Reinforcement Learning on Autonomous Vehicles

Emirhan Balcı 200101XXX,
Ahmet Burak Biçer 210101XXX

January 2025

Figure 1: Sample images from the simulation environment.

1 Introduction

Autonomous vehicles are becoming increasingly prevalent due to recent advancements in artificial intelligence (AI). The AI algorithms employed in navigation and pathfinding pipelines have facilitated the execution of complex tasks in robotics and autonomous vehicles, including task and motion planning. Our project aims to evaluate the performance of the widely used reinforcement learning algorithm, Deep Q-Learning (DQN), in a simulated environment where a vehicle agent completes a predefined route. We designed a simple route within a small neighborhood environment and employed a realistic simulator to enable a smooth transition from the simulated setting to real-world scenarios. We captured images using the camera sensor mounted at the front of the vehicle and used them as input to the DQN network, which outputs predefined actions for the vehicle to adjust its direction based on its current state. These actions were executed in real time to simulate the task. After a specified number of epochs, we saved the model weights, generated re-

ward, and Q-value graphs, and evaluated the predicted routes using the common trajectory metrics in robotics. The source code can be found at `https://github.com/BashMocha/Reinforcement-Learning-in-AirSim`.

The report is organized as follows: the second section presents a brief overview of the research, including the collection of open-source projects and relevant papers. The third section explains the algorithms used, the simulation environment, and the evaluation metrics. The fourth section presents the experimental results and analysis. Finally, the conclusion is provided in the last section.

2 Related Work and Prior Research

We researched and collected relevant resources from prior studies to determine (i) which reinforcement algorithms to apply, (ii) the simulation environment to use for task simulation, and (iii) the evaluation metrics to assess the resulting trajectories.

Vemulapalli et al. [1] employed the DQN algorithm for reinforcement learning to land unmanned aerial vehicles (UAVs) in complex environments. We also found several open-source implementations for the DQN algorithm to navigate a vehicle in complex environments [2–5].

We gathered and evaluated widely used open-source robotics simulations for reinforcement learning applications. Gazebo [6], Habitat [7], RLBench [8], Webots [9], and AirSim [10] are prominent in this field, offering API control to manipulate various agents, including UAVs, vehicles, and embodied agents. We utilized the AirSim simulator to train a vehicle agent for a navigation task, due to its ease of implementation and clear documentation compared to other simulators. Figure 1 presents sample images from the simulation environment.

To evaluate the performance of the trained agent's trajectory relative to the reference path, we reviewed robotics trajectory metrics and selected four commonly used ones, Navigation Error (NE), Success Rate (SR), Success rate weighted by Normalised Dynamic Time Warping (SDTW), Cover Length Score (CLS) [11, 12].

3 Methodology

3.1 DQN Algorithm

We employ a variant of Q-learning called Deep Q-learning (DQL) to train an agent for navigating in a small environment. In this section, we briefly give an overview of the Q-learning and Deep Q-learning algorithms.

Q-learning learns the action-value function $Q(s, a)$ to quantify the effectiveness of taking an action at a particular state. Q is called the action-value function (or Q-value function). In Q-learning, a lookup table/memory table $Q[s, a]$ is constructed during training to store Q-values for all possible combinations of states s and actions a. An action is sampled from the current state, followed by computation of reward R and then the new state s'.

From the memory table, the next action a' is determined based on the maximum of $Q(s', a')$. After this, an action a is performed to seek a reward of R. Based on this one-step look ahead, the target $Q(s, a)$ is set to

$$target = R(s, a, s') + \gamma max_{a'} Q(s', a') \tag{1}$$

The update equations are called Bellman equations and are performed iteratively with dynamic programming. As this update is performed iteratively until convergence, and a running average for Q is maintained.

However, for solving a real-world problem such as path planning and navigation, where the combinations of states and actions are too large, the memory and the computational requirements for Q are very expensive in some cases. To address that issue, a deep Q-network (DQN) framework was introduced to approximate Q(s, a) with the aid of neural network parameters. The goal of this algorithm is to maximize the action-value function:

$$Q(s, a) \approx R(s, a, s') + \gamma max_{a'} Q(s', a') = Q_{tar}(s, a) \tag{2}$$

where $Q_{tar}(s, a)$ is the target action-value function and $Q(s', a')$ is produced by the neural network that is used for choosing the actions a in a state s. Since the duration of the task can be unlimited, the rewards are multiplied by a discount factor γ. The target Q-function $Q_{tar}(s, a)$ is periodically updated using the separate network $Q(s, a)$ to enhance stability and prevent divergence during training.

The reward function was designed to encourage the agent to complete the path while avoiding collisions. The reward increases by 100 points for each waypoint reached and by 500 points when the entire path is completed. A penalty of -100 points is applied if a collision is detected via the AirSim API, causing the simulation to reset and the agent to start over. This reward system promotes successful navigation and discourages collisions.

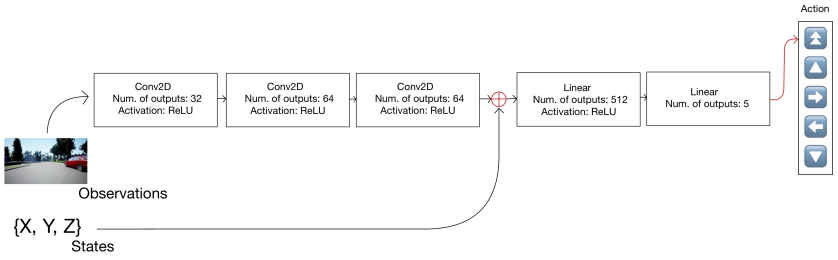

Figure 2: Applied DQN architecture.

3.2 DQN Architecture

We adopted a convolutional neural network model [13], to learn the features and information based on the current observation image and state information from the simulation environment. The DQN model takes the current image observation from the simulation, resized to 84×84, as input. It consists of three convolutional layers that extract features from the image. The filter sizes in these layers decrease from 8×8 to 3×3, with strides adjusting from 4×4 to 1×1, and each layer is followed by ReLU activation. After the last convolutional layer, the outputs are concatenated with the current state of the simulation, represented by the X, Y, and Z coordinates, to inform the network about the vehicle's current position. The concatenated features are passed through a fully connected layer with 512 neurons, followed by another ReLU activation. The final layer outputs five actions, representing the possible navigation decisions for the vehicle.

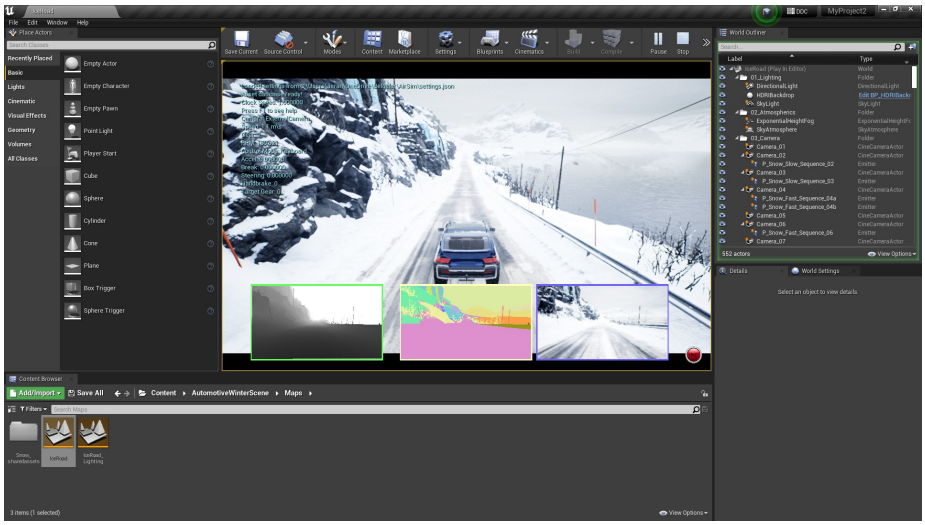

Figure 3: Custom environment on Unreal Engine 4.

3.3 Simulator

Our experiment was carried out using the AirSim simulator [13] with Unreal Engine 4 [14], providing a robust and dynamic simulation environment. AirSim, an open-source simulation platform developed by Microsoft Research and founded in 2017, provides high-fidelity, photorealistic virtual environments for testing autonomous vehicles such as drones and cars. The utilized neighborhood environment is directly derived from the simulator's tutorial document on reinforcement learning[1]. Figure 2 presents a sample image from the simulation

[1] https://microsoft.github.io/AirSim/reinforcement_learning

environment.

We defined a simple route within the environment consisting of seven waypoints to train the agent to follow the path. The first four waypoints form a straight trajectory along the street, followed by a 90° left turn. The final three waypoints guide the agent along a shorter straight path to the end of the predefined trajectory. Figure 4 illustrated the predefined trajectory along with its top-down look from the simulation environment.

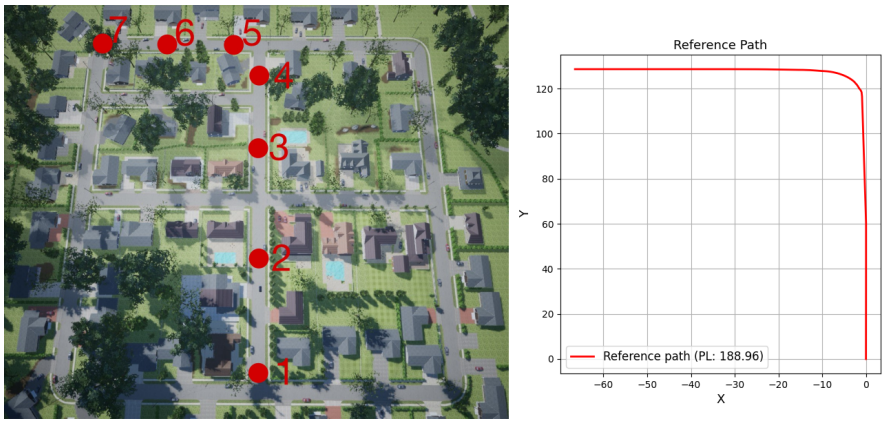

Figure 4: Top-down view of the predefined trajectory with waypoints and the reference plot.

3.4 Evaluation Metrics

We adopt three commonly utilized metrics in our study [15–17]: Success Rate (SR) is a binary indicator that determines whether the last node of the predicted path lies within a threshold distance of the last node of the reference path; Navigation Error (NE), the distance between the stop location to the destination; Success rate weighted by Normalized Dynamic Time Warping (SDTW), considers both the navigation success rate and the similarity between the predicted path and reference path; Cover Length Score (CLS), evaluates the extent to which an agent's trajectory aligns with the entire reference path, rather than merely assessing goal completion. This data served as a comparative metric to assess the alignment between the performed and reference trajectories, reflecting the models' navigation performance. Furthermore, we defined the threshold distance as 5 units, meaning the agent successfully completes the path if it reaches or stops within this distance from the destination point.

Navigation Error (NE) is the distance between the terminal location in the predicted path P and the destination location in the reference path R [15]:

$$\mathrm{NE}(R, P) = d(r_{|R|}, p_{|P|}) \tag{3}$$

SR evaluates if the final node in the predicted path P lies within a threshold distance of d_{th} from the last node in the reference path R [15]:

$$\mathrm{SR}(R, P) = \mathbb{1}[\mathrm{NE}(R, P) \leq d_{th}] \tag{4}$$

Success rate weighted by Normalised Dynamic Time Warping (SDTW), considers both the navigation success rate and the similarity between the predicted path P and reference path R on the successful episodes by combining Success Rate (SR) and normalized Dynamic Time Warping (nDTW) metrics [15]:

$$\mathrm{SDTW}(R, P) = \mathrm{SR}(R, P) \cdot \mathrm{nDTW}(R, P) \tag{5}$$

nDTW measures the similarity between two sequences by calculating the optimal alignment cost, normalized to account for sequence length differences. Unlike standard Dynamic Time Warping (DTW), nDTW normalizes this alignment cost to account for differences in the lengths of the sequences, allowing for more meaningful comparisons across sequences of varying sizes. The closer the nDTW score is to 1, the more similar the two sequences are [16]:

$$\mathrm{nDTW}(R, P) = \exp\left(-\frac{\min_{W \in \mathcal{W}} \sum_{(i_k, j_k) \in W} d(r_{i_k}, p_{j_k})}{|R| \cdot d_{th}}\right) \tag{6}$$

Cover Length Score (CLS), evaluates the extent to which an agent's trajectory aligns with the entire reference path R, rather than merely assessing goal completion. CLS is calculated as the product of the Path Coverage (PC) and Length Score (LS) for the agent's path P with the reference path R [16]:

$$\mathrm{CLS}(R, P) = \mathrm{PC}(R, P) \cdot \mathrm{LS}(R, P) \tag{7}$$

PC represents the average coverage of each node in the reference path R relative to the predicted path P [16]:

$$\mathrm{PC}(R, P) = \frac{1}{|R|} \sum_{r \in R} \exp\left(-\frac{d(r, P)}{d_{th}}\right) \tag{8}$$

LS compares the predicted path length, $\mathrm{PL}(P)$, to the expected optimal length (EPL), based on how well the reference path R covers path P. If, for example, the predicted path covers only half of the reference path (i.e., $PC = 0.5$), the expected optimal length of the predicted path would be half the length of the reference path. Therefore, EPL is calculated as [17]:

$$\mathrm{EPL}(R, P) = \mathrm{PC}(R, P) \cdot \mathrm{PL}(R) \tag{9}$$

LS for a predicted path P is optimal when the path length, $\mathrm{PL}(P)$, equals the expected optimal length, with penalties applied if the predicted path is either shorter or longer than the expected length [17]:

$$\mathrm{LS}(R, P) = \frac{\mathrm{EPL}(R, P)}{\mathrm{EPL}(R, P) + |\mathrm{EPL}(R, P) - \mathrm{PL}(P)|} \tag{10}$$

4 Experimental Results

We experimented with different observation types to determine the most suitable image format for our case. AirSim offers seven sensor types, including an RGB camera, barometer, inertial measurement unit (IMU), global positioning system (GPS), magnetometer, distance sensor, and LiDAR. In the first experiment, two agents were trained over 1000 epochs, lasting 14 hours: one using the RGB camera and the other using the RGBD camera. Their Q-value, total reward per epoch, and resulting trajectories were compared to identify the superior sensor for obstacle avoidance. Figures 5 and 6 present the Q-value and total reward per epoch graphs for the RGBD and RGB agents, respectively.

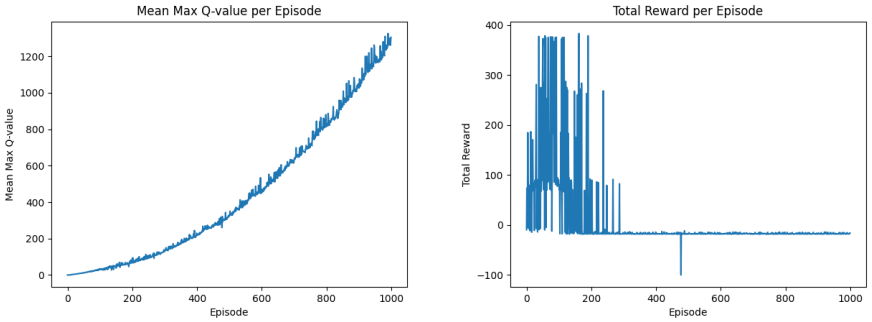

Figure 5: Top-down view of the predefined trajectory with waypoints and the reference plot.

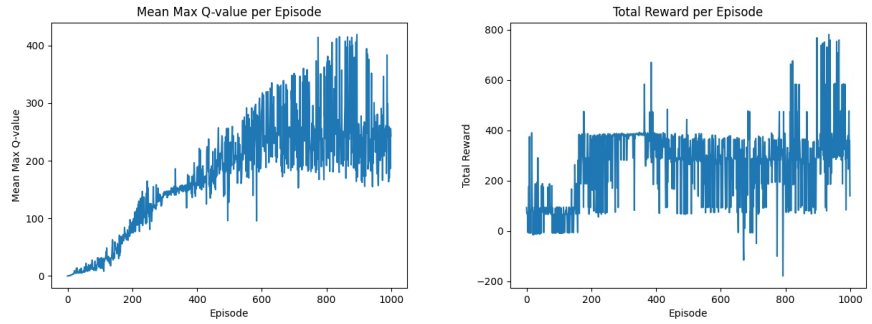

Figure 6: Top-down view of the predefined trajectory with waypoints and the reference plot.

The RGBD agent produced a more stable and steadily increasing Q-value graph compared to the RGB agent. However, it struggled to complete the route beyond the 250th epoch. In contrast, the RGB agent produced a more consistent total reward graph, reflecting the stability of its observation type.

Both agents were unable to complete the route successfully, as demonstrated in Table 1. Additionally, we compared their final trajectories and decided to focus on improving the RGB agent to complete the path successfully. Figure 7 presents the trajectory comparison between these agents.

Table 1: First Experiment Trajectory Results

Model	**SR ↑**	**NE ↓**	**SDTW ↑**	**CLS ↑**
RGBD Agent	0.0	105.99	0.0	0.15
RGB Agent	0.0	66.94	0.0	0.42

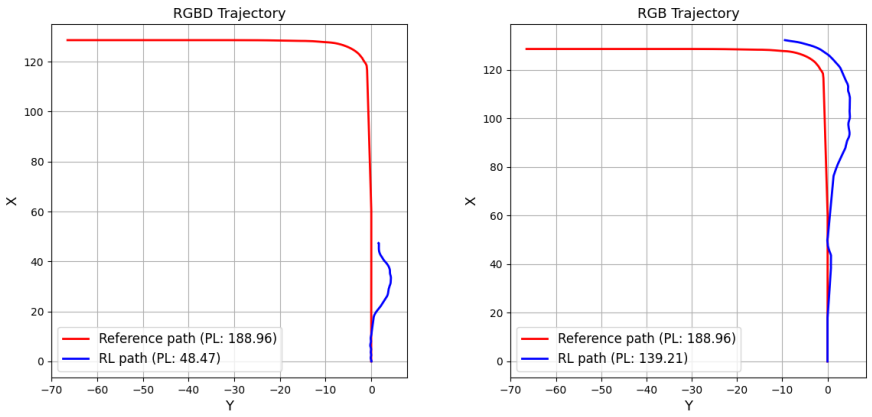

Figure 7: Comparison of Top-down trajectories of RGBD and RGB agent respectively.

Table 2: Second Experiment Trajectory Results

Model	**SR ↑**	**NE ↓**	**SDTW ↑**	**CLS ↑**
Improved RGB Agent	1.0	3.79	0.63	0.6

In the second experiment, we adjusted two key parameters, *epsilon_min*, and *epsilon_decay*, to improve the performance of our agent. By decreasing *epsilon_min*, we reduced the minimum exploration rate, encouraging the agent to exploit its learned policy more effectively. Additionally, reducing *epsilon_decay* accelerated the transition from exploration to exploitation, allowing the agent to rely on its learned behavior earlier in training. These changes were aimed at enhancing the agent's ability to complete the predefined path, to improve its efficiency and reliability in performing the task. The trajectory metrics are shown in Table 2 and Figure 8 presents the top-down route plot.

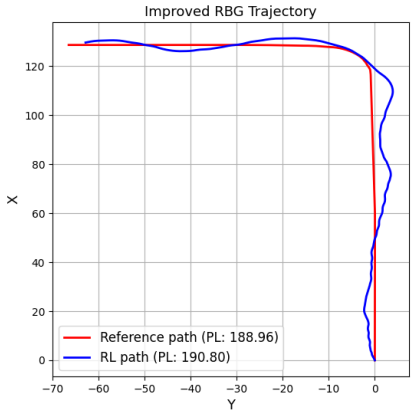

Figure 8: Improved RGB agent's trajectory plot.

5 Conclusion

In this project, we focused on training an agent to successfully navigate a predefined path within a virtual environment, utilizing the Deep Q-Network (DQN) algorithm and the AirSim simulation framework. To enhance the agent's obstacle avoidance capabilities and ensure successful path completion, we conducted experiments with various observation types. Additionally, we modified the epsilon parameters to improve the robustness of the agent, facilitating more effective exploration and exploitation during the learning process.

References

[1] N. S. Vemulapalli, P. Paladugula, G. S. Prabhat, and S. Don, "Reinforcement Learning-Based Autonomous Landing of AirSim Simulated Quadcopter in Unreal Engine," in 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Jun. 2024, pp. 1–7. doi: 10.1109/ICCCNT61001.2024.10725648.

[2] Ho@ng, "Deep Reinforcement Learning for autonomous vehicles with OpenAI Gym, Keras-RL in AirSim simulator," Analytics Vidhya. Accessed: Jan. 03, 2025. [Online]. Available: https://medium.com/analytics-vidhya/deep-reinforcement-learning-for-autonomous-vehicles-with-openai-gym-keras-rl-in-airsim-simulator-196b51f148e4

[3] "bneld/Drone-Hover-RL: A hover system for Microsoft AirSim using reinforcement learning (Q-learning and SARSA)." Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/bneld/Drone-Hover-RL/tree/master

[4] "yujianyuanhaha/AirSim-DQN: Deep Reinforcement Learning for UAV." Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/yujianyuanhaha/AirSim-DQN

[5] "Reinforcement Learning - AirSim." Accessed: Jan. 03, 2025. [Online]. Available: https://microsoft.github.io/AirSim/reinforcement_learning/

[6] "Gazebo," GitHub. Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/gazebosim

[7] facebookresearch/habitat-sim. (Jan. 02, 2025). C++. Meta Research. Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/facebookresearch/habitat-sim

[8] S. James, stepjam/RLBench. (Jan. 03, 2025). Python. Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/stepjam/RLBench

[9] cyberbotics/webots. (Jan. 02, 2025). C++. Cyberbotics Ltd. Accessed: Jan. 03, 2025. [Online]. Available: https://github.com/cyberbotics/webots

[10] "Reinforcement Learning - AirSim." Accessed: Jan. 03, 2025. [Online]. Available: https://microsoft.github.io/AirSim/reinforcement_learning/

[11] E. Balcı, M. Sarıgül, and B. Ata, "Prompting Large Language Models for Aerial Navigation," in 2024 9th International Conference on Computer Science and Engineering (UBMK), Oct. 2024, pp. 304–309. doi: 10.1109/UBMK63289.2024.10773467.

[12] S. Liu, H. Zhang, Y. Qi, P. Wang, Y. Zhang, and Q. Wu, "AerialVLN: Vision-and-Language Navigation for UAVs," Aug. 13, 2023, arXiv: arXiv:2308.06735. doi: 10.48550/arXiv.2308.06735.

[13] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," Jul. 18, 2017, arXiv: arXiv:1705.05065. doi: 10.48550/arXiv.1705.05065.

[14] Epic Games. (2021). Unreal Engine 4 [Online]. Available: https://www.unrealengine.com/zh-CN/

[15] V. Jain, G. Magalhaes, A. Ku, A. Vaswani, E. Ie, and J. Baldridge, "Stay on the Path: Instruction Fidelity in Vision-and-Language Navigation," Jun. 21, 2019, arXiv: arXiv:1905.12255. doi: 10.48550/arXiv.1905.12255.

[16] J. Lin et al., "Advances in Embodied Navigation Using Large Language Models: A Survey," Jun. 07, 2024, arXiv: arXiv:2311.00530. doi: 10.48550/arXiv.2311.00530.

[17] G. Ilharco, V. Jain, A. Ku, E. Ie, and J. Baldridge, "General Evaluation for Instruction Conditioned Navigation using Dynamic Time Warping," Nov. 28, 2019, arXiv: arXiv:1907.05446. doi: 10.48550/arXiv.1907.05446.