

# Design and Implementation of an Autonomous Line-Following Mobile Robot Using Q-Learning and Computer Vision

*K Ch Sekhar*<sup>1\*</sup>, *J Maria Shanthi*<sup>2</sup>, *K R Senthil Kumar*<sup>3</sup>, *Gopikrishnan M*<sup>4</sup>, *Revathi. R*<sup>5</sup>, and *Joshuva Arockia Dhanraj*<sup>6</sup>

<sup>1</sup>Department of Mechanical Engineering, Lendi Institute of Engineering and Technology, Jonnada, Andhra Pradesh 535005, India

<sup>2</sup>Department of Computer Science and Engineering, Anurag University, Hyderabad, Telangana 500088, India

<sup>3</sup>Department of Mechanical Engineering, R.M.K. Engineering College, Kavaraipettai, Tamil Nadu 601206, India

<sup>4</sup>Department of Information Technology, Prathyusha Engineering College, Tiruvallur, Tamil Nadu 602025, India

<sup>5</sup>Department of Computer Science - Artificial Intelligence, PSGR Krishnammal College for Women, Coimbatore, Tamil Nadu 641004, India

<sup>6</sup>Research and Development Cell, Lovely Professional University, Phagwara, Punjab 144411, India

**Abstract.** This research paper discusses the creation and application of a fully automated robot that follows a line by means of reinforcement learning techniques, more specifically the Q-learning algorithm. The system proposed comprises a vision-based perception module that uses OpenCV for its development and also includes the simulation environment of CoppeliaSim thus making the navigation process adaptive as well as robust. By interaction with its surroundings at all times, the agent gets trained on the control policy that is optimal in terms of lateral deviation minimization and hence provides stable trajectory tracking. The research comprises a detailed assessment of the Q-table dimensions and hyper-parameters such as the learning rate, discount factor, and exploration rate, systematically determining their impact on learning performance, convergence, and accuracy. According to the experimental findings, a configuration of a 7×7 Q-table strikes the best balance between precision and convergence speed that in turn results in smooth and even path tracking. The method, though quite effective under controlled conditions, does have its drawbacks in terms of state discretization, generalization, and real-time processing, thereby providing directions for the applications of deep reinforcement learning and adaptive perception models in the future.

---

\* Corresponding author: [sekhar.lendi@gmail.com](mailto:sekhar.lendi@gmail.com)

## 1 Introduction

The domain of mobile robots that can independently navigate has received a lot of attention in the last few years, mainly because of the development of new technologies in AI, computer vision, and control theory. Line following is one of the basic yet difficult tasks in this area that still needs a lot of study; this problem consists of keeping a robot's route on a defined path by using sensors in real time. Up till now, the task has generally been handled by rule-based control or PID control systems, which are effective but often cannot deal with the constantly changing or uncertain environments [1-3].

Reinforcement Learning (RL) is a new option that has a great possibility of being the main method of teaching robots how to create control policies just by interacting with the environment and receiving rewards and punishments as guidance. In contrast to supervised learning, which requires a labeled dataset, RL is about trial-and-error exploration - an agent can discover the best behaviors on its own and without being programmed in a clear way. Q-learning is one of the various RL algorithms; it is known for its easy-to-understand and model-free characteristics along with its ability to work well in discrete decision-making scenario which makes it ideal for robotic control tasks like line tracking [4-6].

A control architecture based on Q-learning has been proposed and implemented in this work to let a mobile robot follow a line in a simulated environment. The CoppeliaSim platform is utilized to simulate the robot's movements and its physical interactions while the OpenCV library performs image acquisition and processing for environmental perception. The robot is trained to keep on the line by adjusting its motor speeds continuously according to the visual input. The adjustment then follows a reward function that penalizes deviations and disconnections [7-10].

This research project has three main goals. The first is to design and implement a reinforcement learning framework that can, through adaptive decision-making and self-learning mechanisms, allow a mobile robot to autonomously perform line-following. The second is to investigate how the Q-table size and the key hyperparameters—learning rate, discount factor, and exploration rate—affect the agent's speed of convergence, stability of learning, and accuracy of the trajectory. Lastly, the project aims to reveal the disadvantages of the tabular Q-learning method thus paving the way for deep reinforcement learning and advanced perception-based control strategies development as future progress areas.

## 2 Literature Review

This section presents the theoretical framework that underpins the construction of the line-following agent. It starts with a discussion of Reinforcement Learning (RL), a model in which the agent acquires the best possible actions by the direct involvement with the environment and by the feedback in the form of rewards. The main building blocks of RL are agent, environment, states, actions, rewards, policy, and value function. A very important point of this procedure is to find the right balance between exploration and exploitation which is usually done by employing the  $\epsilon$ -greedy method [11-12].

RL algorithms are commonly divided into three classes: value-based, policy-based, and actor-critic methods, and based on their use of an explicit environment model, they can be further classified as working with model-free or model-based approaches.

Then, it spotlights on Q-learning, a model-free RL algorithm that iteratively improves action-value estimates to extract a policy. The Q-update rule governs the learning process and includes parameters such as the learning rate ( $\alpha$ ) and the discount factor ( $\gamma$ ). Q-learning is suited for dynamic robotic tasks, as it learns purely from experience without the prior environmental knowledge. Nevertheless, its constraints such as scalability problems, limited generalization across similar states, and overestimation bias have all led to the creation of

variants like Double Q-learning, DQNs, and Dyna-Q, each of which being tailored to deal with a particular specific shortcoming in order to enhance performance and stability [12-14].

Finally, debates the basics of Computer Vision, drawing attention to its crucial role in robot perception. The whole procedure starts with taking a picture, then representation in black and white or RGB formats. The application of preprocessing methods such as the conversion to grayscale, filtering, and binarization is intended to enhance the main features and also to eliminate noise. For detecting lines, the robot focusses on the lower part of the picture where it detects the line's center and also gets its distance from the center of the picture [13-15]. This distance is the feedback signal that lets the robot modify its path, thus, the robot benefits for visual-based autonomous navigation from the perceptual foundation of being able to see.

Reinforcement Learning (RL) allows an agent to learn optimal actions by interacting with the environment and receiving feedback in the form of rewards. Unlike supervised learning, RL relies on trial-and-error exploration, enabling autonomous policy development without explicit programming. In this work, we employ Q-learning, a model-free RL algorithm suited for discrete control tasks such as line following.

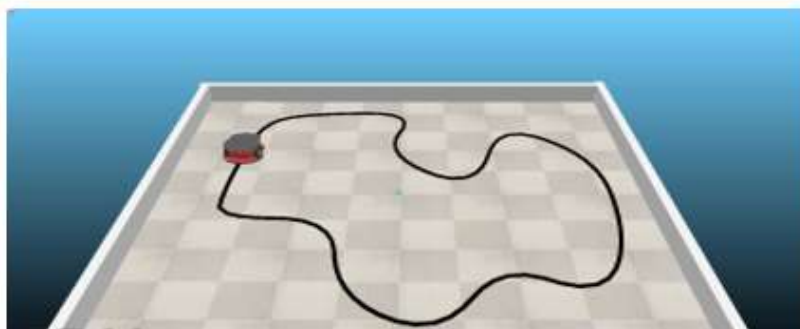
### 3 Development

In this part, the technical realization of the system that was constructed through this project is mentioned along with the main tools and components which make it possible for the robot to teach itself and do the line-following task through reinforcement learning.

#### 3.1 Development Environment

##### 3.1.1 Coppeliasim

Coppeliasim is a simulated environment which is the most significant for complexity of robotic systems development and testing. It is also the best for research and learning in the field of robotics because its built-in physics engine and wide-ranging compatibility with various types of sensors and actuators are features that characterize it. The simulation environment used in this work was tailored for the classic line-following problem. The environment is made of a closed circuit, which is represented by a black line on a light-colored floor, and a Pioneer P3DX robot, which is a compact and lightweight robot having two wheels and two differential drive motors allowing them to rotate at different speeds on the same axis [16-19]. Furthermore, the circuit has a sensor at the starting point that can detect when the robot has completed one lap. Fig.1 shows an outline of the basic training setup for Coppeliasim, where the pioneer p3dx robot is navigating a track marked by a black line.



**Fig. 1.** Basic circuit for agent training in CoppeliaSim.

The ZMQ Remote API serves as the link for communication between the control algorithm and the simulator, making it possible for the Python code to:

- The title of the article
- Initiate, terminate, and observe the simulation status.
- Get sensor information, like the pictures taken by the camera and the signal from the sensor that is located on the circuit.
- Regulate the motor rotation speed of the robot.

### 3.1.2 OpenCV

OpenCV (Open-Source Computer Vision Library) is computer vision and image processing library that is very popular. The library is mainly written in C++, but high-level links for Python make it accessible for very large-scale use. In this project, OpenCV is the basic provider on the robot's perceiving system, as it can even tell the state of the reinforcement learning agent by extracting the important information from the images taken.

The P3DX robot camera is always taking and sending images of its front view. The region of interest (ROI) is extracted from each processed frame, in this case, it is the lower part of the picture. Then a binarization process is followed, and the centroid is found to indicate the robot's location with respect to the center of the line.

The library version used is OpenCV - v4.10.0 [26], which was the latest release available when this work started.

### 3.1.3 Python

Python is the programming language that was chosen for implementing the control system and the reinforcement learning agent. Its adoption for this project was mainly due to its straightforwardness and ability to work seamlessly with CoppeliaSim and OpenCV.

The following are the most important libraries utilized:

- ZMQ RemoteAPIClient - for communicating with CoppeliaSim.
- cv2 (OpenCV) - for processing images.
- numpy - for keeping and updating the Q-table.
- matplotlib - for showing results. The used Python version is 3.9.10

## 3.2 Implementation

### 3.2.1 General System Architecture

The learning system utilized in this research has a modular framework that makes it easy to separate the different parts and supports the independent development, testing, and updating of each module. The following are the three primary modules:

The three main modules are:

- Perception: The module that takes care of receiving images from the robot's camera and extracting useful data from them, which for this case is the robot's location with respect to the line.
- Control: This module decides on the actions (i.e., adjusting the motors' rotation speed) and carries them out based on the current state, which is provided by the perception module.
- Learning: This module is in charge of learning through the application of the Q-learning algorithm, repeatedly updating the value function stored in the Q-table, which maps states to actions.

There is continuous interaction among the modules, which is done in a loop where the perception, control, and learning processes are performed one after the other.

The camera continuously records images at 30 frames per second. The image obtained in every cycle will be directed to the perception module to calculate the line's centroid, which normalizes to represent the robot's offset. The offset will be directed to the control module, which will determine an appropriate action based on the Q-table. The speed required for every wheel will also be calculated. The overall time cycle for this process would be around 50 milliseconds. Therefore, stable real-time control would be possible if the robot's speed is moderate. The control will continuously enable interaction among modules.

### 3.2.2 *Communication with CoppeliaSim*

The ZMQ Remote API is the means through which communication with the CoppeliaSim simulation environment is set up, enabling asynchronous interaction between the Python script running externally and the simulation engine. The control logic for the simulation and robot interaction has been confined to the Coppelia and P3DX classes that are an adaptation of the classes provided by the supervisor of this work as a starting point for developing the reinforcement learning model [28].

The methods provided by these two classes make it possible to:

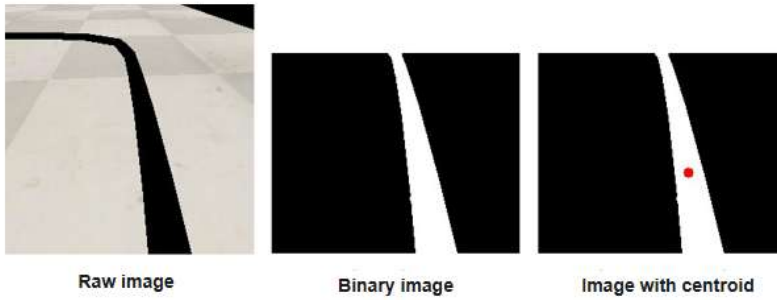
- Connecting with the simulation environment. Simulation starting, stopping, and state controlling.
- Scene component access including the robot camera and motors, and the sensor positioned at the circuit's starting point.

### 3.2.3 *Image Acquisition and Processing*

The robot relies on an onboard camera to monitor the area in front of it where a black line indicates the way to go. To get the most out of the images, a processing workflow using OpenCV is applied:

- Region of interest: The lower part of the image is chosen, and it is the only part that allows detecting the position of the robot in relation to the line.
- Grayscale conversion: The RGB image is converted to grayscale to make the processing of the remaining steps easier.
- Gaussian blurring: The image is smoothed to denoise it and thus avoid interference with line detection.
- Binary segmentation: The image is converted into a representation with two distinct values: white for the line and black for everything else.
- Moment calculation: It finds the centroid of the white area, thereby yielding the coordinates of the line detected.

In case the line is not detected, the system sends a single state back which indicates the line loss. If the line is detected, it returns the centroid position for robot deviation calculation. Fig. 2 depicts an image processing pipeline implemented by OpenCV for image processing applications such as region selection of interest, image binarization, and computation of centroids for line identification.



**Fig. 2.** Processing an image captured with the robot's camera in CoppeliaSim.

### 3.2.4 State Representation

The coding of the robot's position in relation to the line is done as one continuous value and it is normalized horizontal displacement of the centroid of the detected line in comparison to the center of the image that the robot captures. This value is then mapped to the range of  $[-1, 1]$ , in which:

- 0 means that the robot and the line are perfectly aligned
- -1 is the farthest left deviation possible
- 1 is the farthest right deviation possible

In order to adjust the state representation for Q-learning, the continuous horizontal movement is divided into a finite number of states in accordance with the predetermined dimensions of the Q-table. Every single discrete state corresponds to a definite range of the normalized horizontal displacement, thereby implicitly representing both the motion's strength and its direction. The conversion method, illustrated in Listing 4.1, is executed through the formula  $state = \text{int}((\text{offset\_norm} + 1) / (2 / \text{NUM\_STATES}))$ , which translates continuous normalized displacement values into discrete state indices. This technique of dynamic discretization allows the model to easily switch between various levels of resolution or granularity according to the value of the constant NUM\_STATES, which is specified before the model is initialized.

In this framework for reinforcement learning, the system state is represented as the normalized horizontal displacement of the line centroid running across discrete intervals of 'NUM\_STATES.' The action is indicated by motor speed modifications represented as discrete movement commands of 'NUM\_ACTIONS.' The provided framework indicates that the agent receives positive feedback for line follow, characterized as follows:  $\text{reward} = 1 - 2 * |\text{offset}|$  for on-line states, while an additional penalty of -10 is incurred for line loss. The Q-learning update rule is given as follows:  $Q[\text{state}, \text{action}] += \alpha * (\text{reward} + \gamma * \max(Q[\text{new\_state}]) - Q[\text{state}, \text{action}])$ . Thus, improvement is achievable for an agent. The strategy for action selection is provided as an  $\epsilon$  greedy strategy with decay.

### 3.2.5 Action Selection and Application

The robotic system is constrained to a particular range of distinct actions which are indicated by the constant NUM\_ACTIONS. In the most basic setup with three actions, they are basically left turn, forward move and right turn. When the count of actions goes up, it still has to be an odd number to ensure equal turning directions of left and right. Each turning direction is then divided into several intensity levels, which allows better control over the robot's movement. The primary action always stands for moving forward, whereas the other actions are equally spread out to indicate different turning angles in both directions. To keep

the system in sync with the changing number of states and actions, linear interpolation is utilized to map the action indices to the corresponding motor commands.

First, the action index is converted into a proportional value within the interval of  $[-1, 1]$  by means of the formula  $\text{proportion} = (\text{action} / (\text{NUM\_ACTIONS} - 1)) * 2 - 1$ , which expresses the direction and the force of the turn. The left and right wheel motor speeds are then calculated according to the proportion using the formulas  $\text{left\_speed} = \text{base\_speed} - \text{turn\_range} * \text{proportion}$  and  $\text{right\_speed} = \text{base\_speed} + \text{turn\_range} * \text{proportion}$ , and the robot's motor speeds are set through `robot.set_speed(left_speed, right_speed)` in the order mentioned above. This method grants strong and adaptable control thereby keeping performance constant without the need to manually change the motor speeds according to different action configurations.

### 3.2.6 Rewards

The reward function is created in such a way as to bring the agent's lateral deviation from the line's center to a minimum, hence allowing it to make very small movements along the path. The function, which is shown in Listing 4.4, when the robot loses the line ( $\text{state} == -1$ ), gives a heavy penalty of -10, in all other situations the reward is calculated with the help of the formula  $\text{reward} = 1 - 2 * \text{abs}(\text{offset})$ . The latter produces a reward in the area of  $[-1, 1]$ , which becomes less and less as the robot moves away from the line. The arrangement assures that the agent enjoys the maximum reward when it is right on the line, and slowly the rewards grow less as it moves away. Whenever the robot cannot see the line at all, the number of punishments sent out to the agent is high, preventing the agent from repeating such a behavior and making the agent go through the transition to a new episode, thus sending the robot back to where it started. The distribution of rewards has the effect of pushing the agent to persistently remain on the line and at the same time speeding up his learning process by punishing him in failed states quickly.

### 3.2.7 Learning Algorithm: Q-learning

The backbone of the reinforcement learning system is the Q-learning algorithm, which interacts with the environment and slowly but surely improves agent's decision-making strategy. The Q-table is a two-dimensional array  $Q(\text{NUM\_STATES}, \text{NUM\_ACTIONS})$ , which initially consists of zeros and later on is updated gradually according to the formula given in Listing 4.5:  $Q[\text{state}, \text{action}] += \text{ALPHA} * (\text{reward} + \text{GAMMA} * \max(Q[\text{new\_state}]) - Q[\text{state}, \text{action}])$ . This formula uses the parameter ALPHA as the learning rate which decides how much impact new experiences will have on the already existing knowledge, the parameter GAMMA as the discount factor which is responsible for the future rewards' importance, the variable reward as the environment's immediate feedback, and  $\max(Q[\text{new\_state}])$  as the agent's estimation of the best possible return from the next state.

In order to keep a good balance between exploration (implementation of new actions) and exploitation (applying the knowledge that has been learned), an  $\epsilon$ -greedy policy is used, as shown in Listing 4.6. With a probability of  $\epsilon$ , the agent chooses an action randomly, and with a probability of  $(1-\epsilon)$  it selects the action that has the highest current Q-value, thus: `if np.random.rand() < EPSILON: action = random.randint(0, NUM_ACTIONS -1) else: action = np.argmax(Q[state])`. This strategy allows the agent to explore the environment to a large extent during the initial phase of the training and later on to develop the optimal behavior slowly as learning continues. The major hyperparameters—ALPHA, GAMMA, and EPSILON—are the ones that affect the agent's convergence speed, stability, and final performance accuracy most. The tuning of these parameters to the optimal level, which has

a direct correlation with learning efficiency and robustness, is discussed in detail in the next section.

### 3.2.8 Training

The training procedure is divided into several episodes, each of which begins by resetting the robot to its starting position and orientation. A sequential loop utilizing the established modules is performed in each episode. Initially, an image is taken, and the current state is computed. An action is then picked and carried out based on this state. After that, the robot takes the next image to find out the new state, and then the reward is computed based on the new state reached. The Q-table is then updated with this reward, and the loop continues until the robot either loses the line or follows the predefined path. Training goes on until the maximum number of episodes is reached, with the total reward from each episode logged to determine the agent's convergence. Besides, the system keeps track of the robot's average lateral deviation from the line in every episode, which is the main performance indicator used to evaluate the accuracy of navigation.

## 4 Results

Here, the results of the training and evaluation process for the reinforcement learning agent are presented. The Q-table dimensions and hyperparameter values were varied and analyzed to study the impact on the model's learning, convergence, accuracy and robustness systematically in the experiments.

Each episode is regarded as a single trial, starting with the robot at the circuit's start point and stopping when it either finishes a lap or loses the line. The agent's performance is mainly indicated by the mean normalized horizontal displacement per episode. This value, 0 to 1, shows how well the agent follows the line, 0 being perfect alignment. This performance measure is used instead of total reward because it gives an easily understandable value which is not affected by factors like robot speed that changes the total reward in each episode hence making it difficult to compare the performance across different setups.

The next sections first illustrate the influence of Q-table dimensions, then discuss the effect of hyperparameters, provide a summary and comparison of the best configurations, and finally, address the limitations encountered.

### 4.1 Learning Convergence Across Episodes

Reward progression and mean lateral deviation were analyzed over training episodes to evaluate convergence. Fig. 8 illustrates the reward and displacement curves for the  $7 \times 7$  Q-table. The agent achieves stable behavior by episode 65, with a consistent reduction in lateral deviation and increase in cumulative reward. Lower-resolution Q-tables ( $3 \times 3$  and  $5 \times 5$ ) converge faster (within 2–15 episodes), but produce oscillatory paths with higher mean deviation. The  $9 \times 9$  configuration shows slower convergence and occasional instability due to fine discretization, demonstrating the trade-off between resolution and stability.

Fig. 3 shows Reward progression and mean lateral deviation per episode for the  $7 \times 7$  Q-table, showing convergence after 65 episodes.

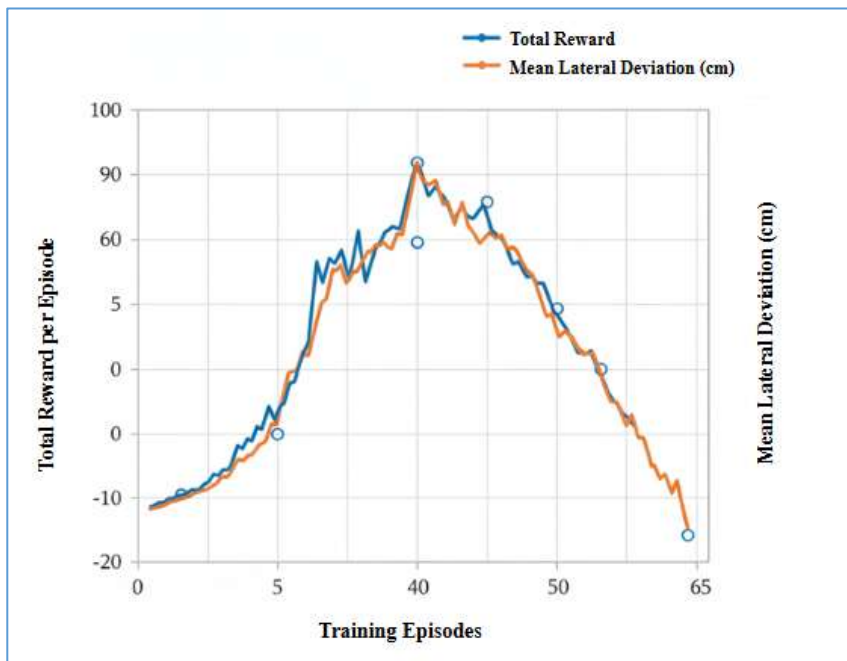


Fig. 3. Reward Progression and lateral Deviation (7x7 Q-Table)

## 4.2 Evaluation of Q-table Size

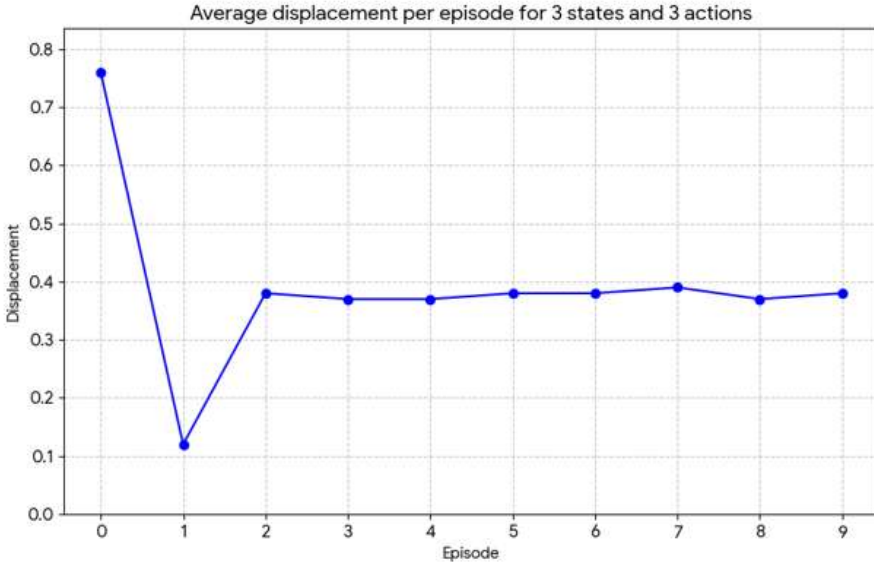
A comparative analysis of different Q-table sizes is presented below, maintaining an odd number of states and actions to preserve symmetry between the left and right spaces. The objective is to evaluate the impact of state-action space resolution on the learning process, highlighting the advantages and disadvantages offered by each configuration.

Although this section focuses on analyzing the impact of Q-table dimensions on the learning process, the most appropriate hyperparameters are applied for each configuration to provide a rigorous comparison, since keeping them static would lead to comparing policies with different levels of maturity. Furthermore, results may vary slightly between training sessions, given the exploratory nature of the learning process. Therefore, for each configuration, the best results obtained after ten complete training sessions are presented, and the average performance is also considered when selecting the results to avoid presenting outliers.

### 4.2.1 3x3

The three-state, three-action model displays almost instant convergence, with a stable policy being reached in just two episodes. The policy obtained, however, is rough, with the final mean displacement being around 0.38, and it also generates paths that are clearly oscillatory. Since the Q-table only contains nine state-action pairs, the search area is rapidly filled without the use of an exploration strategy. The default values of 0.2 for the learning rate and 0.8 for the discount factor were kept and their effect at this level was found to be very limited.

The roughness and the lack of smoothness in controlling the robot made it clear that the resolutions in perception and action space had to be increased. Similarly, the average normalized horizontal displacement per episode for the 3x3 Q-table configuration is shown in Figure 4 and exemplifies the oscillatory nature of the paths.

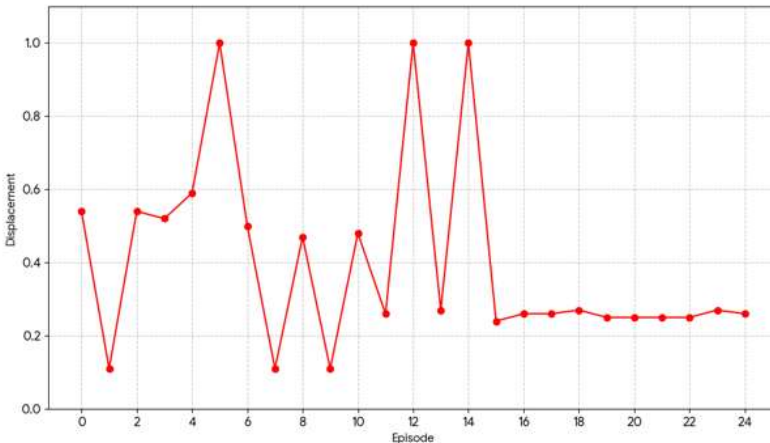


**Fig. 4.** Average normalized horizontal displacement per episode for Q-table.

#### 4.2.2 5x5

This new setup was a huge step forward compared to the previous one, and it managed to get the best possible trade-off between accuracy and speed of convergence. The agent came up with a stable control policy within fifteen episodes and at the same time, the mean offset was reduced significantly to about 0.23. The standard hyperparameter values  $\alpha = 0.2$  and  $\gamma = 0.8$  that had been used in the 3x3 configuration were also kept, and they turned out to be equally efficient in this case. Moreover, the lack of exploration once again gave the best performance, the reason being that the Q-table was still small in size.

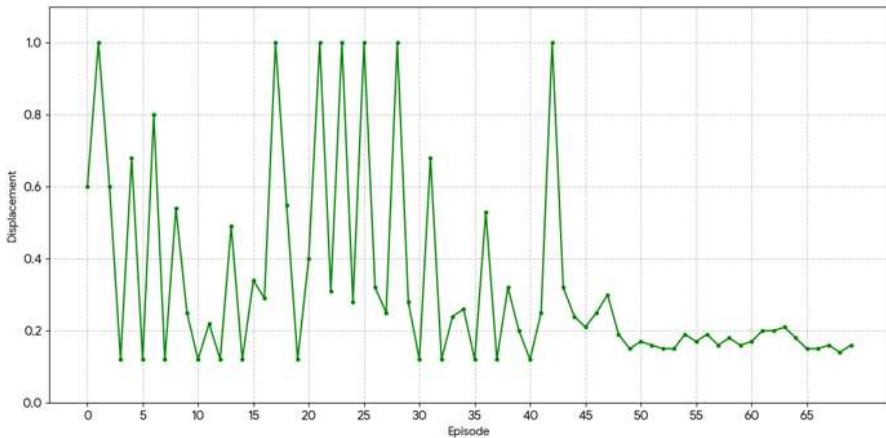
The 5x5 layout showed steady performance all over the circuit, and it was able to provide a great deal of control sharpness without causing too much training time. The 3x3 table had accuracy increased by 40% and at the same time, the ease of parameter tuning was retained. Fig. 5 depicts the average normalized horizontal displacement per episode for the 5x5 Q-table. It shows the progress and convergence of the horizontal movement of an agent.



**Fig. 5.** Average Normalized Horizontal Displacement per Episode for Q-table 5x5.

### 4.2.3 7x7

Two additional states along with seven actions not only contributed to the enhancement of the robot's control but also succeeded in displacing to only 0.16 on average. Notwithstanding, a notable rise in the convergence time was one of the side effects of this, which meant that 50 episodes were necessary for the policy to at least start stabilizing and a total of 65 to attain maximum accuracy. This setup was different from the previous ones in that it was very much dependent on hyperparameter tuning. Therefore, it became mandatory to bring about some amount of exploration to keep convergence from leading to policies of lesser quality, besides, learning rate decay was also required to clear up the noise and deliver greater stability in the higher episodes. Average displacement per episode for 7 states and 7 actions is shown in Fig. 6.



**Fig. 6.** Average displacement per episode for 7 states and 7 actions.

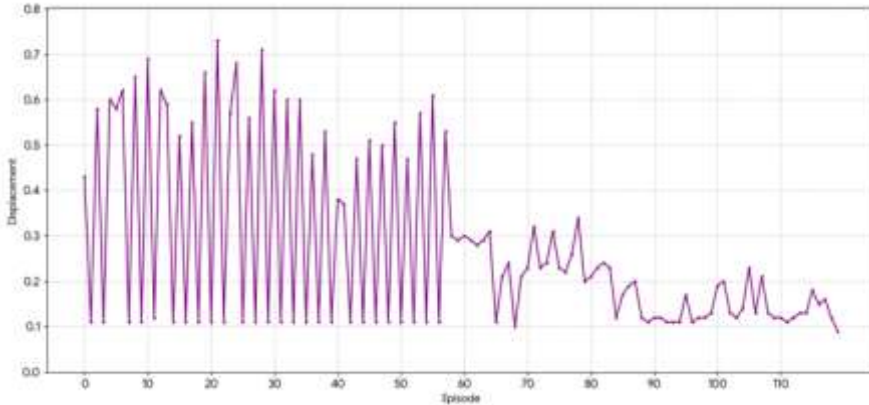
### 4.2.4 9x9

Two additional states along with seven actions not only contributed to the enhancement of the robot's control but also succeeded in displacing to only 0.16 on average. Notwithstanding, a notable rise in the convergence time was one of the side effects of this, which meant that 50.

The 9x9 configuration began to reveal the limitations of excessive Q-table resolution. Although some convergence was observed after 90 or 100 episodes, the resulting policy lacked consistency. While some series of episodes achieved a very small average shift, with a value of approximately 0.12, others showed a noticeable decrease in accuracy or even ended with the complete loss of the line.

Despite introducing decay in the learning and exploration rates, increasing the discount factor to facilitate long-term policy consistency, and increasing the training time to a maximum of 300 episodes, a more stable policy could not be obtained. The agent remained vulnerable to noisy updates and policy degradation, particularly on very steep curves.

This is mainly due to the sensitivity associated with the fine discretization of the state space. As the number of states increases, small variations in visual perception, such as shifts of just a few pixels from the centroid, can result in a transition to a different state associated with a different or insufficiently trained action. This fragmentation prevents the established policy from correctly generalizing between very similar states, leading to inconsistent decision-making. Average normalized horizontal shift per episode for Q-table is shown in Fig. 7.



**Fig. 7.** Average normalized horizontal shift per episode for Q-table.

#### 4.2.5 7x5

This asymmetric configuration was introduced to explore a possible balance between the two best-performing policies; however, it did not provide any measurable advantage. The training time was comparable to that of the 7x7 Q-table, while the maximum accuracy was analogous to the 5x5 configuration. This result highlighted a fundamental design principle: greater granularity in perception cannot be leveraged if the action space lacks corresponding expressiveness. The agent is able to detect more subtle deviations, but is unable to respond with the appropriate level of control.

### 4.3 Hyperparameter Tuning

Getting the hyperparameters properly adjusted is crucial in obtaining stable and efficient learning, particularly when dealing with high-resolution Q-tables. This part of the text reviews the role played by the cooling, the learning rate, and the discount factor in the learning process.

#### 4.3.1 Exploration Strategy

For low-resolution tables (3x3 and 5x5), exploration is not only unnecessary but also counterproductive. The small number of state-action pairs leads to automatic coverage of all Q-table entries, so selecting greedy actions ( $\epsilon = 0$ ) achieves rapid and stable convergence. Unnecessarily introducing exploration causes erratic agent behavior in low-resolution configurations, as inappropriate actions are frequently selected.

On the other hand, the use of higher-resolution tables (7x7 and 9x9) imposes a need for a scanning strategy so that agents will not fall into suboptimal policies through convergence. In these kinds of settings, the total absence of scanning may lead to the agent carrying out a poor practice, for instance, by always keeping the line within one side of the field of view. A positive scanning rate brings about random deviations, which ensures that every state-action pair is at least visited and evaluated at one time or another.

For 7x7 table, a steady value  $\epsilon = 0.01$  was adequate to direct the agent towards the optimal policy in case of the 9x9 table. However, the more demanding configuration warranted an even more advanced strategy: epsilon-decay. It is a practice where the training is started with a high exploration rate (e.g.,  $\epsilon = 0.2$ ) that will be progressively reduced as the

policy matures, thereby allowing for the coverage of most of the 81 states during the initial episodes to avoid suboptimal convergence. This also means that the learned policy will be exploited as training progresses, thus preventing erratic behavior and the overwriting of established values.

### 4.3.2 Learning Rate and Discount Factor

The learning rate affected the performance very little at low Q-table resolutions, where the number of entries quickly gets filled up. In this situation, the policy is quickly stabilized, and the environment's simplicity lowers the chance of good values getting overwritten. On the other hand, when the state and action space increases (7x7 and 9x9), the learning rate becomes a crucial factor for the stability of the learning process and the retention of the policy. As  $\alpha$  is constant in these settings, the agent gets more inclined to forget what it learned formerly. Such a process happens in closed loops primarily, where very small changes in the input path or in the state's perception can cause undesirable updates in almost the same states.

To solve this issue, alpha-decay was applied, which involved gradually lowering the learning rate from 0.4 to 0.05. This tactic has two major advantages: it makes the learning process faster during the first few episodes and it also stabilizes the Q-values during the later training stages thereby making the degradation of the effective policies slower.

The adjustment of the discount factor, though not as crucial as the learning rate, still had its part in the creation of an optimal policy. The larger configurations took advantage of the increase in the discount factor, which allowed for the coordination of longer action sequences. This was very helpful especially in the case of sharp curves, where the agent displayed a minor improvement in consistency and fluency when passing through them, by giving more importance to continuous control rather than short-term corrections.

To put it briefly,  $\alpha$  the learning rate and  $\gamma$  the discount factor have complementary roles, where:

- $\alpha$  controls the rate and stability of the learning
- $\gamma$  determines the depth of the learned policy in time

## 4.4 Optimal Configurations and Trade-offs

Presented in the ensuing Table 1 are the optimal Q-table and hyperparameter configurations, as well as the average shift obtained and the episodes needed for convergence:

**Table 1.** Comparative Table of Q-table and Hyperparameter Configurations.

Q-table	$\epsilon$	$\alpha$	$\gamma$	Average displacement	Convergence episodes
3x3	0	0.2	0.8	0.38	2
5x5	0	0.2	0.8	0.23	15
7x7	0.01	0.4 $\rightarrow$ 0.05	0.9	0.16	65
9x9	0.2 $\rightarrow$ 0.01	0.4 $\rightarrow$ 0.05	0.95	$\sim$ 0.12 (unstable)	120+

The 3x3 initial point quickly converges but this is accompanied by low precision and a clearly oscillatory path.

The 5x5 setup presents a nice compromise regarding the three aspects of convergence speed, accuracy, and stability thus evidencing high consistency all through the circuit and low sensitivity to hyperparameter tuning.

The 7x7 grid is the one that gets the highest stable accuracy in a reasonable training time but it requires thorough hyperparameter tuning to make sure it works properly.

The 9x9 layout sets the limit for Q-table resolution in this line-tracking challenge, showing convergence and stability problems.

Ultimately, the choice of the right configuration is a matter of the specific problem and the importance of performance metrics that are relative to that problem must be viewed against the background of the particular application. In this case, the 7x7 configuration is the best choice, as it is able to provide the most accurate result within the limits of the desired convergence time which is still the fastest among the others.



**Fig. 8.** Robot path on the training circuit with a 7x7 Q-table.

The above Fig. 8 depicts the robot using the most successful configuration (7x7 Q-table) in the training area. The agent's route in one normal episode is shown through a red trail for better visualization. It shows the agent's capability of being closely aligned with the line all the time, even when making curves and with little lateral movement.

Figure 8 shows the progression of the rewards over 65 episodes for a 7x7 Q-Table. This shows that the agent converges to a policy around episode 65. Additionally, the average lateral deviation reduces correspondingly. This shows that the accuracy of the agent in following a line improves.

#### 4.5 Multi-Track Experimental Validation

Quantifying generalizability, an evaluation was performed across different track configurations. The track shapes considered were circular track, oval track, and an eight-track. Comparative measures were used, and results were tabulated in Table 2. These measures were used for comparison based on mean deviation, success rate, and completion time. The  $7 \times 7$  Q-table displayed lower mean deviation per track, which was around 0.16.

**Table 2.** Performance metrics across multiple track configurations.

Track Type	Mean Deviation	Success Rate	Completion Time (s)
Circular	0.16	100%	32.4
Oval	0.17	95%	34.1
Eight-track	0.18	93%	36.2

## 4.6 Baseline Comparisons

For the benchmarking, the  $7 \times 7$  Q-learning agent was implemented and compared with the PID algorithm and the rule-based line-following algorithm. From the results provided in Table 3, the Q-learning agent is superior to the existing controllers in terms of accuracy and smoothness, especially for curves and intersections.

**Table 3.** Comparison of Q-learning with baseline controllers.

Method	Mean Deviation	Success Rate	Notes
Q-learning	0.16	95–100%	Smooth trajectory, adaptive
PID	0.22	90–95%	Moderate oscillations in curves
Rule-based	0.31	85–90%	Large oscillations and delays

## 4.7 Limitations

This Q-learning implementation has been shown to be quite effective in the controlled environment of training and validation. Yet, some limitations still hinder its robustness and generalization. The following points are made considering the theoretical analysis and empirical tests during which a pre-trained  $5 \times 5$  Q-table was utilized to quantify the shortcomings of the developed agent. The proposed agent portrays steadfast performance in line tracking, even with moderate changes in lighting or line thickness. However, performance declines with low contrast or reflective situations, suggesting directions for future improvements that may involve employing techniques such as thresholding or deep learning to perception.

### 4.7.1 Discretization

One more important drawback of the algorithm that was put into practice is its exclusively reactive character, whereby decisions are made for each situation based only on the current perception, thus not possessing any prior knowledge of observations or future predictions. This limitation of temporal depth restricts the agent to using policies that react only and very quickly to local data.

In line following, such a policy can be adequate and can lead to very accurate and stable navigation; nevertheless, it may also restrict the agent when it comes to dealing with unclear situations, for example, recovering from an off-track situation or briefly losing track of the line.

### 4.7.2 Temporal Depth

Another significant limitation of the implemented algorithm lies in its purely reactive nature, where each decision is made solely based on the perceived state at any given moment, lacking any knowledge of past observations or future predictions. This lack of temporal depth limits the agent to employing policies that respond exclusively and immediately to local information.

In the context of line following, this type of policy is sufficient to achieve highly accurate and stable navigation; however, it can limit the agent's ability to handle ambiguous scenarios, such as recovering from a deviation or momentarily losing sight of the line.

### 4.7.3 Temporal Depth

In order to assess the durability and universal application of the acquired policy, a series of trials were performed with a previously trained 5x5 Q-table, which was chosen for its easy handling and steady performance. The agent shows an incredible capability of indulging some environment alterations:

- It manages line thickness variations with no loss of navigation precision.
- It changes smoothly with the variations in traffic speed, provided that the speed does not exceed a certain limit, beyond which the agent does not have enough time to understand the environment and act accordingly.

Nevertheless, there are different situations where the agent's operation is not correct:

The system still has decent performance with some modifications in line color, as long as there is a good contrast with the background. Nevertheless, light colors create a big issue with the detection of the line, revealing the limit of the perception system's sensitivity to the contrast between the line and the ground.

Intersections, which are points where two lines cross at an angle, bring about the erratic behavior of the agent that cannot correctly and consistently determine the direction to follow, which proves the limitation of classic Q-learning to uncover complex patterns.

### 4.7.4 Real-Time Processing

Finally, another practical aspect to consider is the limitation that comes with the real-time execution model. In spite of the fact that Q-learning is very minimally demanding in terms of computation, the agent's output is still limited to the cycling control frequency and the camera's refreshing rate. When operating at high speeds, the agent needs to make really rapid decisions in order to keep the control stable. So, if the delay in the control program is too high, even the best policy will unavoidably break down.

Thus, the maximum effective speed is not only dependent on policy precision but also on the system's computational latency which has to be taken into account when putting reinforcement learning agents on real robots.

## 5 Conclusions

This work has investigated the use of reinforcement learning techniques for controlling a mobile robot following a line, where the Q-learning algorithm was the main one used. In addition, computer vision modules that were based on OpenCV were integrated into the processing of environmental information and the discretization of the state space, which made it possible for the agent to interact with its environment through a continuous trial-and-error process.

The CoppeliaSim simulation environment was used for the experimentation with different Q-table configurations, each one accompanied by a systematic analysis of the hyperparameters' impact: learning rate, discount factor, and exploration rate. The results showed that the agent was capable of learning stable and effective policies even with the use of relatively simple configurations. Furthermore, the rewards given to the agent through the selected reward function were successfully aligned with the agent's goal of accurately following the line.

On the one hand, the main goals of the work have been reached, but on the other hand, several limitations that were identified were high sensitivity to the discretization of the state space, and limited capacity for generalization in more complex environments. These results suggest that tabular Q-learning, despite its validity and computational efficiency, might still turn out to be insufficient for more complex robot tasks.

## References

1. Y. Yin, Z. Chen, G. Liu, J. Yin, J. Guo, Autonomous navigation of mobile robots in unknown environments using off-policy reinforcement learning with curriculum learning. *Expert Syst. Appl.* 247, 123202 (2024).
2. K. R. Kunduru, Y. D. Dwivedi, R. Aruna, G. R. Thippeswamy, S. Selvakumar, M. Sudhakar, Elevating performance for enhancing AI-powered humanoid robots through innovation. In: *Applied AI and Humanoid Robotics for the Ultra-Smart Cyberspace*; IGI Global, pp. 85–119 (2023).
3. M. N. Ab Wahab, A. Nazir, A. Khalil, W. J. Ho, M. F. Akbar, M. H. M. Noor, A. S. A. Mohamed, Improved genetic algorithm for mobile robot path planning in static environments. *Expert Syst. Appl.* 249, 123762 (2024).
4. R. Usha, R.S. Selvan, A.B. Reddy, P. Chandrakanth, Development of CNN Model to Avoid the Food Spoiling Level. In: *2023 International Conference on New Frontiers in Communication, Automation, Management and Security (ICAMS)*, vol. 1, pp. 1–7. IEEE (2023).
5. T. R. Saravanan, S. Suvitha, D. Banavath, S. Gogula, J. Upadhyay, M. Sudhakar, AI and machine learning integration in medical assistive robotics. In: *Fostering Cross-Industry Sustainability with Intelligent Technologies*; IGI Global, pp. 130–151 (2023).
6. P. Chandra Kanth, M.S. Anbarasi, A generic framework for data analysis in privacy-preserving data mining. In: *Computational Intelligence in Data Mining*, pp. 653–661. Springer Singapore, Singapore (2019).
7. D. Dobriborsci, R. Zashchitin, M. Kakanov, W. Aumer, P. Osinenko, Predictive reinforcement learning: Map-less navigation method for mobile robot. *J. Intell. Manuf.* 35, 4217–4232 (2024).
8. R. Chitharaj, H. Perumal, M. Almeshaal, P. Manoj Kumar, Optimizing performance of a solar flat plate collector using Box–Behnken design. *Sustainability* 17, 461 (2025).
9. R. Raj Mohan, R. Sharma, R. Venkatraman, Ankit, S. Raghuraman, A. Sarojwal, P. M. Kumar, S. Rajkumar, Influence of planetary ball mill parameters on powder flowability of AlSi10Mg with niobium carbide using central composite design (CCD). *Adv. Mater. Sci. Eng.* 2869225 (2022).
10. J. Huang, Z. Zhang, X. Ruan, An improved Dyna-Q algorithm inspired by the forward prediction mechanism in the rat brain for mobile robot path planning. *Biomimetics* 9, 315 (2024).
11. L. M. Amaya-Mejía, M. Ghita, J. Dentler, M. Olivares-Mendez, C. Martinez, Visual servoing for robotic on-orbit servicing: A survey. In *Proc. Int. Conf. Space Robot. (iSpaRo)*, Luxembourg, 24–27 June (2024).
12. C. A. Contreras, A. Rastegarpanah, M. Chiou, R. Stolkin, A mini-review on mobile manipulators with variable autonomy. *Front. Robot. AI* 12, 1540476 (2025).
13. M. Montero-Vega, M. Estrada, M. Prouvier, A. Siebeneich, Autonomous delivery robots: Differences in consumer’s acceptance across regions. *J. Urban Mobil.* 7, 100110 (2025).
14. Z. Qian, Y. Dong, Y. Hou, H. Zhang, S. Fan, H. Zhong, A geometric approach for homography-based visual servo control of underactuated UAVs. *Meas. Control* 57, 1513–1523 (2024).
15. X. Chen, M. E. Kiziroglou, E. M. Yeatman, Onboard visual micro-servoing on robotic surgery tools. *Microsyst. Nanoeng.* 11, 112 (2025).

16. C. Lakshmanpriya, A. Kumaravel, M. Saravanan, P. Manoj Kumar, Selecting the optimal green supplier and order allocation under linear discount. *Math. Probl. Eng.* 2453703 (2022).
17. K. B. Prakash, A. Amarkarthik, M. Ravikumar, P. Manoj Kumar, S. Jegadheeswaran, Optimizing performance characteristics of blower for combustion process using Taguchi based grey relational analysis. In: *International Conference on Advances in Materials Research*, pp. 155–163 (2019).
18. M. Balaji, S. N. Dinesh, S. V. Vetrivel, P. M. Kumar, R. Subbiah, Augmenting agility in production flow through ANP. *Mater. Today Proc.* 47, 5308–5312 (2021).
19. T. Ravichandra, P. Murugeswari, M. Revathi, S. Senthilkumar, D. S. Rathore, M. Sudhakar, Future of machine learning and robotics in digital technology for hospitality. In: *Cutting-Edge Technologies for Business Sectors*; IGI Global, pp. 401–428 (2023).