

# Dynamic Obstacle Avoidance Using Q-Learning and Deep Q-Learning Techniques

Pranav ANV

*Maryland Applied Graduate Engineering*  
*University of Maryland*  
College Park, USA  
anvpran@umd.edu

Manoj Kumar Selvaraj

*Maryland Applied Graduate Engineering*  
*University of Maryland*  
College Park, USA  
manojks@umd.edu

**Abstract**—Dynamic obstacle avoidance is a critical component of autonomous navigation systems, with significant applications in areas such as autonomous driving. This paper explores the implementation of Q-learning and extends it to Deep Q-Learning for dynamic obstacle avoidance. The primary objective is to navigate a robot through a dynamic environment, aiming to reach from the start to the goal point using Q-Learning and Deep Q-Learning. Upon completion, the results obtained from both methods will be compared. Python will serve as the programming language for the implementation, while Pygame will be used as the simulation platform. For Deep Q-Learning, the necessary neural network models are created using PyTorch, which is a popular deep learning framework. This research serves as a foundational step towards developing more sophisticated autonomous driving systems capable of real-time decision-making in complex and dynamic environments.

**Keywords**—Dynamic obstacles, Q-Learning, Deep Q-Learning, Ray sensor.

## I. INTRODUCTION

The rapid advancement of autonomous systems has brought forth significant interest in developing robust obstacle avoidance techniques, essential for applications such as autonomous driving. Autonomous vehicles must navigate dynamic environments, avoiding collisions while reaching their destinations efficiently. This paper investigates the application of Q-learning and Deep Q-learning for dynamic obstacle avoidance, focusing on training an agent to navigate a grid environment with static and moving obstacles. In Q-Learning, the agent's navigation method is enhanced by its constant updating of its knowledge base, which makes it ideal for settings that are unpredictable and dynamic. The extension to Deep Q-Learning leverages neural networks to handle high-dimensional state spaces, further enhancing the agent's capability to generalize from past experiences to new, unseen scenarios.

The objective is to navigate a robot through a dynamic environment, aiming to reach from the start to the goal point using Q-Learning and Deep Q-Learning. Upon completion, the results obtained from both methods will be compared. Python will serve as the programming language for the implementation, while Pygame will be used as the simulation platform. For Deep Q-Learning, the necessary neural network models will be created using PyTorch, a popular deep learning framework.

This study provides insights into the potential of reinforcement learning for enhancing the safety and efficiency of autonomous navigation systems. By simulating various scenarios, the research demonstrates the feasibility of applying these techniques to real-world problems such as autonomous driving, where vehicles must adapt to everchanging surroundings and obstacles.

## II. LITERATURE SURVEY

The paper "Q-Learning for Autonomous Mobile Robot Obstacle Avoidance" by Ribeiro describes the approach of using Q-Learning, a reinforcement learning technique, to tackle the autonomous mobile robot obstacle avoidance problem. The authors implemented and compared two different Q-Learning methods on a simulated Bot'n Roll ONE, a robot navigating increasingly complex mazes using only infrared sensors. They performed extensive hyperparameter tuning and studied exploration/exploitation strategies like epsilon-greedy and optimistic initial values. The results showed could successfully learn policies to solve all three mazes efficiently, while the other proposed method struggled with the more complex mazes. The work highlighted the potential of Q-Learning algorithms for developing intelligent obstacle avoidance behaviors in robots with limited sensory inputs.

The paper by Beakcheol Jang and Myeonghwi Kim provides a comprehensive review and classification of Q-learning algorithms, which are a family of off-policy reinforcement learning techniques. The authors first explain the background and mathematical foundations of Q-learning, including Markov decision processes, value functions, and the Bellman equation. They then classify Q-learning algorithms into two main categories: single-agent and multi-agent. For single-agent algorithms, they describe prominent approaches like deep Q-learning, hierarchical Q-learning, double Q-learning, and others. For multi-agent settings, they cover methods such as modular Q-learning, Q-learning and swarm-based Q-learning. The paper also investigates recent research trends aimed at improving various aspects of Q-learning, as well as reviewing key application areas where Q-learning has been successfully applied, including industrial process control, computer networking, game theory, robotics, operations research, and artificial intelligence tasks like image classification. Overall, this comprehensive survey highlights the evolution, variants, and widespread adoption of Q-learning across diverse domains.

The paper “Mapless Navigation Based on Continuous Deep Reinforcement Learning” presents a novel approach to robot navigation without relying on pre-defined maps. By using Proximal Policy Optimization (PPO), the method addresses the shortcomings of traditional navigation techniques in dynamic environments. The study found that PPO significantly outperforms the discrete Deep Q-Network (DQN) in both training efficiency and success rate within the Gazebo simulation environment. The PPO-trained model demonstrated effective obstacle avoidance and navigation when transferred to a real robot without additional training, achieving high success rates for both single-target and multi-target navigation. The findings highlight PPO’s potential in enhancing mobile robot navigation and obstacle avoidance.

The research paper “Robot Path Planning Based on Deep Reinforcement Learning” integrates deep learning with reinforcement learning to address the limitations of traditional Q-learning in complex environments. The proposed method replaces the large Q-value table with a convolutional neural network, improving convergence speed and handling dimensionality issues. The introduction of memory matrices and loss functions helps avoid local optima and reduce errors. Simulation results in MATLAB demonstrate the deep Q-learning algorithm’s ability to efficiently plan collision-free, optimal paths for robots, showcasing its potential for practical applications.

Various algorithms, including Q-Learning, Deep Q-Learning, and Proximal Policy Optimization (PPO), have been evaluated for their applicability in addressing the challenges posed by dynamic obstacles. Through comparative analyses and reviews of previous works such as Ribeiro’s exploration of Q-Learning for robot obstacle avoidance and studies on mapless navigation using continuous deep reinforcement learning, insights have been gained into the strengths and limitations of each approach. Ultimately, the decision to focus on Q-Learning and Deep Q-Learning was influenced by their demonstrated effectiveness in similar scenarios, their adaptability to dynamic environments, and their potential for real-time decision-making.

### III. METHODOLOGY

#### A. Environment

The simulation environment is a 500x500 pixel grid divided into 25x25 cells. Each cell can be occupied by the agent, obstacles, or the target. The environment consists of:

##### 1. Static Obstacles:

Static obstacles are fixed in place and do not move throughout the simulation. They represent permanent fixtures in the environment, such as walls or large objects, that the agent must navigate around.

##### 2. Dynamic Obstacles:

Dynamic obstacles move within the environment, changing positions randomly or following predefined paths. These obstacles add complexity to the simulation by requiring the agent to constantly update its path planning to avoid collisions.

#### 3. Ray Sensor:

The agent is equipped with ray sensors that detect obstacles and the target within a certain range (typically next grid). These sensors provide the agent with real-time information about its surroundings, enabling it to make informed decisions about its movements. Each sensor emits rays that return information about the distance to the nearest object in their path.

#### 4. Target:

The destination the agent must reach. Once the agent reaches the target, the target position is changed randomly within the grid.

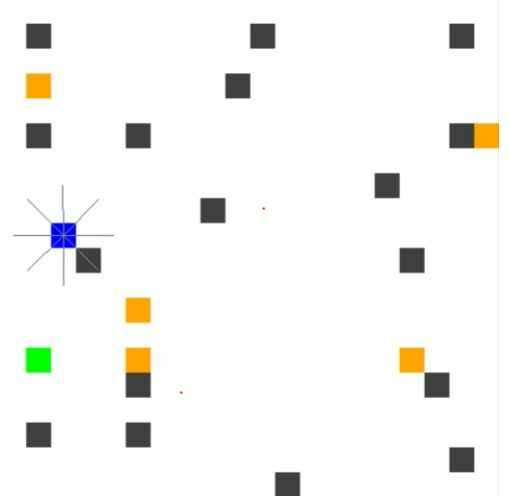


Fig. 1. Environment

From Fig.1. the green target is the destination the agent must reach, while the blue box represents the agent equipped with ray sensors for perceiving its surroundings. Static obstacles, depicted in grey, are immovable elements that obstruct the agent’s path, requiring strategic navigation around them. Dynamic obstacles, illustrated in orange, add complexity by moving within the environment, prompting the agent to continuously adjust its path planning to avoid collisions. The methodologies used for Q-Learning and Deep Q-learning are explained below.

### IV. Q-LEARNING

Q-learning is a model-free reinforcement learning algorithm that seeks to find the best action to take given the current state. It’s considered model-free because it doesn’t require a model of the environment, that is, it doesn’t need to know the transition probabilities from one state to another, which would be the case in model-based methods. Instead, it aims to learn the value of an action in a particular state based on the reward outcome, which it experiences directly by interacting with the environment.

By utilizing observable rewards and transitions to update Q-values, the robot agent in Q-Learning determines the best course of action. By continuously updating its knowledge base, the agent improves its navigation strategy, making it well-suited for dynamic and unpredictable environments. The state space represents the location and orientation of the robot, and movement commands are reflected in the actions. Through iterative exploration and updates, Q-values allow the agent to

travel towards goals while avoiding barriers by predicting the expected payoff for specific actions from given states.

The Q-learning algorithm follows these steps:

- 1.Initialization:** Initialize the Q-table with zeros.
- 2.State Observation:** Observe the current state.
- 3.Action Selection:** Choose an action based on the epsilon-greedy policy.
- 4.Action Execution:** Execute the chosen action and observe the reward and next state.
- 5.Q-Value Update:** Update the Q-value using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- 6.State Transition:** Transition to the next state and repeat the process until the goal is reached or a termination condition is met.


	1	2	3	4
1				✓
2				✗
3				

Fig. 2. An example of Q-Learning in a grid environment

## V. IMPLEMENTATION OF Q-LEARNING

### A. Environment

The Game Environment class manages the grid, including the agent, obstacles, and target, with functions for:

- Target repositioning.
- Moving dynamic obstacles.
- Reward calculation based on the agent's position relative to the target and obstacles.

### B. Details of Implementation

**Initialization:** The *Q-Learning Agent* class is created with its own Q-table that stores the estimated rewards for state- action pairs. This Q-table is initialized to zeros, allowing the agent to learn and update values based on interactions with the environment.

**State Representation:** The *sense environment* method generates a state representation from the agent's perspective, combining sensor readings and the relative position of the target. The agent uses sensors to detect obstacles at various angles, translating these readings into a comprehensive view of the environment.

**Action Selection:** In the *decide* method, action selection happens based on the epsilon-greedy strategy. If the current state is not in the Q-table, a new entry is created. The agent chooses actions either by exploring randomly or exploiting known Q-values. The epsilon value gradually decays over time, reducing the probability of random exploration as the agent learns more about the environment.

**Learning:** After the agent takes an action and receives a reward, the update *q* method updates the Q-value for the state-action pair in the Q-table using the Bellman equation. This

process involves calculating the expected future rewards and adjusting the Q-value accordingly to reflect the new knowledge gained from the action taken.

**Exploration Rate Decay:** Gradually shifting from exploration to exploitation is achieved by the decay of the exploration rate ( $\epsilon$ ) as learning proceeds. This encourages the agent to leverage its learned knowledge more over time, balancing the trade-off between exploring new actions and exploiting known rewarding actions.

## VI. OBSERVATIONS ON Q-LEARNING WITH DYNAMIC OBSTACLES

The Q-learning algorithm demonstrated a robust ability to adapt to a dynamic environment, efficiently navigating towards targets while avoiding both static and moving obstacles. Initially, the agent required a higher number of steps to reach targets due to the exploration phase, characterized by random actions influenced by a high epsilon value. As training progressed, the agent's performance improved, with a noticeable reduction in steps required for each subsequent target.

Dynamic obstacles introduced complexity, necessitating continuous policy updates based on changing positions. The agent successfully avoided collisions by utilizing sensor inputs that provided information on obstacle proximity. The random movements of dynamic obstacles created unpredictable scenarios, testing the agent's adaptability and decision-making capabilities.

The variability in the agent's performance, as visualized by the bar chart of steps required per target, highlighted the algorithm's adaptability. Some targets were reached more quickly due to favorable initial conditions or random obstacle movements creating easier paths. Overall, there was a clear trend of improved efficiency as the agent refined its Q-values through repeated interactions with the environment.

In summary, the Q-learning algorithm effectively navigated a grid environment with both static and dynamic obstacles, demonstrating its potential for real-world applications where environments are unpredictable and require adaptive, intelligent navigation strategies.

## VII. DEEP Q-LEARNING

Deep Q-Learning (DQL) is an advanced variant of the traditional Q-learning algorithm, designed to tackle more complex tasks by leveraging the power of deep neural networks. At its core, DQL retains the fundamental concept of learning the value of actions in specific states through interaction with the environment. However, instead of maintaining a table of Q-values, DQL employs a deep neural network to approximate the Q-function, allowing it to handle large and continuous state spaces more efficiently.

In DQL, the process begins with the initialization of the Q-network, where the weights of the neural network are set to random values. Unlike in traditional Q-learning, where a Q-table is initialized with zeros, the neural network in DQL starts with randomly initialized parameters. This network serves as the function approximator for the Q-values, taking the state as input and outputting Q-values for all possible actions.

Upon observing the current state, the agent selects an action based on its current policy. This action selection process can incorporate exploration strategies such as epsilon-greedy, enabling the agent to balance between exploiting known actions and exploring new ones.

Once an action is selected and executed, the agent observes the resulting reward and the next state. Here comes the crux of DQL: the Q-network update. Instead of directly updating a Q-table, DQL calculates a target Q-value using the Bellman equation, which estimates the expected cumulative reward for taking a specific action in the current state and then following the optimal policy thereafter. This target Q-value is computed using the Q-values predicted by the neural network for the next state and is used to update the network's weights.

The process of updating the Q-network involves minimizing the difference between the predicted Q-value and the target Q-value. This is typically achieved by optimizing a loss function, such as the mean squared error, which quantifies the disparity between the predicted and target Q-values. Through techniques like stochastic gradient descent, the network's weights are adjusted to minimize this loss, effectively improving its ability to approximate the true Q-function.

One crucial aspect of DQL is the utilization of a separate target network, which periodically synchronizes its weights with the main Q-network. This helps stabilize the training process by providing more consistent targets for Q-value estimation, mitigating the risk of overestimation or divergence during training.

By iteratively updating the Q-network and interacting with the environment, the agent learns to navigate complex state spaces, adapt to dynamic environments, and optimize its decision-making process over time. Deep Q-Learning has demonstrated remarkable success across various domains, including robotics, autonomous systems, and gaming, showcasing its potential as a versatile and powerful reinforcement learning algorithm.

## VIII. IMPLEMENTATION OF DEEP Q-LEARNING

### A. Initialization

The agent and environment are initialized with their respective parameters, including the agent's position, learning rate, discount factor, and exploration rate. The Q-network, a neural network responsible for approximating the Q-function, is initialized with random weights.

### B. Architecture

There are two hidden layers, each containing 64 neurons and utilizing the Rectified Linear Unit (ReLU) activation function. The input layer receives state representations, such as sensor readings and agent positions, while the output layer produces Q-values for each possible action in the given state. With this configuration, the input layer processes environmental information, which is then transformed and processed through the hidden layers to approximate the Q-function. Finally, the output layer generates Q-values, indicating the expected cumulative future rewards for each action, thereby guiding the agent's decision-making process.

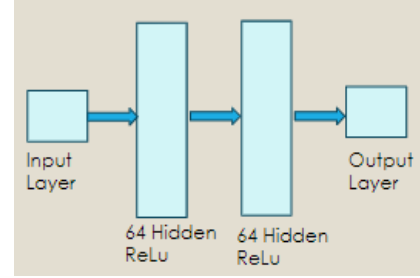


Fig. 3. Architecture of Network

### B. Details of Implementation

**Interaction with Environment:** The agent interacts with the environment by selecting actions based on its current state. Initially, actions are chosen randomly to explore the environment. The environment provides feedback in the form of rewards, guiding the agent's learning process. Rewards are based on the agent's actions and the resulting state transitions.

$$\text{Rewards} = \begin{cases} 100 & \text{,if agent reaches goal} \\ -100 & \text{, if agent hits the obstacle} \\ -1 * \text{dist\_to\_goal} & \text{, to reach goal faster} \end{cases}$$

**Action Selection:** The agent moves with 4 action sets (UP, DOWN, LEFT and RIGHT) and it follows an  $\epsilon$ -greedy policy to balance exploration and exploitation. With probability  $\epsilon$ , it selects a random action to explore the environment. Otherwise, it selects the action with the highest Q-value, exploiting the learned policy.

**Updating Q-Network:** After selecting an action and observing the resulting state and reward, the agent updates its Q-network using the observed experience. The Q-network is trained to minimize the Mean Squared Error loss between predicted and target Q-values. Target Q-values are calculated based on the observed rewards and estimated future rewards.

To stabilize training and improve sample efficiency, the agent stores experiences (state, action, reward, next state) in a replay buffer.

During training, experiences are randomly sampled from the replay buffer to update the Q-network. This helps break correlations between consecutive experiences and reduces the impact of outliers.

**Iterative Training:** Over time, the agent's exploration rate ( $\epsilon$ ) is decayed to gradually shift from exploration-heavy behavior to exploitation of learned policies. Decay strategies such as exponential decay or linear decay are employed to systematically reduce the exploration rate throughout training. The agent repeats the process of interacting with the environment, selecting actions, updating the Q-network, and decaying the exploration rate over multiple episodes.

With each episode, the agent's policy improves as it learns to navigate the environment efficiently, avoid obstacles, and reach the goal while maximizing cumulative rewards.

## IX. OBSERVATIONS ON DEEP Q-LEARNING

The agent demonstrates the capability to navigate through the environment despite the presence of dynamic obstacles that move randomly within the grid and the agent's learning progress is evident through the plotted graphs in Fig.4 and Fig.5 depicting the total reward per episode and the exploration rate over the course of training. This illustrates the agent's ability to adapt its navigation strategy over time, gradually improving its performance as it learns to effectively navigate the environment with dynamic obstacles. Additionally, the successful convergence of the training process, as indicated by the diminishing exploration rate and increasing total reward per episode, underscores the effectiveness of the DQL algorithm in addressing the challenges posed by dynamic obstacles in the environment.

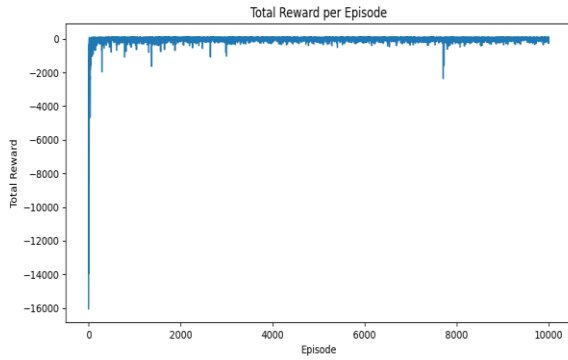


Fig. 4. Total rewards per Episode

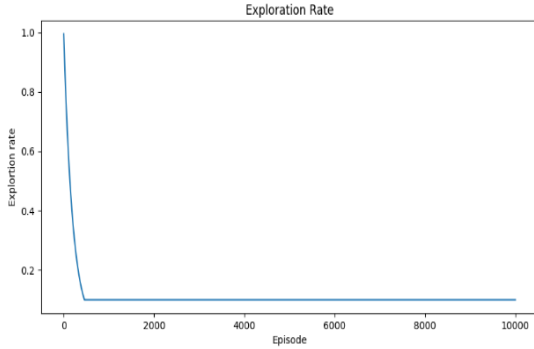


Fig. 5. Exploration vs Exploitation

As the agent explores the environment and gathers experience, it adjusts its Q-values based on the temporal difference error between predicted and target Q-values. The presence of a target Q-network stabilizes training by providing more consistent target values for Q-learning updates, mitigating the potential for overestimation bias and facilitating more effective learning. Additionally, the target Q-network helps smooth out fluctuations in Q-values, allowing the agent to learn more robust policies over time. Therefore, sudden increases in rewards may occur when the agent successfully exploits learned strategies or capitalizes on favorable changes in the environment, aided by the stability and guidance provided by the target Q-network.

## X. RESULTS

Q-Learning exhibited a commendable ability to navigate the dynamic environment, albeit with some variability in efficiency. Initially, the agent's journey towards goals was marked by a higher number of steps, primarily attributed to the exploration phase. During this phase, the agent's actions were influenced by a high epsilon value, leading to more random movements as it explored the environment. However, as training progressed, a noticeable improvement in performance was observed. The agent became more adept at efficiently reaching targets, with a discernible reduction in the number of steps required for each subsequent goal. This improvement underscored the adaptive nature of Q-Learning, as the agent learned from past experiences and adjusted its navigation strategy accordingly. Despite some variability in performance, with certain targets reached more quickly than others due to varying environmental conditions, Q-Learning demonstrated promising potential for adaptive navigation in dynamic environments.

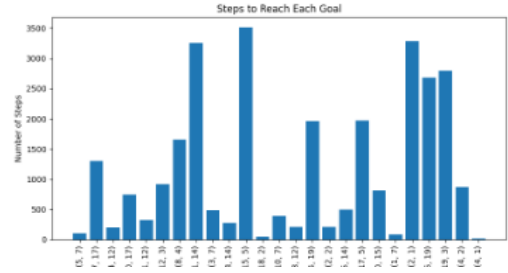


Fig. 6. Total steps in Q-Learning

In contrast, Deep Q-Learning (DQL) showcased superior performance compared to Q-Learning, consistently achieving goals with fewer steps. The utilization of neural networks enabled DQL to efficiently handle the complexities of large and continuous state spaces, providing the agent with enhanced decision-making capabilities. Through the iterative training process, DQL demonstrated remarkable adaptability and efficiency in navigating the environment with dynamic obstacles. The diminishing exploration rate and increasing total reward per episode indicated the algorithm's ability to effectively learn and optimize its navigation strategy over time. Additionally, the presence of a target Q-network contributed to stable training, resulting in smoother learning curves and the acquisition of more robust policies. Overall, DQL emerged as a powerful reinforcement learning algorithm, capable of addressing the challenges posed by dynamic obstacles and paving the way for sophisticated autonomous systems capable of real-time decision-making in complex environments.

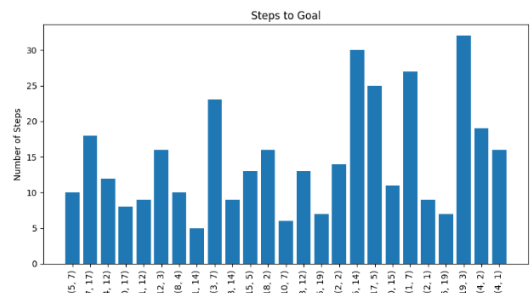


Fig. 7. Total steps in Deep Q-Learning



Comparing the performance of Q-Learning and DQL revealed distinct advantages of the latter in navigating through dynamic environments. While Q-Learning demonstrated adaptability and improvement over time, DQL exhibited a higher level of efficiency and consistency in achieving goals. The integration of neural networks allowed DQL to effectively learn from experiences, optimize decision-making, and navigate complex state spaces more efficiently. The stability provided by the target Q-network further enhanced DQL's training process, leading to the acquisition of more robust navigation policies. These findings highlight the potential of DQL as a versatile and powerful algorithm for dynamic obstacle avoidance and navigation tasks, with implications for the development of advanced autonomous systems.

## XI. CONTRIBUTION

The git repository <https://github.com/anassinator/dqn-obstacle-avoidance> implements a Deep Q-Network (DQN) agent designed to navigate a simulated environment populated with obstacles, both fixed and dynamic, while aiming to reach a specified goal position. This git repository uses a MIT's director visualization tool to simulate the environment and it uses a Deep Q-Network using TensorFlow to train the agent.

While the referenced Git repository lays the groundwork for implementing ray sensors and obstacle spaces within a simulated environment, the implementation presented introduces modifications and enhancements to these concepts and the environment in which it was simulated. Specifically, the ray sensor implementation differs in its design or functionality compared to the one referenced. This could involve changes in sensor range, resolution, or the method by which sensor readings are processed. For instance, the ray sensor in this implementation may have a different angular resolution or a wider field of view, allowing the agent to perceive its immediate grid surroundings. Additionally, the method for interpreting sensor readings and incorporating them into the agent's decision-making process might be refined or adapted to suit specific requirements or challenges within the simulated environment.

Regarding environment creation, the implementation likely involves the generation of obstacle spaces and dynamic obstacles within the simulated environment. Obstacle spaces define regions within the environment grid where obstacles are placed, influencing the agent's navigation and obstacle avoidance behavior. These obstacles may vary in size, shape, and density, providing a diverse and challenging environment for the agent to navigate through. Dynamic obstacles further augment the complexity of the environment by introducing elements that change position or behavior over time. This dynamic aspect introduces additional challenges for the agent, requiring adaptability and real-time decision-making to navigate effectively. Randomization techniques may be employed to introduce variability and unpredictability, ensuring that the agent's learning process is robust and generalizable across different scenarios. Additionally, the creation of goal states within the environment provides clear objectives for the agent to strive towards, guiding its learning and incentivizing successful navigation.

Overall, the implementation builds upon the concepts of ray sensors and obstacle spaces from the referenced repository,

introducing modifications and enhancements to suit specific requirements or challenges within the simulated environment. By tailoring these elements to the task at hand and carefully crafting the environment, the implementation aims to provide a rich and challenging learning environment for the agent to develop effective navigation and obstacle avoidance strategies.

The implemented code uses a Deep Q-Learning algorithm to train an agent to navigate through an environment to reach a goal while avoiding obstacles. The environment consists of a grid with static and dynamic obstacles, as well as a goal position. The agent is equipped with sensors that detect obstacles within a certain range.

The "DeepQLearningAgent" class represents the agent, initialized with its position and sensor parameters. It utilizes a neural network (Q-Network) to approximate the Q-values for different actions. The Q-network is loaded with pretrained weights from a saved model file.

The Environment class defines the environment, including the agent, obstacles, and goal. It handles the movement of dynamic obstacles and calculates rewards based on the agent's actions. Rewards are assigned based on the distance to the goal, with penalties for colliding with obstacles. The environment also provides methods to visualize the grid.

The main program runs for a set number of episodes, during which the agent interacts with the environment. In each episode, the agent selects actions based on its current state, moves accordingly, and receives rewards. The environment's state is updated, and the loop continues until the episode ends.

During each episode, frames of the environment are captured to create a video visualization of the agent's navigation. OpenCV is used to save these frames as a video file.

After all episodes are completed, the rewards and steps taken in each episode are plotted to visualize the agent's learning progress. Additionally, a bar plot shows the number of steps taken to reach each goal position, providing insights into the agent's efficiency in navigating to different goals.

The GitHub repository introduces a Deep Q-Network (DQN) agent tasked with navigating obstacles to reach a goal position, referencing concepts like ray sensors and obstacle spaces. However, while it lays the foundation for these elements, the implementation provided diverges notably in both training methodology and neural network architecture. Instead of TensorFlow, this implementation employs PyTorch for Deep Q-Learning (DQL), indicating a shift in training strategy and possibly network design. While both implementations share the goal of training an agent to navigate a complex environment, this approach likely introduces distinct optimizations and network configurations to achieve efficient learning and decision-making in the simulated scenario.

## XII. CONCLUSION

In conclusion, this study delved into the realm of dynamic obstacle avoidance, a critical facet of autonomous navigation systems, particularly in the context of applications like autonomous driving. The investigation centered on employing Q-learning and Deep Q-learning techniques to navigate a robot through a dynamic environment, aiming to reach predefined goal points. The research, conducted through simulation and implementation using Python and Pygame, alongside the PyTorch framework for Deep Q-learning, aimed to contribute foundational insights into developing advanced autonomous

driving systems capable of real-time decision-making in complex and ever-changing environments.

Comparative analysis between Q-learning and Deep Q-learning revealed distinct advantages of the latter, particularly in terms of efficiency and consistency in achieving navigation goals. Deep Q-learning, with its neural network-based approach, exhibited superior performance in handling complex state spaces and dynamic environments, paving the way for more sophisticated autonomous systems.

In essence, this study serves as a stepping stone towards the development of advanced autonomous navigation systems, showcasing the potential of reinforcement learning techniques like Q-learning and Deep Q-learning in addressing the challenges of dynamic obstacle avoidance. Moving forward, further research in this domain could explore advanced methodologies, incorporate real-world data, and tackle more complex scenarios to propel the field of autonomous navigation towards greater efficiency and reliability.

### XIII. REFERENCES

- [1]. M. Gu and Y. Huang, "Dynamic Obstacle Avoidance of Mobile Robot Based on Adaptive Velocity Obstacle," 2021 36th Youth Academic Annual Conference of Chinese Association of Automation (YAC), Nanchang, China, 2021, pp. 776-781.
- [2]. X. Chen, L. Su and H. Dai, "Mapless navigation based on continuous deep reinforcement learning," 2021 China Automation Congress (CAC), Beijing, China, 2021, pp. 6758-6763.
- [3]. Y. Long and H. He, "Robot path planning based on deep reinforcement learning," 2020 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS), Shenyang, China, 2020, pp. 151-154.
- [4]. B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in IEEE Access, vol. 7, pp. 133653-133667, 2019
- [5]. T. Ribeiro, F. Gonçalves, I. Garcia, G. Lopes and A. F. Ribeiro, "Q-Learning for Autonomous Mobile Robot Obstacle Avoidance," 2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC), Porto, Portugal, 2019, pp. 1-7.