seeds and evaluation scenarios. Future work might consider incorporating curriculum learning strategies or domain randomization techniques to further strengthen policy adaptability and generalization.

E. Debugging and Evaluation Instrumentation

Developing a stable and Gym-compatible custom environment within the PyBullet physics engine introduced substantial engineering challenges, notably regarding state encoding, dynamic obstacle generation, and rendering integration. Employing comprehensive visualization tools—such as training reward curves, evaluation metrics tables, and animated policy demonstrations—proved essential for debugging, performance diagnostics, and interpreting agent behaviours. These diagnostic frameworks facilitated environment validation, identified and mitigated simulation artifacts, and significantly enhanced experimental reproducibility.

Collectively, these encountered challenges emphasize the critical importance of meticulous reward function design, carefully selected performance evaluation criteria, and robust debugging and visualization infrastructure. Addressing these considerations is indispensable for successfully applying deep reinforcement learning to realistic and complex simulation tasks inspired by real-world scenarios.

V. Reflection

This project provided a valuable opportunity to consolidate theoretical knowledge and practical skills across reinforcement learning (RL), simulation engineering, and deep neural network development. Implementing a Deep Q-Network (DQN) agent in a dynamic PyBullet environment unveiled numerous practical challenges extending beyond standard textbook RL formulations.

A. Deepened Understanding of Reward Dynamics

A pivotal insight from this project involved recognizing the subtle yet critical relationship between reward magnitude and resultant agent behaviour. Iterative experimentation underscored that reward shaping significantly influences not only the convergence speed but also the quality and stability of the learned policy. Developing the concept of a *success threshold* helped mitigate ambiguities during evaluation, highlighting the necessity of aligning reward design closely with meaningful and interpretable performance metrics.

B. Iterative Nature of Hyperparameter Tuning

Engagement with hyperparameter tuning reinforced the iterative and context-sensitive nature inherent in RL experiments. Key hyperparameters—including learning rate, ϵ decay schedules, and reward-related constants—required extensive empirical adjustments and validation through repeated testing cycles. This iterative refinement substantially enhanced problem-solving abilities and diagnostic proficiency, particularly facilitated by visualization tools such as training reward curves and performance evaluation plots.

C. Navigating from Failure to Policy Convergence

Initial training episodes frequently resulted in unsuccessful outcomes, characterized by the agent exhibiting erratic movements, such as repetitive spinning or unintended collisions. These early failures reflected deficiencies in reward feedback mechanisms or inadequate initial exploration. Through methodical incorporation of shaped reward terms and calibrated adjustments to the ϵ-greedy exploration strategy, the agent transitioned gradually from exploratory randomness to coherent, goal-oriented navigation behaviours. This evolution highlighted the significance of patience and structured experimentation in achieving effective RL solutions.

D. Technical Growth in Simulation and Evaluation Engineering

On the technical implementation side, constructing a custom Gym-compatible environment within PyBullet necessitated mastering several intricate processes, including environment registration, robust state encoding strategies, and visual debugging techniques. Integrating dynamic obstacles with randomized positions and sizes enhanced realism but also required careful attention to observation representation to ensure training stability.

Moreover, managing evaluation protocols, developing informative plots, and generating qualitative visualizations via animated episode recordings significantly honed technical communication skills. These tasks also emphasized best practices for ensuring experimental reproducibility and reliability.

E. Connection with Prior Robotic Control Experiences

This project effectively extends previous experiences in robotics, specifically multi-robot navigation systems developed in ROS1 and leader–follower formation control implemented via model predictive control (MPC). Compared to these earlier rule-based methods, the RL-driven navigation policy developed here demonstrated enhanced adaptability and autonomy in uncertain, dynamic scenarios. This comparison underscores the complementary and powerful role that reinforcement learning can play alongside traditional robotic control methodologies.

Overall, this project served as an integrative experience, bridging theoretical understanding with practical execution. It clearly illustrated reinforcement learning's potential for robotic control tasks while providing authentic contexts for rigorous debugging, comprehensive evaluation, and reflective growth as a robotics engineer and researcher.

VI. Conclusion

This project demonstrated the successful implementation of Deep Q-Networks (DQN) for autonomous navigation within a simulated PyBullet environment characterized by randomly positioned and variably sized obstacles. By utilizing a discrete action space and carefully designed reward-shaping strategies, the developed agent effectively learned to navigate towards dynamically assigned goals while consistently avoiding collisions, even amidst stochastic and unpredictable environmental conditions.

The employed training and evaluation framework encompassed both quantitative performance metrics—such as average cumulative rewards, success rates, and average steps-to-goal—and qualitative analyses facilitated by visual demonstrations and GIF animations. These multi-dimensional evaluation approaches offered comprehensive insights into the robustness and effectiveness of the learned policy. Additionally, the introduction of a clearly defined *success threshold* enhanced the clarity and interpretability of performance assessment, bridging the gap between cumulative rewards and task-oriented success criteria.

Throughout the project, iterative hyperparameter tuning, reward-shaping experimentation, and environmental complexity adjustments played critical roles. These iterative refinements not only enhanced policy convergence and stability but also provided deeper insights into core reinforcement learning concepts, such as exploration–exploitation trade-offs, temporal difference (TD) learning, and effective policy generalization in randomized scenarios.

The outcomes and insights from this project lay a robust foundation for future advancements in related fields. Promising directions include:

- Transitioning from discrete to continuous action spaces and employing more sophisticated function approximators, such as actor–critic architectures.
- Extending the navigation paradigm to multi-agent cooperative settings, facilitating coordinated vehicle interactions under shared environmental constraints.
- Combining reinforcement learning with traditional model-based control strategies, particularly model predictive control (MPC), to enhance policy safety, interpretability, and precision.
- Moving beyond simulation environments toward physical robotic implementations, leveraging sim-to-real transfer learning techniques to bridge the reality gap effectively.

Furthermore, this project effectively leveraged prior experiences gained from rule-based multi-robot formation control using MPC within the ROS1 framework. In contrast, this RL-driven approach underscores reinforcement learning's unique capability to produce adaptable, robust policies without reliance on explicit trajectory planning. Synthesizing traditional control methodologies with data-driven reinforcement learning approaches emerges as a promising paradigm, offering

enhanced precision and adaptability in real-world robotic deployments.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT press, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] E. Coumans and Y. Bai, "Pybullet physics sdk," https://pypi.org/project/pybullet/, 2024, accessed: 2025-04-01.

[4] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms," https://pypi.org/project/gym/, 2024, accessed: 2025-04-01.

[5] Anaconda Inc., "Anaconda official documentation," https://www.anaconda.com/docs/main, 2024, accessed: 2025-04-01.

[6] M. Galarnyk, "Python tutorials for data science and machine learning," https://github.com/mGalarnyk/Python_Tutorials, 2018, accessed: 2025-04-01.

[7] P. Wen, "Deep reinforcement learning practice with tensorflow," https://github.com/princewen/tensorflow_practice, 2018, accessed: 2025-04-01.

[8] OpenAI, "Spinning up in deep rl," https://spinningup.openai.com/, 2018, accessed: 2025-04-01.

Appendix

This appendix presents visual references to key implementation components of the project. Each figure highlights an important module from the Deep Q-Network (DQN) pipeline, including the agent architecture, training loop, environment dynamics, and evaluation logic.

The complete project source code is available on GitHub: https://github.com/caesar1457/RL_simple_car

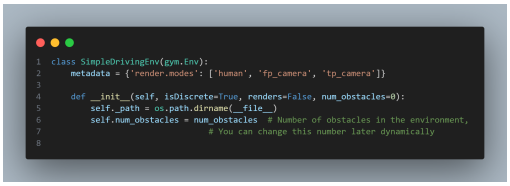
```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import gym
import matplotlib.pyplot as plt
import os

# ==============================
# Part 1: QNetwork
# ==============================
class QNetwork(nn.Module):
    """
    A simple multilayer perceptron to estimate Q-values for each action.
    Input is a 2D state (relative position to the goal), output is Q-values for 9 discrete actions.
    input_dim: Dimension of input state (2)
    output_dim: Dimension of output Q-values (9)
    hidden_dim: Dimension of hidden 128
    """
    def __init__(self, input_dim=8, output_dim=9, hidden_dim=128):
        super(QNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),  # Input layer -> Hidden layer
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)  # Hidden layer -> Output layer
        )

    def forward(self, x):
        return self.model(x)

# ==============================
# Part 2: ε-greedy Action Selection
# ==============================
def select_action(model, state, epsilon, env):
    """
    Select an action based on the ε-greedy strategy:
      - With probability epsilon, choose a random action (exploration)
      - Otherwise, choose the action with the highest Q-value (exploitation)
    Parameters:
        model: QNetwork model
        state: Current state (must be torch.tensor)
        epsilon: Exploration rate
        env: Gym environment (used for sampling random actions)
    Returns:
        action: Chosen action index (integer)
    """
    if np.random.rand() < epsilon:
        # Prefer forward-moving actions more than others
        action_probs = np.array([0.25, 0.2, 0.2, 0.1, 0.1, 0.05, 0.05, 0.025, 0.025])
        action_probs /= action_probs.sum()  # Normalize
        action = np.random.choice(np.arange(env.action_space.n), p=action_probs)
    else:
        with torch.no_grad():
            q_values = model(state)
            action = torch.argmax(q_values).item()
    return action

# ==============================
# Part 3: Q-learning Training Loop
# ==============================
def train_q_learning(env, input_dim, output_dim, hidden_dim, episodes=500, max_steps=200):
    """
    Train the agent using Q-learning. During training, MSELoss is used to calculate TD error,
    and the Adam optimizer is used to update network parameters.
    At each time step, Q-values are updated using the Bellman equation:
        target = reward + gamma * max(Q(next_state)) (if not done)
    The ε-greedy strategy is used for exploration.

    Returns:
        model: Trained QNetwork model
        reward_history: Total reward per episode (list)
    """
    model = QNetwork(input_dim=input_dim, output_dim=output_dim, hidden_dim=hidden_dim)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.MSELoss()

    epsilon = 1.0
    epsilon_min = 0.1
    epsilon_decay = 0.995
    gamma = 0.99

    reward_history = []

    for ep in range(episodes):
        state, _ = env.reset()
        state = torch.tensor(state, dtype=torch.float32)
        total_reward = 0

        for step in range(max_steps):
            action = select_action(model, state, epsilon, env)
            next_state, reward, done, _, _ = env.step(action)
            next_state_tensor = torch.tensor(next_state, dtype=torch.float32)

            with torch.no_grad():
                q_next = model(next_state_tensor).max().item()
            target = reward + gamma * q_next * (1 - int(done))

            pred = model(state)[action]
            loss = criterion(pred, torch.tensor(target))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            state = next_state_tensor
            total_reward += reward

            if done:
                break

        reward_history.append(total_reward)
        epsilon = max(epsilon_min, epsilon * epsilon_decay)

        if ep % 50 == 0:
            print(f"Episode {ep}, Total Reward: {total_reward:.2f}, Epsilon: {epsilon:.3f}")

    return model, reward_history

# ==============================
# Part 4: Training Reward Plotting
# ==============================
def plot_training_curve(reward_history, save_path=None, smooth_window=10):
    """
    Plot the total reward curve during training, with optional moving average smoothing.

    Parameters:
        reward_history: List of total rewards per episode
        save_path: If specified, saves the plot to this path
        smooth_window: Window size for moving average smoothing
    """
    plt.figure(figsize=(10, 5))
    plt.plot(reward_history, label="Raw Reward", alpha=0.4)

    if len(reward_history) >= smooth_window:
        smooth = np.convolve(reward_history, np.ones(smooth_window)/smooth_window, mode="valid")
        plt.plot(np.arange(smooth_window - 1, len(reward_history)), smooth,
                 label=f"Smoothed (window={smooth_window})", color='orange')

    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("Training Reward Curve")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()

    if save_path:
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        plt.savefig(save_path)
        print(f"📊 Training reward curve saved to: {save_path}")
    else:
        plt.show()
```

```python
import torch
import matplotlib.pyplot as plt
import numpy as np

# ==============================
# Part 1: Evaluation (Model Testing)
# ==============================
class Evaluator:
    def __init__(self, env, model, success_threshold=100.0):
        """
        Parameters:
            env: Gym environment object
            model: Trained QNetwork model
            success_threshold: Reward threshold to determine if the task is successful
        """
        self.env = env
        self.model = model
        self.success_threshold = success_threshold

    def evaluate(self, episodes=10, max_steps=200, epsilon=0.0, plot=True):
        """
        Evaluate the model's performance over multiple episodes.
        Returns evaluation metrics.
        """
        rewards = []
        steps_to_success = []
        success_count = 0

        for ep in range(episodes):
            state, _ = self.env.reset()
            state = torch.tensor(state, dtype=torch.float32)
            total_reward = 0
            step_count = 0

            for step in range(max_steps):
                action = self.select_action(state, epsilon)
                next_state, reward, done, _, _ = self.env.step(action)
                state = torch.tensor(next_state, dtype=torch.float32)
                total_reward += reward
                step_count += 1
                if done:
                    break

            rewards.append(total_reward)
            if total_reward >= self.success_threshold:
                success_count += 1
                steps_to_success.append(step_count)

            print(f"[Episode {ep+1}] Total Reward: {total_reward:.2f}, Steps: {step_count}")

        avg_reward = np.mean(rewards)
        avg_steps = np.mean(steps_to_success) if steps_to_success else float('nan')
        success_rate = success_count / episodes

        print("\n Evaluation Summary:")
        print(f" Average Reward      : {avg_reward:.2f}")
        print(f" Success Rate        : {success_rate * 100:.1f}%")
        print(f" Avg Steps to Success: {avg_steps:.1f}")

        if plot:
            self._plot_rewards(rewards)

        return {
            'average_reward': avg_reward,
            'success_rate': success_rate,
            'avg_steps_to_success': avg_steps,
            'rewards': rewards
        }

    def select_action(self, state, epsilon):
        """
        Select action using ε-greedy strategy
        """
        if np.random.rand() < epsilon:
            return self.env.action_space.sample()
        else:
            with torch.no_grad():
                q_values = self.model(state)
                return torch.argmax(q_values).item()

    def _plot_rewards(self, rewards):
        plt.figure(figsize=(8, 4))
        plt.plot(rewards, marker='o')
        plt.title("Episode Reward Curve")
        plt.xlabel("Episode")
        plt.ylabel("Total Reward")
        plt.grid()
        plt.tight_layout()
        plt.show()
```

Fig. 5. Evaluator class used for computing average reward, success rate, and average steps-to-goal over multiple evaluation episodes.

```python
def getExtendedObservation(self):
    # self._observation = []  #self._racecar.getObservation()
    carpos, carorn = self._p.getBasePositionAndOrientation(self.car.car)
    goalpos, goalorn = self._p.getBasePositionAndOrientation(self.goal_object.goal)
    invCarPos, invCarOrn = self._p.invertTransform(carpos, carorn)
    goalPosInCar, goalOrnInCar = self._p.multiplyTransforms(invCarPos, invCarOrn, goalpos, goalorn)

    # obstaclepos, obstacleorn = self._p.getBasePositionAndOrientation(self.obstacle)
    # obsPosInCar, _ = self._p.multiplyTransforms(invCarPos, invCarOrn, obstaclepos, obstacleorn)

    # observation = [goalPosInCar[0], goalPosInCar[1],
    #                obsPosInCar[0], obsPosInCar[1]]

    obs_relative_positions = []
    for obstacle_id in self.obstacles:
        obstaclepos, obstacleorn = self._p.getBasePositionAndOrientation(obstacle_id)
        obs_pos_in_car, _ = self._p.multiplyTransforms(invCarPos, invCarOrn, obstaclepos, obstacleorn)
        obs_relative_positions.extend([obs_pos_in_car[0], obs_pos_in_car[1]])

    observation = [goalPosInCar[0], goalPosInCar[1]] + obs_relative_positions

    return observation
```

Fig. 6. Function for computing extended observations, including goal and obstacle positions relative to the car.



```python
class SimpleDrivingEnv(gym.Env):
    metadata = {'render.modes': ['human', 'fp_camera', 'tp_camera']}

    def __init__(self, isDiscrete=True, renders=False, num_obstacles=0):
        self._path = os.path.dirname(__file__)
        self.num_obstacles = num_obstacles  # Number of obstacles in the environment,
                                            # You can change this number later dynamically
```

Fig. 7. Gym environment class declaration with render modes and dynamic obstacle count configuration.

```python
# src/RL_simple_car/agent/main.py

import os
import sys
import torch
import gym
import time

# === 0. Add simple_driving to sys.path to support environment registration ===
current_dir = os.path.dirname(os.path.abspath(__file__))
root_dir = os.path.abspath(os.path.join(current_dir, ".."))
if root_dir not in sys.path:
    sys.path.insert(0, root_dir)

# === 0.1 Register custom Gym environment ===
from gym.envs.registration import register
try:
    register(
        id="SimpleDriving-v0",
        entry_point="simple_driving.envs.simple_driving_env:SimpleDrivingEnv"
    )
except gym.error.Error:
    pass  # Already registered

# === 1. Import modules ===
from dqn import QNetwork, train_q_learning
from evaluator import Evaluator
from report_charts import (
    plot_training_curve,
    plot_evaluation_metrics,
    plot_epsilon_distribution,
    create_evaluation_gif
)
import imageio

OUTPUT_DIM = 9
HIDDEN_DIM = 128
NUM_OBSTACLES = 3
INPUT_DIM = 2 + 2 * NUM_OBSTACLES

# === Config ===
show_plot = False  # Toggle this to control whether to display final figures interactively

if __name__ == "__main__":
    # === 2. Create training environment ===
    print("🖥️ Creating training environment...")
    train_env = gym.make("SimpleDriving-v0",
                         apply_api_compatibility=True,
                         renders=False,
                         isDiscrete=True,
                         num_obstacles=NUM_OBSTACLES
                         )

    # === 3. Set up directories and timestamp ===
    timestamp = time.strftime("%Y%m%d-%H%M%S")
    data_dir = os.path.join(current_dir, "..", "data")
    model_dir = os.path.join(data_dir, "models")
    video_dir = os.path.join(data_dir, "videos")
    graph_dir = os.path.join(data_dir, "graphs")
    os.makedirs(model_dir, exist_ok=True)
    os.makedirs(video_dir, exist_ok=True)
    os.makedirs(graph_dir, exist_ok=True)

    model_path = os.path.join(model_dir, f"simple_driving_qlearning_{timestamp}.pth")
    curve_path = os.path.join(graph_dir, f"training_curve_{timestamp}.png")
    gif_path = os.path.join(graph_dir, f"evaluation_animation_{timestamp}.gif")
    metrics_path = os.path.join(graph_dir, f"evaluation_metrics_{timestamp}.png")
    epsilon_path = os.path.join(graph_dir, f"epsilon_distribution_{timestamp}.png")

    # === 4. Train model ===
    print("🚗 Starting training of DQN model...")
    trained_model, reward_history = train_q_learning(
                                                    train_env,
                                                    input_dim=INPUT_DIM,
                                                    output_dim=OUTPUT_DIM,
                                                    hidden_dim=HIDDEN_DIM,
                                                    episodes=500,
                                                    max_steps=200
                                                )
    torch.save(trained_model.state_dict(), model_path)
    print(f"✅ Model saved to: {os.path.relpath(model_path)}")
    train_env.close()

    # === 5. Reload model for evaluation ===
    print("\n📦 Loading model for evaluation...")
    loaded_model = QNetwork(input_dim=INPUT_DIM, output_dim=OUTPUT_DIM, hidden_dim=HIDDEN_DIM)
    loaded_model.load_state_dict(torch.load(model_path))
    loaded_model.eval()
    print("✅ Model loaded successfully.\n")

    # === 6. Create evaluation environment for video ===
    print("🎥 Creating evaluation environment (third-person view)...")
    eval_env = gym.make("SimpleDriving-v0",
                        apply_api_compatibility=True,
                        renders=False,
                        isDiscrete=True,
                        render_mode='tp_camera',
                        num_obstacles=NUM_OBSTACLES
                        )

    # === 7. Run and record ONE episode ===
    print("🎬 Running evaluation episode and capturing frames...")
    state, _ = eval_env.reset()
    state = torch.tensor(state, dtype=torch.float32)
    frames = []
    total_reward = 0

    for step in range(200):
        with torch.no_grad():
            action = torch.argmax(loaded_model(state)).item()
        next_state, reward, done, _, _ = eval_env.step(action)
        state = torch.tensor(next_state, dtype=torch.float32)
        total_reward += reward
        frame = eval_env.render()
        if frame is not None:
            frames.append(frame)
        if done:
            break

    eval_env.close()
    print(f"🎯 Total evaluation reward: {total_reward:.2f}")

    # === 8. Create new env for evaluation metrics ===
    metrics_env = gym.make("SimpleDriving-v0",
                           apply_api_compatibility=True,
                           renders=False,
                           isDiscrete=True,
                           num_obstacles=NUM_OBSTACLES
                           )
    evaluator = Evaluator(metrics_env, loaded_model, success_threshold=100.0)
    evaluation_result = evaluator.evaluate(episodes=10, max_steps=200, epsilon=0.0, plot=False)
    metrics_env.close()

    # === 9. Save training and evaluation charts ===
    plot_training_curve(reward_history, save_path=curve_path)
    # plot_evaluation_metrics(evaluation_result, save_path=metrics_path)
    create_evaluation_gif(frames, save_path=gif_path)
    mp4_path = os.path.join(video_dir, f"evaluation_episode_{timestamp}.mp4")
    imageio.mimsave(mp4_path, frames, fps=30)
    print(f"[✓] MP4 video also saved to: {mp4_path}")

    # Optional: ε-greedy distribution if used non-uniform
    action_probs = [0.25, 0.2, 0.2, 0.1, 0.1, 0.05, 0.05, 0.025, 0.025]
    plot_epsilon_distribution(action_probs, save_path=epsilon_path)

    # === 10. Optionally show plots interactively ===
```

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import imageio

# Create output directories (e.g., data/graphs and data/videos)
current_dir = os.path.dirname(os.path.abspath(__file__))
data_dir = os.path.join(current_dir, "..", "data")
graph_dir = os.path.join(data_dir, "graphs")
video_dir = os.path.join(data_dir, "videos")
os.makedirs(graph_dir, exist_ok=True)
os.makedirs(video_dir, exist_ok=True)


def plot_training_curve(reward_history, smooth_window=10, save_path="training_reward_curve.png"):
    plt.figure(figsize=(10, 5))
    plt.plot(reward_history, label="Raw Reward", alpha=0.4)

    if len(reward_history) >= smooth_window:
        smooth = np.convolve(reward_history, np.ones(smooth_window)/smooth_window, mode="valid")
        plt.plot(np.arange(smooth_window - 1, len(reward_history)), smooth,
                 label=f"Smoothed (window={smooth_window})", color="orange")

    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("Training Reward Curve")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(save_path)
    print(f"[✓] Training reward curve saved to: {save_path}")
    plt.close()


def create_evaluation_gif(frames, save_path="evaluation_animation.gif", fps=10):
    imageio.mimsave(save_path, frames, fps=fps)
    print(f"[✓] Evaluation animation saved as: {save_path}")


def plot_evaluation_metrics(evaluation_result, save_path="evaluation_metrics.png"):
    metrics = {
        "Avg Reward": evaluation_result.get("average_reward", 0),
        "Success Rate (%)": evaluation_result.get("success_rate", 0) * 100,
        "Avg Steps": evaluation_result.get("avg_steps_to_success", 0)
    }
    labels = list(metrics.keys())
    values = list(metrics.values())

    plt.figure(figsize=(8, 6))
    bars = plt.bar(labels, values, color=["blue", "green", "orange"])
    plt.title("Evaluation Metrics")
    plt.ylabel("Value")

    for bar in bars:
        yval = bar.get_height()
        plt.text(bar.get_x() + bar.get_width() / 2.0, yval + 0.05 * yval, f"{yval:.2f}", ha="center", va="bottom")

    plt.tight_layout()
    plt.savefig(save_path)
    print(f"[✓] Evaluation metrics saved to: {save_path}")
    plt.close()


def plot_epsilon_distribution(probabilities, save_path="epsilon_distribution.png"):
    labels = [f"Action {i}" for i in range(len(probabilities))]

    plt.figure(figsize=(8, 8))
    plt.pie(probabilities, labels=labels, autopct="%1.1f%%", startangle=140)
    plt.title("Epsilon-Greedy Exploration Distribution")
    plt.tight_layout()
    plt.savefig(save_path)
    print(f"[✓] Epsilon distribution saved to: {save_path}")
    plt.close()


```

Fig. 9. Utility functions for generating reward curves, evaluation metrics, action distribution plots, and GIFs.

```python
def reset(self):
    self._p.resetSimulation()
    self._p.setTimeStep(self._timeStep)
    self._p.setGravity(0, 0, -10)
    # Reload the plane and car
    Plane(self._p)
    self.car = Car(self._p)
    self._envStepCounter = 0

    # Set the goal to a random target
    x = (self.np_random.uniform(5, 9) if self.np_random.integers(2) else
         self.np_random.uniform(-9, -5))
    y = (self.np_random.uniform(5, 9) if self.np_random.integers(2) else
         self.np_random.uniform(-9, -5))
    self.goal = (x, y)
    self.done = False
    self.reached_goal = False

    # obstacle_path = os.path.join(self._path, "..", "resources/simplegoal.urdf")
    # obstacle_pos = [2, 2, 0]
    # self.obstacle = self._p.loadURDF(obstacle_path, basePosition=obstacle_pos)
    # self.obstacle_pos = obstacle_pos[:2]

    # Create multiple random red obstacles
    self.obstacles = []
    self.obstacle_pos_list = []
    red_rgba = [1, 0, 0, 1]

    for _ in range(self.num_obstacles):
        obs_x = self.np_random.uniform(-4, 4)
        obs_y = self.np_random.uniform(-4, 4)
        obs_pos = [obs_x, obs_y, 0]

        radius = self.np_random.uniform(0.4, 0.8)

        visual_shape_id = self._p.createVisualShape(
            shapeType=self._p.GEOM_SPHERE,
            radius=radius,
            rgbaColor=red_rgba
        )
        collision_shape_id = self._p.createCollisionShape(
            shapeType=self._p.GEOM_SPHERE,
            radius=radius
        )
        obstacle_id = self._p.createMultiBody(
            baseMass=0,
            baseCollisionShapeIndex=collision_shape_id,
            baseVisualShapeIndex=visual_shape_id,
            basePosition=obs_pos
        )

        self.obstacles.append(obstacle_id)
        self.obstacle_pos_list.append(obs_pos[:2])

    # Visual element of the goal
    self.goal_object = Goal(self._p, self.goal)

    # Get observation to return
    carpos = self.car.get_observation()

    self.prev_dist_to_goal = math.sqrt(((carpos[0] - self.goal[0]) ** 2 +
                                        (carpos[1] - self.goal[1]) ** 2))
    car_ob = self.getExtendedObservation()
    return np.array(car_ob, dtype=np.float32)
```

Fig. 10. Environment reset function with randomized goal and obstacle initialization.

```python
def step(self, action):
    # Feed action to the car and get observation of car's state
    if (self._isDiscrete):
        fwd = [-1, -1, -1, 0, 0, 0, 1, 1, 1]
        steerings = [-0.6, 0, 0.6, -0.6, 0, 0.6, -0.6, 0, 0.6]
        throttle = fwd[action]
        steering_angle = steerings[action]
        action = [throttle, steering_angle]
    self.car.apply_action(action)
    for i in range(self._actionRepeat):
      self._p.stepSimulation()
      if self._renders:
        time.sleep(self._timeStep)

      carpos, carorn = self._p.getBasePositionAndOrientation(self.car.car)
      goalpos, goalorn = self._p.getBasePositionAndOrientation(self.goal_object.goal)
      car_ob = self.getExtendedObservation()

      if self._termination():
        self.done = True
        break
      self._envStepCounter += 1

    # Compute reward as L2 change in distance to goal
    # dist_to_goal = math.sqrt(((car_ob[0] - self.goal[0]) ** 2 +
    #                           (car_ob[1] - self.goal[1]) ** 2))
    dist_to_goal = math.sqrt(((carpos[0] - goalpos[0]) ** 2 +
                              (carpos[1] - goalpos[1]) ** 2))
    # reward = max(self.prev_dist_to_goal - dist_to_goal, 0)
    reward = -dist_to_goal
    self.prev_dist_to_goal = dist_to_goal

    # Done by reaching goal
    if dist_to_goal < 1.5 and not self.reached_goal:
        #print("reached goal")
        self.done = True
        self.reached_goal = True
        reward += 300  # Reward for reaching goal

    ob = car_ob
    return ob, reward, self.done, dict()
```

Fig. 11. Environment step function: applies discrete actions, simulates forward dynamics, computes rewards, and checks for termination.

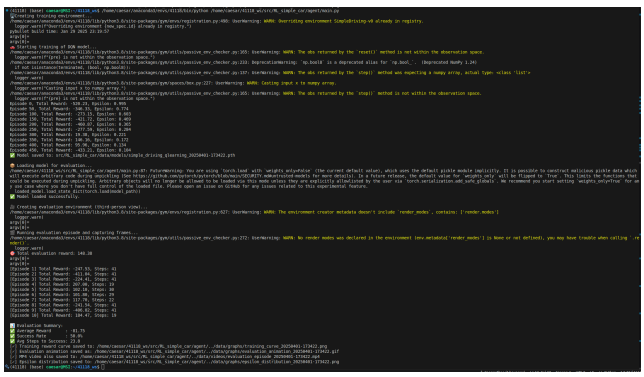


Fig. 12. Terminal output showing reward history and evaluation summary for 10 independent test episodes.