Autonomous Navigation and Obstacle Avoidance Using DQN with Turtlebot

Suhas Nagaraj
University of Maryland
suhas99@umd.edu
UID: 119505373

Swaraj Mundruppady Rao
University of Maryland
swarajmr@umd.edu
UID: 120127007

***Abstract*—In recent times, there has been increased focus on autonomous vehicles, from autonomous cars for personal/commercial use to autonomous drones for space exploration and military use. Navigation by these vehicles in an unknown and dynamic environment is a crucial capability. The conventional methods of navigation involve localizing the robot, building maps of the environment, using algorithms to generate a path to the goal and move the robots towards the goal. If the environment was dynamic, then the steps of building the map and planning a path must be done repeatedly in real time. Hence the output of these conventional methods cannot be reused due to the changing environment.**

In contrast, deep reinforcement learning models trained for autonomous navigation and obstacle avoidance can be used as it addresses the shortcomings of the conventional methods. The model can be reused in different environments to get the desired outcome. In this project, we have trained a mobile robot to navigate to the goal pose from its initial pose, while navigating through a set of obstacles which are static and dynamic in nature using the Deep Q-Network model. Through Deep Q Network the mobile robot can learn the environment through wandering and learning to navigate to the goal autonomously through input from Lidar sensor and odometry. After thorough training, the robot can reach the target destination without colliding with any obstacles (Static/ Dynamic).

***Index Terms*—Autonomous Navigation, Obstacle Avoidance, DQN, Deep Reinforcement Learning, Mobile Robot**

I. Introduction

As the range of applications for robots continues to expand, the environment in which robots are intended to operate is becoming increasingly complex. To autonomously navigate through an environment and perform a variety of tasks, many mobile robot systems heavily rely on the map and their own location information. The robot needs to reason the next action based on the known information. However, due to the limited environmental information accessible to robots and the unpredictability of environmental conditions, the practical application of conventional navigation algorithms is restricted. Reinforcement learning (RL) techniques can learn appropriate actions from the environment states. In the process of interaction between the agent and the external environment, the agent repeatedly learns through trial and error, obtains environmental information, and continuously optimizes the action strategy of the agent. This optimization method gives the RL an excellent decision-making ability. At present, reinforcement learning has been successfully applied in mobile robot path planning. However, the performance of RL usually heavily depends on the selection of artificial features. The quality of features selected significantly affects the learning outcomes.

Reinforcement learning (RL) algorithms, particularly Q-Learning, have shown remarkable success in solving various tasks by learning optimal policies through trial and error. However, traditional Q-Learning methods face challenges when applied to environments with large state and action spaces due to the impracticality of maintaining explicit Q-value matrices. To address this limitation, Deep Q-Learning (DQL) emerges as a promising solution by leveraging deep neural networks to approximate Q-values, enabling scalability to complex environments. We have shown the implementation of DQN algorithm and its potential to be used for autonomous navigation and obstacle avoidance in a dynamic environment, where there are dynamic obstacles (moving cylinders, can be related to humans or any other moving object in real world scenarios) as well as static obstacles (obstacles such as walls etc. which are static in the real world as well).

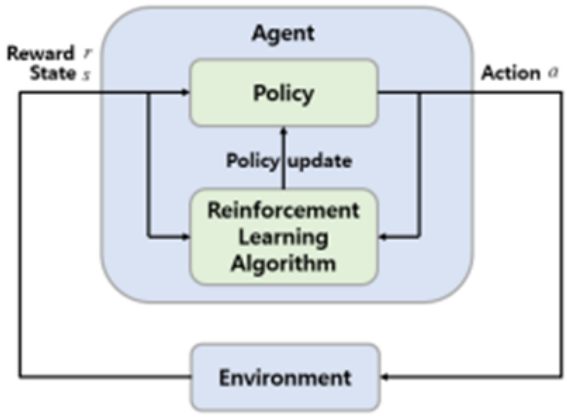

Fig. 1. Basic RL Network

We will be using an architecture leveraging ROS2 (Robot Operating System 2) and the Gazebo simulation environment to implement a reinforcement learning framework for autonomous navigation tasks. Our system centers around the

Turtlebot3 platform. Within this framework, we utilize LIDAR (Light Detection and Ranging) sensor and Odometry data as input for the autonomous navigation system. The LIDAR sensor provides rich environmental information crucial for mapping and localization tasks, enabling the robot to perceive its surroundings accurately, whereas the Odometry data gives the real time position and orientation of the robot. To realize intelligent decision-making capabilities, we integrate Deep Q-Networks (DQN). The DQN algorithm learns to navigate the Turtlebot3 autonomously by directly interacting with the environment, optimizing its actions based on rewards obtained through successful navigation tasks. This architecture combines the strengths of ROS2 for robust communication and modular system development, Gazebo for realistic simulation, and DQN for efficient and adaptive learning, offering a comprehensive solution for autonomous robotic navigation research.

II. Related Work or Literature Review

In [2], a reinforcement learning method is used for exploring a corridor environment with the depth information from RGB-D sensor alone. The system is based on Deep Q-Network framework in which Convolution Neural Networks, CNN, are used for Q-value estimation. They have leveraged Deep Q-Network (DQN) architecture, separating it into supervised deep learning and Q-learning components. The approach achieves obstacle avoidance through pre-training feature maps and demonstrates robustness across various corridor environments in TurtleBot experiments.

In [3], they have implemented Deep Q-Learning for simulation of self-driving car which moves in left, right, up, and down directions. Using this algorithm, they monitored what actions were being performed by the agent and what could be the next move or action it may perform based on its experience. The Performance of the agent was demonstrated via a simulation environment where the agent had to cover a path from origin to destination, receiving some reward. The environment does include some obstacles, which may provide some penalties if they come in contact with the agent. For the model they are using 3 layers, the first layer being the input layer with inputs from signals taken from the car and one for the orientation of the car. The last layer is the output with three neurons, where the three actions are for going straight, left, or right. The neural network uses a ReLU as its activating function which is being used to forward propagate the input signals to the network.

In [5], they have studied the problem of robot's autonomous navigation in dynamic environment using the DRL algorithm. They selected DQN, Double DQN and Dueling DQN as the robot navigators. Due to the poor performance of the traditional reward shaping method, they designed a novel improved reward shaping method and combined it with the three standard deep reinforcement learning algorithms. With the improved shaping method, the DQN algorithm obtained great increases both in performance and effectiveness, compare to the original ones. And with an improved shaping method added, all the DQN, Double DQN and Dueling DQN converged to greater average success rates.

In[6], the article presents a novel control algorithm based on a DQN network which has been designed as a node through ROS API. The architecture integrated different open-source systems that are represented as the agent and the environment, and they are interrelated through states, actions and rewards to learn through reinforcement. Among the optimization algorithms tested, RMSprop demonstrated the best results. When utilizing RMSprop, the robot exhibited a higher success rate in reaching the goal and effectively avoided obstacles. The reward values steadily increased over time, indicating that the network was functioning optimally, and the algorithm was learning from its experiences. The environment complexity was increased and the DQN algorithm was able to handle the complex and dynamic environment with a higher score showcasing the ability of DQN to handle complex environments.

Autonomous Navigation of mobile robots using fusion of sensor data by a Double Deep Q-Network[8] with collision avoidance by detecting moving people using computer vision techniques. Two data fusion methods: Interactive and Late fusion strategy are used to integrate sensor information from GPS, IMU and RGB-D cameras. The proposed collision avoidance module is implemented along with sensor fusion architecture to prevent the robot from colliding with moving people. The results had a performance increase of nearly 27% in relation to navigation without sensor fusion.

Considering the various methodologies and their respective advantages, the choice of DQN for this project is justified by its simplicity and effectiveness. DQN is a framework that provides a robust foundation for developing and testing autonomous navigation algorithms. Its ability to handle complex sensor inputs makes it an ideal choice for the requirements of this project. Implementing improvements to the model and tuning the parameters will ultimately lead to a more efficient autonomous navigation system.

III. Methodology

This section will explain the methodology followed to achieve the autonomous navigation by avoiding static and dynamic obstacles.

A. DQN

The development of AI techniques, especially in the field of reinforcement and deep reinforcement learning has contributed to the growth in the field of robotics for autonomous navigation and obstacle avoidance. In reinforcement learning, feedback in the form of rewards is given to the agent when it interacts with the environment. When this interaction produces a positive result which is more inclined towards the desired behavior, the agent earns a positive reward. In contrast, if the interaction between the agent and the environment is against the desired behavior, the agent receives a negative reward for its action. The model learns strategies through the interaction between agents and the environment to maximize the rewards, which leads to desired behavior. The model can use the information that it gets from the agent's environment and learn optimal

strategies, without any prior knowledge of the environment. Hence, this method is useful in cases where little to no knowledge is available of the agent's environment and the model must learn from the information it receives.

Therefore, the basis of obstacle avoidance and autonomous navigation under the reinforcement learning method is to let the agent make sequential decisions to maximize the reward by completing the objective of reaching the goal by avoiding obstacles.

DQN (Deep Q-network) refers to a Q-learning algorithm (reinforcement learning) based on deep learning. It combines Q learning's goal of learning the optimal action-selection policy for an agent interacting with an environment and neural network technology and uses the target network and experience replay to train the network. DQN is a value-based algorithm; It focuses on estimating the value of different actions in a given state and selecting the action with the highest value. In the value-based algorithm, what they learn is not the strategy but the evaluation of the action. In DQN, the current state of the environment is given as an input. The state is extracted/estimated by sensors and the sensor data is given as an input to the model. This neural network processes the state and extracts feature that are relevant for estimating the Q-values of different actions.

Hence, DQN allows agents to interact with the environment in real-time and improve obstacle avoidance and autonomous navigation skill by constantly accumulating experience in the form of updated Q values, which is very similar to how a human learns new skills. In both cases, learning is a continual process of gathering information, evaluating actions, and updating strategies to improve performance over time.

In addition, DQN also considers the constraints of navigation rules and conventions in the form of reward functions, action space selection and exploration strategy, when designing the value function. The DQN agents implement actions according to specific strategies, according to the rewards and environmental feedback of the new state. This allows agents to learn which actions they should take in a specific state to maximize their rewards.

DQN algorithm incorporates the parameters of a neural network into its algorithm with synaptic weights and bias. Normally the network layer used is deep, which has more than 3 layers and the aim is to estimate the optimal state-action value function. The DQN does not need a tabular representation to store and map agent states and actions, like traditional Q Learning. The DQN state action mapping is provided by the neural network itself. Also, in DQN the data are dynamic in nature and are updated at each step of the iteration.

In this project, we are using the Deep Q-Network technique to train a mobile robot (TurtleBot) to avoid obstacles and reach its goal. As depicted in Figure, the architecture integrates multiplatform systems such as ROS, enabling seamless message transmission among different components of the system. Additionally, the Gazebo simulator is employed to facilitate autonomous learning through DQN reinforcement.

B. Network Architecture

The input to the DQN includes data from a Lidar sensor, specifically 60 Lidar readings taken at different angles around the robot, combined with 2-dimensional relative goal information (goal position and angle) obtained using the Odometry sensor and 2-dimensional robot-action information (Linear and Angular velocity). Each feature or piece of information in the input is represented by a separate neuron in the input layer. Hence, the input layer size is 64 in our architecture.

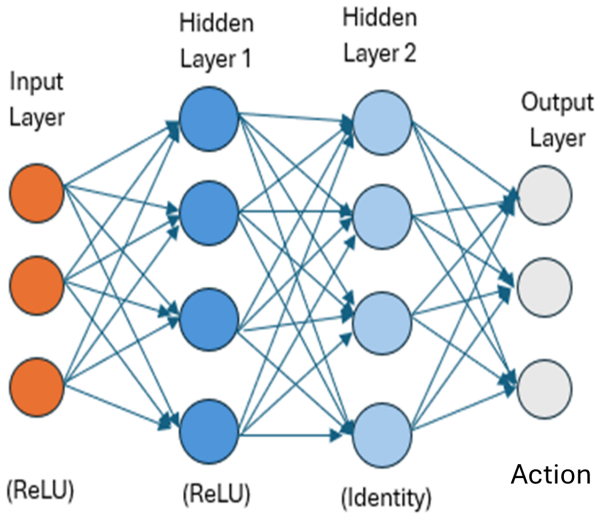

Fig. 2. Neural Network Model

The input layers are connected with two dense layers, each comprising of 1024 nodes/neurons. Dense layers, also known as Fully Connected Layers, imply that each neuron in each layer is connected to every neuron in the other layer. Rectified Linear Unit (ReLU) activation functions are employed between consecutive layers. ReLU activation introduces nonlinearity to the network. Nonlinearity is essential because it allows neural networks to model and learn complex relationships and patterns in data that may not be captured by linear functions. The ReLU function is defined as f(x) = max(0,x) where x is the input to the neuron. This function outputs the input value if it is positive, and outputs zero otherwise. ReLU is used as the activation function due to its simplicity and effectiveness in promoting sparse activations and countering the vanishing gradient problem.

The output of the network is an action set which acts as the Q table for all possible actions. The number of nodes in the output layer will be equal to the number of possible actions, which is 12 in our case, where each node represents a particular action. This layer represents the estimated Q-values for each possible action.

The weights of the network are initialized using the Xavier uniform initialization method. The method sets the weights

with values drawn from a uniform distribution bounded between $-\frac{\sqrt{6}}{\sqrt{n_i+n_o}}$ and $\frac{\sqrt{6}}{\sqrt{n_i+n_o}}$, where n_i and n_o are the number of input and output features of the layers respectively. This type of initialization is designed to keep the scale of the gradients roughly the same in all layers, to prevent issues such as exploding or vanishing gradients during training of the deep networks.

During training, our network is trained using backpropagation, where gradients are propagated backward through the network to update the weights. If the gradients become too small (vanishing gradients) or too large (exploding gradients), it can lead to slow convergence or instability in the training process. By performing proper initialization, selecting suitable activation functions, and employing gradient clipping, these problems are countered.

The primary function of this DQN network is to evaluate the potential reward from taking different actions given the current state of the environment. This is a fundamental aspect of reinforcement learning, where an agent learns to make sequential decisions in an environment to maximize the rewards. The DQN network plays a crucial role in this process by approximating the value function, which represents the expected cumulative reward that the agent can achieve from a given state and action. The value function is typically denoted as Q(s,a), where s is the state and a is the action. By learning to approximate this function, the DQN can predict the potential future reward associated with each action in a given state. During training the network weights are updated based on the reward outcomes and the MSE loss computed through interaction with the environment.

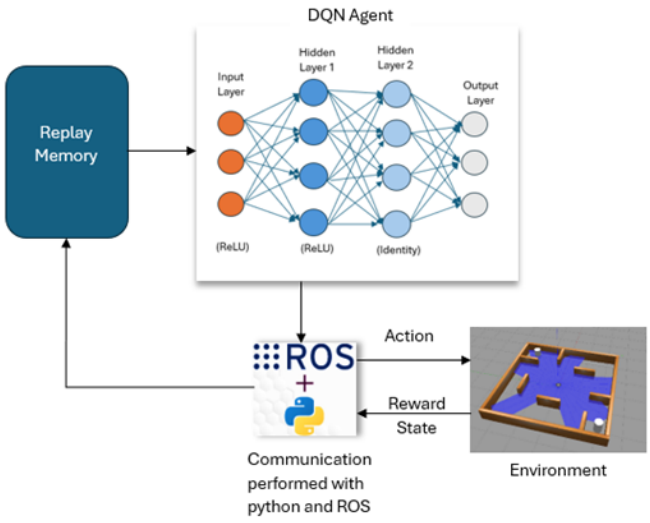

Fig. 3. Training Network

The replay buffer stores information collected during the interaction of the agent with the environment. Each entry in the buffer contains information on state, action, reward, next state, and a flag variable. During training, the DQN algorithm samples mini batches of the information from the replay buffer to update the network's weights. This process breaks the correlation between consecutive samples and facilitates improved data efficiency, more efficient and stable learning from diverse experiences. It is a mechanism for the agent to remember and learn from its past interactions with the environment, ultimately leading to better decision-making and performance. The target Q-network, which is a clone of the Actor network, is used to generate stable target Q-values during training, helping to reduce overestimation and stabilize learning. Initially identical to the actor network, the target network is updated periodically (either using hard or soft updates) to slowly track the learned Actor network. In hard updates, the weights of the primary Q-network are copied to the target Q-network. In contrast, in soft updates, the target Q-network is updated gradually by blending its weights with those of the primary Q-network over time, using a parameter, Tau.

The training loop uses an epsilon-greedy policy to balance between exploration and exploitation. With probability ϵ, the agent chooses a random action to explore the environment (Exploration). This random exploration helps the agent to discover new states and actions. With probability $1-\epsilon$, the agent selects the action with the highest Q-value predicted by the Actor network (Exploitation). This exploitation step allows the agent to choose actions that are believed to be optimal based on the current knowledge. Initially, the ϵ value is 1, which means that the agent will always choose random actions at any state. However, after each episode, the ϵ value is multiplied by an epsilon decay rate. This multiplication ensures a gradual shift from more exploration to more exploitation. The ϵ value is capped at a minimum value to ensure that the agent continues to explore the environment to some extent even as training progresses.

The loss between the predicted Q-values and the target Q-values is calculated using mean squared error (MSE) between the predicted Q-values and the target Q-values. The MSE loss measures the average squared difference between the predicted Q-values and the target Q-values across all actions and the goal is to minimize this loss. In Q-learning, the goal is to learn a policy that maximizes the rewards gained over time. The MSE loss function helps in achieving this goal by providing a clear and direct signal to the neural network on how to update its weights to reduce prediction errors. By minimizing the MSE loss, the network learns to approximate the Q-values more accurately, leading to better decision-making and ultimately improving the policy's performance. Mathematically MSE is given by:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(Q_{\text{predicted},i} - Q_{\text{target},i})^2$$

MSE is sensitive to large errors due to its squaring term. This property makes it effective at bringing down errors in predictions, especially when there are outliers or large differences between the predicted and target values. This

ensures that the optimizer focuses more on correcting larger mistakes, which is important for matching predicted Q-values closely with their targets and improving the overall training performance.

During training, backpropagation is used to compute the gradient of the loss function with respect to the network weights. This process involves calculating how much each weight contributes to the overall error, starting from the output layer and propagating backwards through the network. Clipping of the gradients of the actor networks parameters is performed during backpropagation to prevent the gradients from becoming too large. If they become too large, it can lead to unstable training (exploding gradients).

The AdamW optimizer is instantiated when initializing the networks. AdamW combines the advantages of both Adam and L2 regularization. L2 regularization penalizes large weights to prevent overfitting by adding a regularization term to the loss function. Once the loss gradients are computed through backpropagation, AdamW adjusts the parameters based on these gradients and the learning rate. This process is repeated for each batch over multiple episodes, progressively refining the network's parameters to minimize the loss function. This optimization process, combined with backpropagation and loss computation, forms the backbone of the training process.

C. Reward Function Setup

In our project, the agent, which is the turtlebot3 robot, is expected to move in the Euclidean plane towards the goal in the presence of static and dynamic obstacles. As the robot moves from one state to another, it observes the environment directly by considering the range data from its LIDAR sensor and indirectly by calculating the distance and angle to the goal using its Odometry sensor. It feeds this information to the DQN model which then selects an action for the agent to execute based on the Q-values calculated and assigned to each action choice (during exploitation) based on the current internal network weights. Based on the output of the action or series of actions, rewards are assigned to the agent. Rewards are like feedback the robot/agent receives from its environment after performing actions. If the action results in a behavior which is more inclined towards the desired or target behavior, the agent receives a positive reward. In contrast, the agent receives a negative reward if it performs an action which results in behavior which is not desired. The robot or the model tries to maximize the rewards that it obtains, and thus tries to select that action which might result in a higher cumulative future reward. Hence, it is essential to design a reward function based on the target behavior, current and previous states to ensure accurate training of the robot.

At the end of each episode, the agent receives a cumulative reward based on its interaction with the environment at each step. This cumulative reward is a combination of individual rewards it receives based on different behaviors observed in the agent. The rewards assigned are explained below:

- **Yaw Reward** (r_{yaw}): This is a reward assigned by considering the current orientation and the direction of the goal. If the difference between the current heading and the desired heading (desired heading is the direction towards the goal) is high, a higher negative reward is assigned. As the agent tries to maximize the reward (or minimize the negative reward), it learns to move towards the goal which results in a lesser negative reward.
- **Distance to Goal Reward** (r_{distance}): This is a reward that is assigned by considering the initial distance to the goal at the start of the episode and how the distance changes at each step the agent takes. The function used assigns a greater reward to the agent as the agent moves closer to the goal. Hence, as the agent tries to maximize the reward, it learns to move towards the goal to get a higher reward.
- **Minimum Obstacle Distance Reward** ($r_{\text{min\_obstacle\_dist}}$): This is a reward that gets assigned to the agent if it is too close to any obstacle. The proximity information is obtained from the scan data of the LIDAR sensor. The minimum data value is considered as the minimum distance to the goal and hence, it is used to assign the reward. A negative reward is assigned to the agent if the LIDAR readings fall below a threshold.
- **Episode Outcome Reward**: If the robot reaches the goal, a positive reward will be awarded. In contrast, if the robot crashes into any obstacle (static or dynamic) a negative reward will be awarded. This will train the robot to avoid crashing into obstacles and encourage it to reach the goal.

The rewards 1 to 3 mentioned above are assigned at every step taken by the robot, whereas reward 4 mentioned above is only assigned at the end of the episode.

The reward collected at each step and the reward awarded at the end of the episode based on the outcome are added together to get the cumulative reward the agent gets per episode.

D. Training Environment

The training environment for the TurtleBot3 robot is set up in the Gazebo simulation, which provides a realistic and dynamic setting for the robot to learn autonomous navigation and obstacle avoidance. The environment consists of a bounded area with static walls and dynamic obstacles represented by white cylinders. The environment setup and goal management are crucial for the robot to experience varied scenarios, promoting robust learning.

The white cylinders in the map represent dynamic obstacles. These obstacles can move within the environment, providing an additional layer of complexity as the robot must learn to react to moving objects. The map is enclosed by walls, creating a confined space where the robot operates. The walls form corridors and pathways, challenging the robot to navigate through constrained spaces. For the goal generation random points are generated within the map space. Once the random point is generated, it will check if the point lies in the obstacle space. Only if the point is feasible to navigate to, then the point is chosen as the goal position to which the robot has to

navigate to by avoiding any obstacle in its path.
The environment setup used has the presence of both static walls and dynamic obstacles ensures that the robot learns to navigate in varied and unpredictable scenarios, which is crucial for real-world applications. The random placement of goals challenges the robot to adapt its navigation strategy continuously. The Gazebo simulation provides a realistic environment that closely mimics real-world conditions, enhancing the transferability of the learned behaviors to physical robots. The reward function incentivizes the robot to move towards the goal while avoiding obstacles.
The environment used has a size of 4.2 m x 4.2 m, the turtlebot is spawned during the start of each episode at (0,0) location on the gazebo map.

The environment visualisation in the gazebo environment is as shown in the figure below

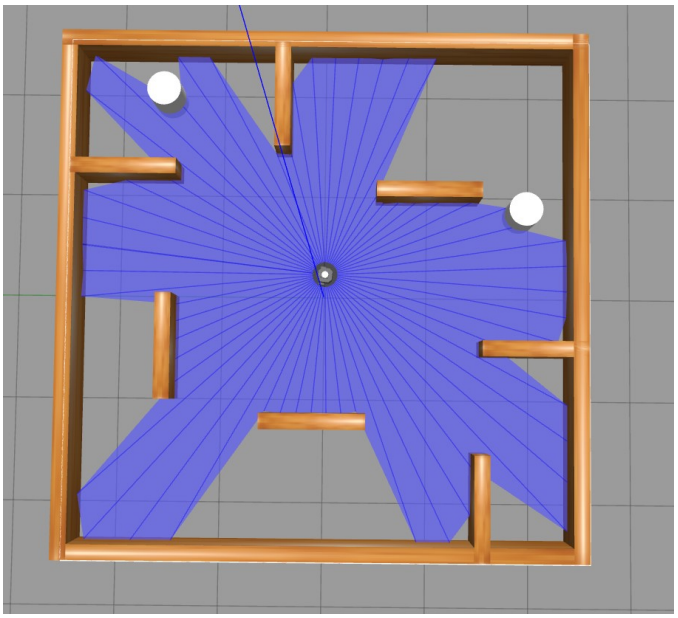

Fig. 4. Training Environment

IV. Our Contribution

For this project, the following GitHub repository is used as the main reference: https://github.com/tomasvr/turtlebot3_drlnav
The DQN architecture described in the GitHub repository was used as a reference and we built our own DQN architecture for the purpose of training a mobile robot to avoid obstacles and autonomously navigate in an environment. The important contributions to the project from our side are explained in the below subsections

A. Redefined Action Set

During testing of the reference code, we observed that the robot's movement was hindering its ability to manage static and dynamic obstacle avoidance. The robot's actions set is a combination of linear and angular velocities. The action set in the referenced code comprised of five different actions

Action Set	**1**	**2**	**3**	**4**	**5**
Linear Velocity	0.3	0.3	1.0	0.3	0.3
Angular Velocity	-1.0	-0.5	0.0	0.5	1.0

TABLE I
ACTION SETS WITH CORRESPONDING LINEAR AND ANGULAR VELOCITIES IN THE REFERENCE

Upon close observation of the action set it can be inferred that all the actions set mandated some sort of forward movement, which might not always be desirable, especially in high dynamic environments. We have expanded the action set to improve its navigation capabilities. Additions to the original action set

Action Set	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Linear Velocity	0.0	0.6	0.6	0.5	0.6	0.6	0.0
Angular Velocity	-0.5	-1.0	-0.5	0.0	0.5	1.0	0.5

TABLE II
ADDED ACTION SETS WITH CORRESPONDING LINEAR AND ANGULAR VELOCITIES

Owing to the additions, the current action set has the following advantages:

- **Enhanced Maneuverability:** The inclusion of additional actions with varied linear and angular velocities significantly enhances the robot's maneuverability. In environments with dynamic obstacles, such as moving people or vehicles, the ability to quickly adapt its movement strategy is crucial. Actions that allow the robot to stop or change directions sharply can prevent collisions and enable more responsive navigation.
- **Increased Action Choices:** By expanding the action set from five to twelve options, the robot has a broader repertoire of moves to choose from at any given moment. This variability is important not only for avoiding obstacles but also for optimizing its path towards a target. For example, actions that combine higher linear velocities with moderate angular velocities can be optimal in open spaces where the robot needs to cover ground quickly while making gradual adjustments to its course.
- **Zero Movement Action:** Introducing an action with zero linear and angular velocities. This is especially useful in scenarios when the obstacle is too close to navigate away from.
- **Rotation on the Spot:** Actions that focus on angular changes while maintaining minimal to no linear movement allow the robot to rotate about its own axis. This is beneficial in tight spaces where the robot needs to reorient its heading without changing its position, facilitating precise adjustments in orientation without straying from its current location.
- **Reduction in Negative Rewards:** In navigation tasks, especially those driven by reinforcement learning, reducing negative rewards is crucial for efficient learning. The expanded action set can minimize penalties associated

with undesirable orientations or collisions. By allowing the robot to align more quickly and accurately with its desired trajectory, it can reach its goal more efficiently, thereby maximizing positive reinforcement signals.

B. Enhanced Input Architecture and Lidar Resolution

In the original reference model, the inputs given to the network consisted of 40 Lidar Scan Data (State) and 4 other inputs consisting of Relative Goal Position, Relative Goal Angle, Linear Velocity, and Angular Velocity data. Lidar Data sampling was conducted for every 9-degree interval.

The challenge of missing edges due to Lidar data being sampled at 9-degree intervals can significantly affect a robot's ability to navigate effectively, particularly in environments where obstacles are positioned close to one another or at angles that the Lidar fails to capture efficiently. This limitation can be addressed by enhancing both the resolution of the Lidar data and the overall input architecture of the neural network.

To increase the resolution of the scan samples and to balance the complexity of the architecture, we chose an interval of 6 degrees instead of 9 degrees. This increase in resolution was done by changing the LIDAR sensor specification related to the number of LIDAR scan samples in the .sdf file (in the plugin section) of the TurtleBot robot and it was verified using ROS2. This higher resolution can be crucial for detecting smaller or narrow obstacles that might be missed at lower resolutions.

The advantages of increasing the resolution are as follows:

- **Enhanced Detail Capture:** By reducing the angular interval between consecutive Lidar scans, the Lidar sensor captures more data points within the same 360-degree sweep. This increase in data points results in a more detailed and better representation of the environment, allowing the robot to detect smaller features that would otherwise be missed with coarser scans.
- **Edge and Obstacle Detection:** Finer angular resolution improves the robot's ability to identify the edges of obstacles and openings between them. This is particularly valuable in cluttered or highly dynamic environments where navigating safely depends on recognizing precise details about nearby objects.

The current input size is 64, of which 60 is LIDAR scan data and the rest are Relative Goal Position, Relative Goal Angle, Linear Velocity, and Angular Velocity data.

C. Increased Hidden Layer Size

As both the Input layer size and the number of action sets were increased, it was important to give more capability to the network to facilitate better learning, reduce overfitting and for better generalization of the data. Hence, the number of neurons in the hidden layers were increased from 512 to 1024. Other possible advantages that we considered before increasing the hidden layer size are given below:

- The environment or task that the DQN model is tackling is complex as it involves large state and action space due to dynamic obstacles in the environment; a larger hidden layer size provides the network with more capacity to learn and represent these complexities.
- A larger hidden layer size allows the neural network to capture more intricate patterns from the data. This can be beneficial for tasks where subtle features between inputs and outputs are important to consider for making accurate predictions.
- Increasing the hidden layer size increases the number of parameters in the network, providing it with greater flexibility to learn from the training data. This leads to faster convergence and better overall performance.

D. Tweaked Update Mechanism

In reinforcement learning (RL), maintaining an up-to-date target network is crucial for stable learning. The target network serves as a stable baseline to estimate the future value of actions, and its weights are frozen during the learning updates of the primary network to reduce correlation between the target and the estimated Q-values. In the original reference, only hard updates were carried out on the target network for every 1000 episodes. We believe this was one of the potential reasons the network had a bad training record even after 7000 episodes. The implications of infrequently updates networks are as follows

- **Delayed Learning Feedback:** By updating the target network only every 1000 episodes, the feedback loop is significantly delayed. This means that the learning process is based on potentially outdated information, which can lead to slower convergence or even divergence in policy learning.
- **Policy Lag:**The primary (online) network might learn and adapt to the environment nuances much faster than the target network updates, leading to a significant lag in the policies reflected by the two networks. This disparity can cause the learning to oscillate or deviate from optimal policies.
- **Stability vs. Responsiveness:**While less frequent updates can sometimes aid stability by preventing drastic changes in the policy (as abrupt changes can destabilize the learning process), they can also make the system less responsive to changes in the environment or to new strategies that the agent has learned.

To overcome these issues, the hard update interval was reduced from 1000 to 500, also a soft update feature was added, at an interval of every 10 episodes, to improve the learning. This was introduced due to the following reasons

- **Stability:**By using a soft update mechanism, updates to the target network are smoother and less disruptive compared to hard updates. This helps in maintaining stability in learning as the Q-values do not fluctuate abruptly. It is particularly useful in environments where the state space is large or complex.
- **Periodic Hard Resets:**While frequent soft updates help in stabilizing and speeding up the learning, occasional hard updates ensure that any small errors or deviations

that accumulate over time in the target network do not impact the long-term performance. Performing a hard update every 500 episodes resets these errors, aligning the target network closely with the main network.

- **Balanced Convergence:**The blend of soft and hard updates provides a balance between stability and convergence speed. Soft updates alone might slow down the convergence as the target network changes very slowly, whereas hard updates alone might destabilize the training. The combination ensures that the network converges at a good pace while retaining the necessary stability.
  Overall, by reducing the interval for hard updates and introducing regular soft updates, the learning process becomes more adaptive and robust. This method leverages the benefits of both stability and responsiveness, ensuring that the target network provides a reliable yet current baseline for evaluating future actions. This results in a more effective and efficient learning process, better suited to the complexities of the environment.

E. Updated Reward Function

The reward function in reinforcement learning (RL) acts as the guiding principle for an agent's behavior, determining how it should interact with its environment to achieve its goals. This function is indeed one of the most crucial components of the RL setup because it directly influences the learning and decision-making process of the agent. The reward function explicitly defines what the agent is supposed to optimize. It translates the objectives of an RL problem into a numerical form that the agent can understand and use as a basis for learning. Without a clear reward function, the agent would not know what actions are desirable The old reward function (from reference) is as follows:

- **Yaw Angle reward:**

$$r_{yaw} = -1 \times |\text{goal\_angle}| \tag{1}$$

- **Angular velocity reward:**

$$r_{vangular} = -1 \times (\text{action\_angular})^2 \tag{2}$$

- **Distance to goal reward:**

$$r_{distance} = \frac{2 \times \text{goal\_dist\_initial}}{\text{goal\_dist\_initial} + \text{goal\_dist}} - 1 \tag{3}$$

- **Proximity to obstacle reward:**

$$\text{if min\_obstacle\_dist} < 0.22 : r_{obstacle} = -20 \tag{4}$$

$$\text{else: } r_{obstacle} = 0 \tag{5}$$

- **Linear velocity reward:**

$$r_{vlinear} = -1 \times ((0.22 - \text{action\_linear}) \times 10)^2 \tag{6}$$

- **Total reward per step:**

$$\text{reward} = r_{yaw} + r_{distance} + r_{obstacle} + r_{vlinear} + r_{vangular} - 1 \tag{7}$$

- **Reward per episode (Based on outcome):**
  - if success: reward = 2500
  - else if collision: reward = -2000

The modified reward function (ours) is as follows:

- **Yaw Angle reward:**

$$r_{yaw} = -0.5 \times |\text{goal\_angle}| \tag{8}$$

- **Distance to goal reward:**

$$r_{distance} = \frac{5 \times \text{goal\_dist\_initial}}{\text{goal\_dist\_initial} + \text{goal\_dist}} - 1 \tag{9}$$

- **Proximity to obstacle reward:**

$$\text{if min\_obstacle\_dist} < 0.25 : r_{obstacle} = -25 \tag{10}$$

$$\text{else: } r_{obstacle} = 0 \tag{11}$$

- **Total reward per step:**

$$\text{reward} = r_{yaw} + r_{distance} + r_{obstacle} - 1 \tag{12}$$

- **Reward per episode (Based on outcome):**
  - if success: reward = 3500
  - else if collision: reward = -2500

The reward function was changed for following reasons:

- In yaw angle reward, decreasing the coefficient from -1 to -0.5 reduces the penalty for deviating from the goal angle. It was observed that the agent was giving more priority in taking straight line paths to the goal, which resulted in the agent crashing with obstacles in tight spaces and edge conditions. To ease this behavior of the robot, the penalty for orienting away from the goal was reduced.
- In distance to goal reward, the coefficient in the reward was increased from 2 to 5. This makes the agent prioritize reaching the goal quicker. So, the agent gives priority in reducing the relative distance between its current position and the goal, leading to quicker navigation to the goal.
- In proximity to obstacle reward, Increasing the threshold from 0.22 to 0.25 for triggering a large penalty (-25) indicates a stricter avoidance behavior. It was observed that the robot was getting too close to the walls while moving to the goal, resulting in frequent collisions with both static and dynamic obstacles. To encourage safe behaviour by making the robot avoid going too close to the obstacles, the threshold was increased, and the penalty was increased.
- In reward per episode, the outcome rewards were tweaked to indicate a stronger importance on successful completion and avoidance of collisions.

V. DQN Learning Approach

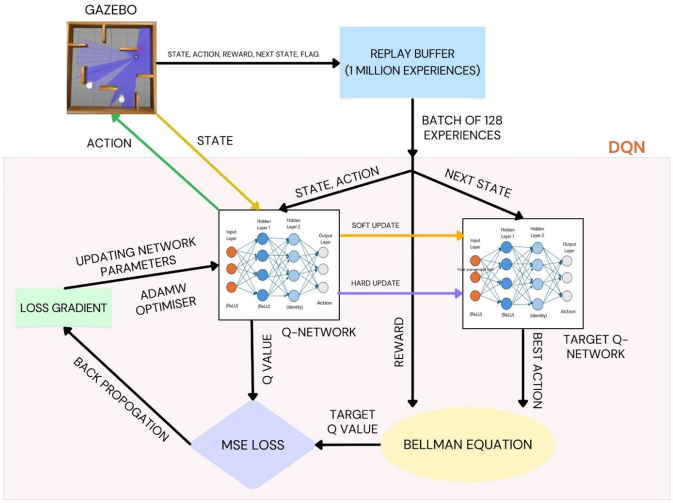

Fig. 5. DQN Learning Network

Based on the changes made to the architecture and the parameters chosen, the learning by the Deep Q network is explained in this section. The overview of the learning process is sequentially described below:

1) The agent interacts with the environment and collects experiences in the form of state, action, reward, and next state at each step. The state and the next state consist of 60 LIDAR scan samples and the relative position and orientation to the goal extracted from Odometry, before and after taking the action (selected linear and angular velocity for a step time of 0.01 seconds).
2) The experiences collected by the robot are stored in a replay buffer. The buffer can store up to 1 million experiences.
3) A batch of 128 experiences is randomly sampled from the replay buffer.
4) The state (62) and the action (2) selected are given as an input to the Q-Network. The Q-Network outputs the approximated Q-Value for that particular action.
5) The next state (62) is given as an input to the Target Q-Network. The Target Q-Network outputs the approximated Q-Values for all possible actions (12). The action corresponding to the highest approximated Q-Value is selected as the best action. The Q-Value of the best action is selected for further computation.
6) This Q-Value of the best action obtained from the Target Q-Network is then given as an input along with the reward to the Bellman equation to obtain the Target Q-Value.
7) The Q-Value and the Target Q-Value are then compared to get the MSE loss.
8) This MSE loss is backpropagated through the Q-Network to get the loss gradients.
9) The loss gradients are then used to update the Q-Network parameters using the AdamW optimizer.
10) The Target Q-Network gets updated at specified intervals. A soft update occurs every 10 episodes and a hard update occurs every 500 episodes.
11) During exploitation or testing, the current state of the agent is given as an input to the Q-Network, and it outputs the approximated Q-Values based on the current network parameters. The best action is selected corresponding to the largest Q-Value, and it is given as an input to the agent.

VI. Results

In the previous section, we described a network architecture, target update methodology and action set which were designed by us for better performance of the TurtleBot. Using this novel approach and the hyperparameters defined in the following subsection, we trained the agent for 2700 episodes. The outcomes of the training and testing are further discussed under subsection "outcomes". For comparison, we also trained the agent with the reference architecture (obtained from the GitHub repository) for 7000 episodes. The comparison of the training and the behavior of the agent is further explained under subsection "comparison".

A. Hyperparameters

The hyperparameters used in training and their significance are mentioned in the below table

TABLE III
Hyperparameters and their Roles

Parameter	Value	Role of the parameter
Action Size	12	Defines the number of possible actions the agent can take
Hidden Size	1024	Defines the number of neurons in the hidden layer of the neural network, which impacts the model's capacity to learn complex relationships
Batch Size	128	Specifies the number of experiences sampled from the replay buffer for each training update. It affects the balance between exploration and exploitation
Buffer Size	1000000	Defines the maximum number of experiences stored in the replay buffer. Larger the buffer the more is the learning from a wide range of situations but requires more memory.
Discount Factor	0.99	Controls how much importance the agent assigns to future rewards compared to immediate rewards. A higher discount factor prioritizes long-term goals.
Learning Rate	0.003	The step size the optimizer takes when updating the neural network weights during training. A smaller learning rate leads to more cautious updates, while a larger one can lead to faster learning but also instability.
Tau	0.003	It controls the soft update of the target network weights towards the online network weights. A smaller value of tau leads to slower updates, making the target network more stable.

Step Time	0.01	Time considered for each step of the robot.
Loss Function	torchf. smooth_l1_loss	Defines the loss function used to train the difference between the predicted Q-values from the Neural Network and the target Q-values. Also called the Huber Loss, this is a combination of the benefits of both MSE and absolute error. It is less sensitive to outliers and minimizes the squared difference, encouraging the network to get closer to the target Q-value for those states and actions.
Epsilon	1.0	This parameter controls the agent's exploration rate, which is the probability of taking a random action instead of the one suggested by the current policy (exploitation). It's initially set to 1.0 (full exploration) and decays over time using epsilon decay and epsilon minimum.
Epsilon Decay	0.9995	Defines the rate at which epsilon decreases.
Epsilon Minimum	0.075	Sets the minimum exploration rate, ensuring some degree of randomness even after learning.
Target Update Frequency	500	Determines how often the target network's weights are synchronized with the behavior network's weights. This is the number of episodes after which the hard update takes place.
Soft Update Rate	10	This is the number of episodes after which the soft update takes place.
Observe Steps	25000	At training start random actions are taken for N steps for better exploration.
Arena Length and Width	4.2 meters	Specifies the dimensions of the environment in which the agent operates, which can be used to set boundaries or calculate distances within the simulation.
Lidar Distance Cap	3.5 meters	This parameter caps the maximum distance it can measure, useful for limiting the perception field.
Threshold Collision	0.13 meters	Represents the minimum distance from an object at which a collision is considered to have occurred, used to calculate rewards or penalties.
Goal Threshold	0.20 meters	Sets how close the agent needs to get to a goal to consider it reached.

B. Training Outcome

The novel architecture that we explained in the contributions section was used for training the model. The training performance of the model is shown in figures below:

Inference on Episode Outcome Trends: In the above figure we can observe the trends of different episode outcomes against time during the training of the DQN model. The x-axis represents the number of episodes, while the y-axis represents the cumulative count of each outcome types. In the early episodes, from 0 to 300, there is a high number of collisions, both with static (red) and dynamic obstacles (cyan). This

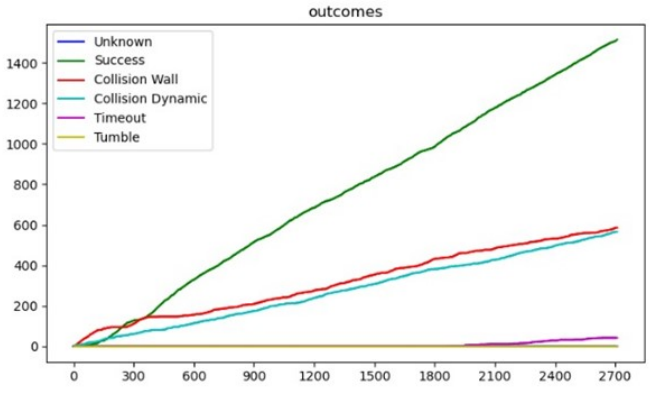

Fig. 6. Network Training Outcome

shows that the model is initially struggling to avoid obstacles and reach the goal, resulting in frequent collisions. The success rate (green) is very low at the beginning, indicating that the model has not yet learned an effective strategy for autonomous navigation and obstacle avoidance. As training progresses, between 300 to 1200 episodes, there is increase in the number of successful episodes (green). This indicates that the model is learning and improving its navigation strategy over time. The rate of collisions with static (red) and dynamic obstacles (cyan) starts to be stabilizing, indicating that the model is avoiding obstacles better. Between episodes 1200 and 2700, The success rate continues to rise further, indicating that the model is learning, improving, and becoming better at autonomous navigation and obstacle avoidance. The rates of collisions with static and dynamic obstacles are seen to be decreasing. From this, we can infer that the model is learning to avoid obstacles more effectively as training progresses. Overall, the trends highlight the model's learning process and its gradual improvement in navigation and obstacle avoidance, showcasing the effectiveness of the training approach.

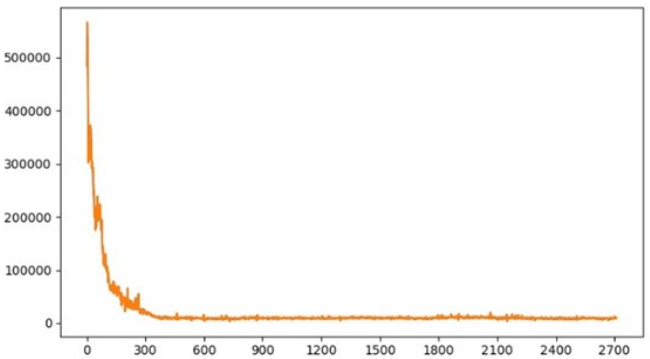

Fig. 7. Average MSE Loss

The above figure shows the average MSE loss over the course of training episodes. The x-axis represents the number of episodes, while the y-axis indicates the average MSE loss.

At the beginning of training, the MSE loss is extremely high. This high initial loss indicates that the Q-values predicted by the network are far from the target Q-values. Then we can observe a sharp decrease in the MSE loss within the first 100 episodes. This rapid decline suggests that the network is quickly learning to approximate the Q-values more accurately as it receives feedback from the environment and updates its weights. Over the next few episodes, the MSE loss continues to decrease. This continued decline indicates ongoing learning and refinement of the Q-function. From around episode 600, the MSE loss stabilizes and remains consistently low. This stability indicates that the network has reached a point where the Q-values are relatively accurate, and the policy is near-optimal.

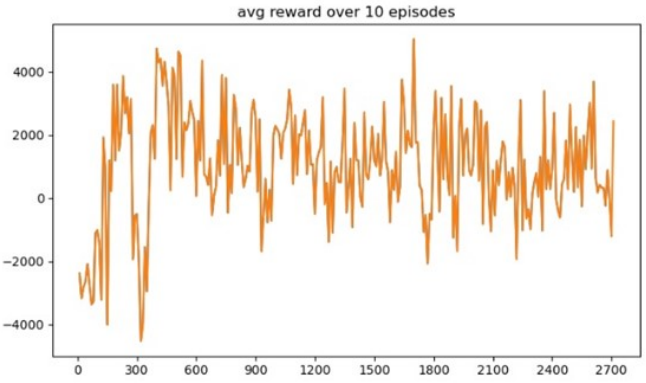

Fig. 8. Average Reward

The above figure shows the trend of the average reward over 10 episodes during the training of a DQN. The x-axis represents the number of episodes, while the y-axis indicates the average reward. In the early episodes, the average reward fluctuates significantly, ranging from highly negative to highly positive values. This shows that the model is exploring various actions and receiving inconsistent feedback from the environment. It can also be observed that the model is initially taking poor decisions that is resulting in not optimal outcomes. But, as training progresses, it is observed that there is an increase in the average reward. This upward trend indicates that the model is learning, resulting in better decision-making and higher rewards. It is also observed that the fluctuations in reward become less, showing that the model is stabilizing and making better decisions. Even though the graph shows an overall positive trend, fluctuations are still present in the average reward values. These fluctuations suggest that the model continues to explore and is trying to respond to varying environment conditions and trying to refine its strategy. Overall, the trends in the figure highlight that the model is learning and adapting as training progresses, leading to higher and more stable average rewards.

C. Testing Outcome

After training for 2700 episodes, the model was tested with the model weights saved at 1200 episodes and 2600 episodes. After 1200 episodes of training, the model achieved a success rate of 75% after testing for 33 episodes as seen in the following video: https://youtu.be/a864T2ihajU

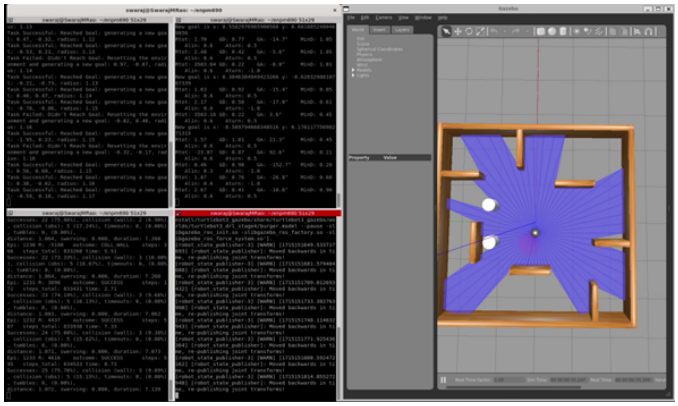

Fig. 9. After 1200 Episodes of Training

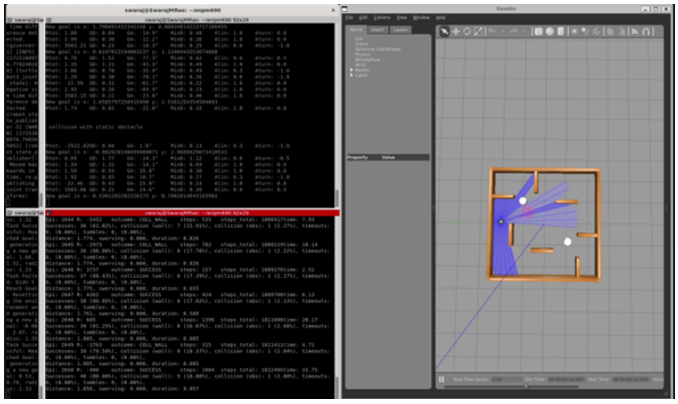

Fig. 10. After 2600 Episodes of Training

The testing outcome after 1200 episodes and 2600 episodes of training is summarised in the table below:

TABLE IV
PERFORMANCE COMPARISON BETWEEN 1200 AND 2600 EPISODES

	1200 Episodes	**2600 Episodes**
Number of episodes tested	33	50
Number of successes	25	40
Percentage of successes	75.76%	80.0%
Number of collisions with static obstacle	3	9
Percentage of collisions with static obstacle	9.09%	18%
Number of collisions with dynamic obstacle	5	1
Percentage of collisions with dynamic obstacle	15.15%	2%

D. Training Comparison

In the previous subsection we have described the results that we obtained by training a DQN model with our architecture.

But we wanted to compare the performance of our model with a reference model. For that, we trained a DQN model using the same architecture mentioned in the GitHub repository that we are referring to for the project. An overview of this reference architecture is given in the "our contributions" section. A DQN model was trained for 10500 episodes using this reference model and the performance is given by the 3 graphs below.

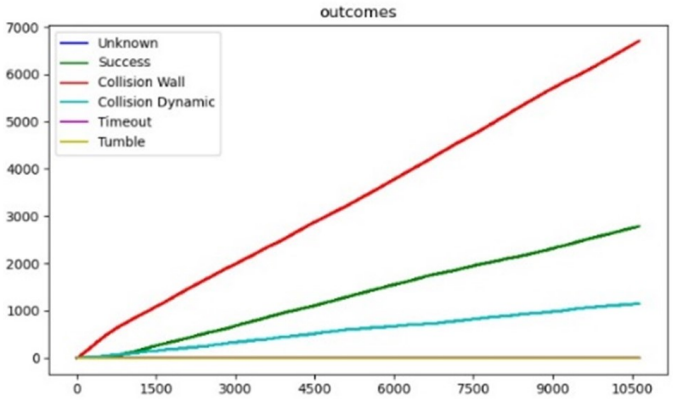

Fig. 11. Training Outcomes v/s Episodes

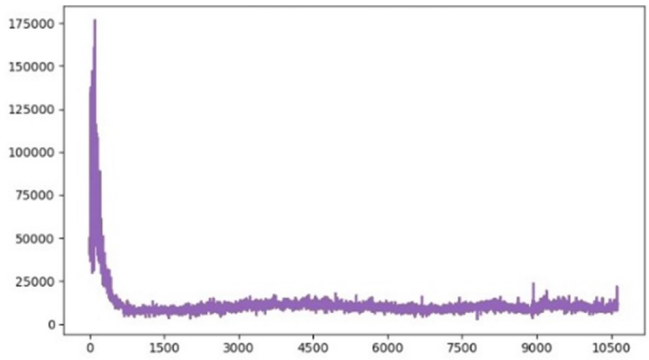

Fig. 12. Loss v/s Episode

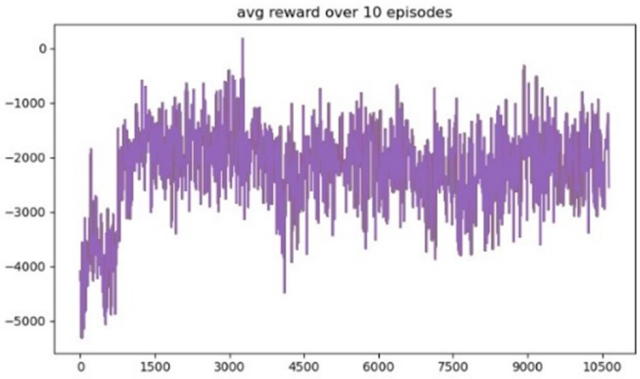

Fig. 13. Reward v/s Episode

Hence, the following comparisons can be made between the model trained using our architecture and the model trained using reference architecture.

TABLE V
COMPARISON BETWEEN OUR MODEL AND REFERENCE MODEL

Our Model	**Reference Model**
Initially, a high collision rate is observed but it stabilizes, and the success rate increases overtaking the number of collisions. This indicates that the model has effectively learned an optimal strategy for obstacle avoidance and autonomous navigation.	The collision rate does not stabilize even after 10000 episodes of training. There is an increase in the number of successful episodes, but the number of collisions dominates the number of collisions.
It is observed that the MSE loss stabilizes around 300 episodes of training. This indicates a faster rate of convergence to an optimal strategy.	The MSE loss stabilizes at around 1000 episodes of training, showing slower convergence.
In our model, a gradual increase in average reward is observed with relatively low variability and minor fluctuations after the model converges to an optimal strategy.	In the reference model, the average reward initially shows a gradual increase but shows high fluctuations even after converging to an optimal strategy.

VII. CONCLUSION

In this project, we designed a Deep Q-Network architecture of our own, evaluated a model trained on this architecture and compared the results with a reference model. The results demonstrated that our method achieves all the objectives that we initially planned to do, and it outperforms the reference model. Through this project, we were able to gain significant knowledge on how deep reinforcement learning architecture and model works and how different factors influence the training and the outcome. By studying the method and architecture in deep, we were able to logically change some parameters and build an architecture of our own to train a working model. Our model showed a higher success rate and better convergence than the reference model.

A. Challenges Faced

While working on this project, we faced several challenges that halted our progress and required attention and troubleshooting. Sole of these challenges are:

- ROS2 Version Mismatch: As we planned to use ROS2 with Gazebo as our environment, we initially faced some compatibility issues related to ROS packages that we had to resolve to proceed. Ensuring all dependencies and packages were aligned with a compatible ROS2 version (Humble in our case) was critical but time-consuming.
- Time Complexity of Training: Training the DQN models was a time-consuming process, with cumulative training time exceeding 200 hours. Balancing training time with model performance was a persistent challenge.
- Graphics Driver Issues: We experienced significant interruptions due to problems with graphics drivers. These issues led to system crashes and halted our progress as we worked on identifying and resolving driver conflicts.
- Parameter Tuning: Finding the optimal parameters for the DQN models was a complex and iterative process. Initial parameters often did not yield good results; hence, multiple rounds of training and validation were conducted.
- System memory issues: The system experienced instability with nodes crashing frequently due to memory issues. These crashes disrupted the training process.

In spite of facing these challenges, we successfully developed and trained a DQN model for autonomous navigation. Addressing these issues provided valuable learning experience.

B. Future Work

Future work could explore using other deep reinforcement learning techniques for better model performance. Different DRL techniques can also be combined to find the right combination that can outperform standalone DQN model for achieving the task. The integration of additional sensor data to further enhance the resolution with which the environment is perceived, navigation accuracy and robustness can also be explored. Moreover, testing our method in different dynamic environments would provide deeper insights into its adaptability and generalization capabilities.

VIII. References

1) S. Zhou, X. Liu, Y. Xu and J. Guo, "A Deep Q-network (DQN) Based Path Planning Method for Mobile Robots," 2018 IEEE International Conference on Information and Automation (ICIA), Wuyishan, China, 2018, pp. 366-371, doi: 10.1109/ICInfA.2018.8812452. keywords: Path planning;Mobile robots;Training;Planning;Feature extraction;Data preprocessing;Global Path planning;DQN;Mobile Robot,
2) L. Tai and M. Liu, "A robot exploration strategy based on Q-learning network," 2016 IEEE International Conference on Real-time Computing and Robotics (RCAR), Angkor Wat, Cambodia, 2016, pp. 57-62, doi: 10.1109/RCAR.2016.7784001. keywords: Robot sensing systems;Learning (artificial intelligence);Collision avoidance;Feature extraction;Training;Games,
3) H. Soni, R. Vyas and K. K. Hiran, "Self-Autonomous Car Simulation Using Deep Q-Learning Algorithm," 2022 International Conference on Trends in Quantum Computing and Emerging Business Technologies (TQCEBT), Pune, India, 2022, pp. 1-4, doi: 10.1109/TQCEBT54229.2022.10041614. keywords: Q-learning;Quantum computing;Computational modeling;Neural networks;Games;Market research;Autonomous automobiles;Deep Q-Learning;Self-Driving autonomous car;neural network,
4) Feng S, Sebastian B, Ben-Tzvi P. A Collision Avoidance Method Based on Deep Reinforcement Learning. Robotics. 2021; 10(2):73. https://doi.org/10.3390/robotics10020073
5) X. Qiu, K. Wan and F. Li, "Autonomous Robot Navigation in Dynamic Environment Using Deep Reinforcement Learning," 2019 IEEE 2nd International Conference on Automation, Electronics and Electrical Engineering (AUTEEE), Shenyang, China, 2019, pp. 338-342, doi: 10.1109/AUTEEE48671.2019.9033166. keywords: Navigation;Heuristic algorithms;Machine learning;Robot sensing systems;Neural networks;Collision avoidance;robot navigation;deep reinforcement learning;deep Q network;reward shaping,
6) Escobar-Naranjo J, Caiza G, Ayala P, Jordan E, Garcia CA, Garcia MV. Autonomous Navigation of Robots: Optimization with DQN. Applied Sciences. 2023; 13(12):7202. https://doi.org/10.3390/app13127202
7) de Sousa Bezerra CD, Teles Vieira FH, Queiroz Carneiro DP. Autonomous Robotic Navigation Approach Using Deep Q-Network Late Fusion and People Detection-Based Collision Avoidance. Applied Sciences. 2023; 13(22):12350. https://doi.org/10.3390/app132212350
8) Carlos Daniel de Sousa Bezerra, Flávio Henrique Teles Vieira, and Daniel Porto Queiroz Carneiro. "Autonomous Robotic Navigation Approach Using Deep Q-Network Late Fusion and People Detection-Based Collision Avoidance." Applied Sciences, 2023, 13, 12350. DOI: 10.3390/app132212350.
9) https://www.geeksforgeeks.org/deep-q-learning/
10) https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc