NAVIGATION OF A UGV FOR TOMATO CROPS IN GREENHOUSES USING REINFORCEMENT LEARNING

JUAN SEBASTIAN RADA REY
Director: Ph.D., M.Sc. CARLOS BORRÁS PINILLA
Deputy director: M.Sc. CARLOS ALBERTO FLÓREZ ARIAS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Engineering

Universidad Industrial de Santander
Facultad de Ingenierías Fisicomecánicas
Escuela de Ingeniería Mecánica
Maestría en Ingeniería Mecánica
2025

Contents

List of Figures

List of Tables

ABSTRACT

TITLE: NAVIGATION OF A UGV FOR TOMATO CROPS IN GREENHOUSES USING REINFORCEMENT LEARNING*

AUTHOR: JUAN SEBASTIAN RADA REY.**

KEY WORDS: DRL, AGRO-ROBOTICS, NAVIGATION.

DESCRIPTION:

The demand for increased food production continues to rise, driving interest in precision agriculture and agro-robotics as effective means to enhance crop productivity and resource efficiency. One of the critical challenges in this domain is enabling robust and reliable autonomous navigation for robots in greenhouse environments, which are typically constrained, dynamic, and GPS-denied. This study focuses on the design and implementation of a navigation system for an unmanned ground vehicle (UGV) operating in a simulated tomato greenhouse. The methodology employs a Double Deep Q-Network (DDQN) trained entirely in simulation to address navigation tasks while avoiding collisions with structural elements and crop rows. The proposed system integrates LiDAR-based perception, goal position tracking, and discrete velocity command selection within a reinforcement learning framework. Evaluation was conducted through 30 test episodes using varied starting and goal positions to assess adaptability. Results indicate a 90% success rate in reaching goals without collisions, with the robot demonstrating consistent path planning and effective obstacle avoidance, even in narrow corridors. The integration of prioritized replay memory significantly improved performance by increasing successful episodes and reducing collisions. These findings highlight the potential of DRL-based approaches for greenhouse automation and pave the way for future deployment in real-world agricultural environments.

*Master Thesis.

**Facultad de ingenierías fisicomecánicas. Escuela de ingeniería mecánica. Director Carlos Borrás Pinilla. Codirector Carlos Alberto Flórez Arias.

RESUMEN

TÍTULO: NAVIGATION OF A UGV FOR TOMATO CROPS IN GREENHOUSES USING REINFORCEMENT LEARNING*

AUTOR: JUAN SEBASTIAN RADA REY.**

PALABRAS CLAVE: DRL, AGRO-ROBÓTICA, NAVEGACIÓN.

DESCRIPCIÓN:

La creciente demanda de producción de alimentos impulsa el interés en la agricultura de precisión y la agro-robótica como medios efectivos para aumentar la productividad de los cultivos y optimizar el uso de recursos. Uno de los retos más importantes en este ámbito es lograr una navegación autónoma robusta y confiable para robots en entornos de invernadero, caracterizados por ser espacios reducidos, dinámicos y sin señal GPS. Este estudio se centra en el diseño e implementación de un sistema de navegación para un vehículo terrestre no tripulado (UGV) operando en un invernadero de tomates simulado. La metodología emplea una red neuronal de tipo Double Deep Q-Network (DDQN), entrenada completamente en simulación, para abordar tareas de navegación mientras se evitan colisiones con estructuras y hileras de cultivo. El sistema propuesto integra percepción basada en LiDAR, seguimiento de la posición objetivo y selección de comandos de velocidad discretos dentro de un marco de aprendizaje por refuerzo. La evaluación se realizó en 30 episodios de prueba con distintas posiciones iniciales y de destino para medir su adaptabilidad. Los resultados muestran una tasa de éxito del 90% en alcanzar los objetivos sin colisiones, con una planificación de trayectoria consistente y una evasión eficaz de obstáculos, incluso en pasillos estrechos. La incorporación de memoria de repetición priorizada mejoró notablemente el rendimiento, incrementando los episodios exitosos y reduciendo las colisiones. Estos hallazgos respaldan el potencial de los enfoques basados en DRL para la automatización de invernaderos y sientan las bases para su futura aplicación en entornos agrícolas reales.

*Trabajo de grado.

**Facultad de ingenierías fisicomecánicas. Escuela de ingeniería mecánica. Director Carlos Borrás Pinilla. Codirector Carlos Alberto Flórez Arias.

1 INTRODUCTION

The demand for food will continue to increase during this century [Til+11]. Precision agriculture and agro-robotics seek to provide technological solutions to increase the productivity of crops, this can contribute to facing the challenges that arise from the need to produce more food. Many of these solutions depend on the ability of autonomous vehicles to navigate agricultural environments, which are often quite difficult to navigate.

As deep reinforcement learning has recently been shown to be a very promising approach to robot navigation [ZZ21], this project seeks to develop the navigation of a mobile robot for tomato greenhouses using deep reinforcement learning to determine if this approach fits the requirements of the environment.

This project aligns with the ongoing initiatives pursued by the DICBOT research group within the School of Mechanical Engineering at UIS, particularly within its robotics research hub dedicated to agro-robotics. The group, with its focus on advancing agricultural technologies, has been actively engaged in various projects related to Simultaneous Localization and Mapping (SLAM) for Unmanned Ground Vehicles (UGVs) in agriculture, the control of drones for crop monitoring, and the identification of diseases in leaves, among other endeavors... It should be noted that the research group possessed the necessary infrastructure, a wealth of experience in the field, and extended its support to this particular project.

2 PROBLEM STATEMENT

The problem addressed in this project is the navigation of a UGV in tomato greenhouses, so that it is able to move efficiently and avoid collisions with plants and other obstacles. Navigation in agricultural environments is a complicated task as it requires flexible systems that can work in irregular environments [OMS21]. Due to the complexity of agricultural environments, it has been shown that it is necessary to improve existing navigation methods or to propose new ones and verify their practicability and effectiveness [Gao+18]. Specifically in greenhouses, navigational systems face several challenges, such as a limited space to move, a terrain that may be variable, potential human interactions, and the need to avoid contact with fragile plants. In addition, it is important to consider that greenhouses usually face problems with localization services, such as GPS and that the platforms used will face an environment with high moisture and pesticides. This challenges difficult the implementation of classical navigation techniques such as potential fields, GPS-based or sonar-based.

The problem of navigation in greenhouses is a key problem in agriculture as the automation of main tasks in agriculture such as crop monitoring [AR16], pesticide application or harvesting as well as the implementation of new technologies in precision agriculture depend on the capabilities of autonomous systems to properly navigate safely in the environment.

2.1 Hypothesis

The use of deep reinforcement learning (DRL) algorithms, specifically a DDQN, for UGV navigation in tomato greenhouses will result in more efficient and adaptive navigation patterns to overcome obstacles while moving inside the greenhouse.

3 JUSTIFICATION

Navigation is a major task for a mobile robot. Although there are many techniques for navigation of mobile robots, it has been shown that even though DRL navigation is not perfect all of the time, and sometimes presents performance fluctuations, it is now the most promising technique for achieving breakthroughs in navigation capabilities [ZZ21]. The use of DRL for robot navigation in environments with unknown rough terrains has also been successfully tested [Zha+18]. Taking this into account, it is reasonable to test the use of DRL for navigation in greenhouses as they are complex environments.

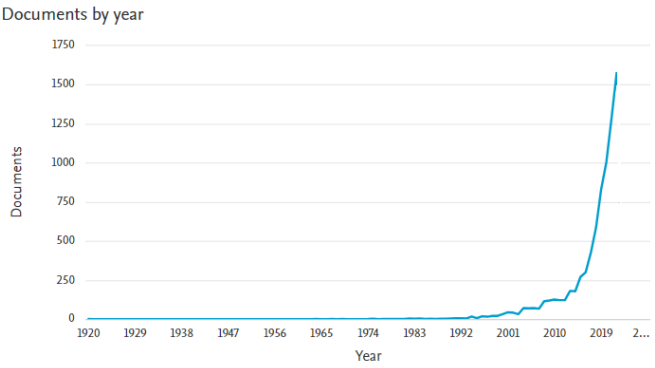


Figure 1: The number of publications on topics related to this research over time.

It is also noticeable the increase of publications that address the topics discussed in this project. Figure 1 shows the growing trend of published scientific documents in agro-robotics, unmanned vehicles (UGVs and UAVs) in agriculture, and the use of reinforcement learning in unmanned vehicles.

4 THEORETICAL FRAMEWORK

4.1 Kinematic model of a four-wheel mobile robot

The platform acquired for this project is the A4WD1 Rover, a four-wheel skid-steering mobile robot, it is shown in figure 2.

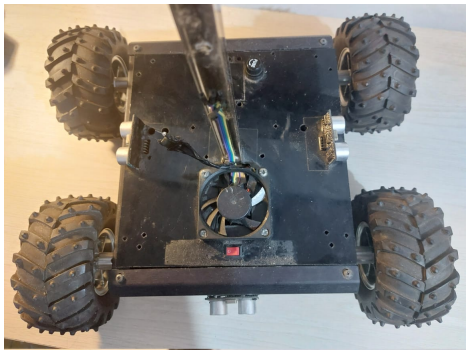


Figure 2: Platform used

The kinematic model of a four-wheel skid-steering mobile robot moving on a planar surface is exposed in [KP04]. The coordinates of the center of mass are in an inertial frame are (X, Y, Z). Since the motion is planar, the Z coordinate will be constant. The linear velocity in the local frame is $v = [v_x, v_y, 0]^T$, and the angular velocity is $w = [0, 0, w]^T$. To define the location of the robot, the X and Z coordinates are needed, as well as the orientation, therefore the state vector will be $q = [X, Y, \theta]^T$ and its derivative $\dot{q} = [\dot{X}, \dot{Y}, \dot{\theta}]^T$.

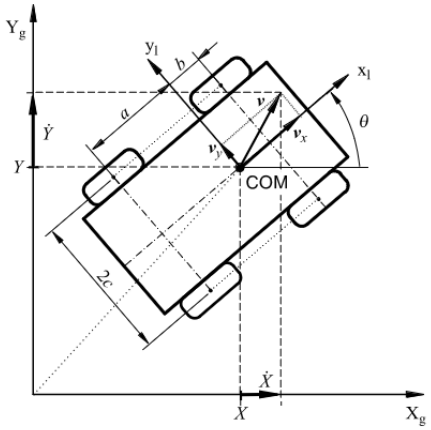


Figure 3: Free body diagram [KP04]

From figure 3, it is possible to see that the velocities $\dot{X}$ and $\dot{Y}$ can be obtained from the local velocities:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{1}$$

When the longitudinal slipping of the wheel is considered negligible, the longitudinal component of the velocity for every i wheel is:

$$v_{ix} = r_i w_i \tag{2}$$

Where r_i is the radius from the center of rotation to the wheel, and w_i is the angular velocity. For a kinematic model, it is necessary to consider the four wheels. Then, the distances from the center of rotation to the wheels is $d_i = [d_{ix}, d_{iy}]^T$, and the distance from the center of rotation to the center of the robot is $d_C = [d_{Cx}, d_{Cy}]^T$.

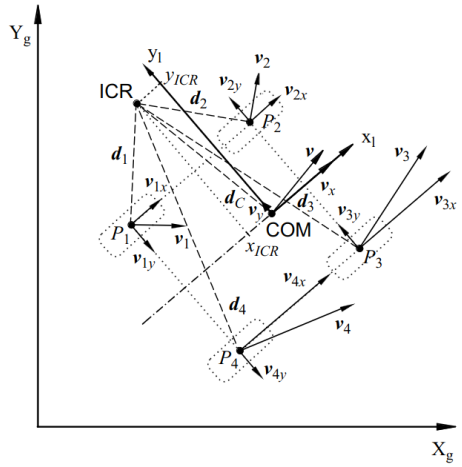


Figure 4: Wheel velocities [KP04]

From figure 4, the next relation between velocities can be stated:

$$\frac{||v_i||}{||d_i||} = \frac{||v||}{||d_C||} = w \tag{3}$$

The coordinates of the instantaneous center of rotation in the local frame are defined as $(x_{ICR}, y_{ICR}) = (-d_{xC}, -d_{yC})$. Therefore:

$$\frac{v_x}{y_{ICR}} = -\frac{v_y}{x_{ICR}} = w \tag{4}$$

From figure 4, the next relations about the distances can be stated:

$$\begin{aligned} d_{1y} = d_{2y} = d_{Cy} + c, \\ d_{3y} = d_{4y} = d_{Cy} - c, \\ d_{1x} = d_4 x = d_{Cx} - a, \\ d_{2x} = d_{3x} = d_{Cx} + b \end{aligned} \tag{5}$$

Where a, b, and c are parameters defined based on the geometry of the robot. Combining (3) and (5):

$$\begin{aligned} v_L = v_{1x} = v_{2x}, \\ v_R = v_{3x} = v_{4x}, \\ v_F = v_{2y} = v_{3y}, \\ v_B = v_{1y} = v_{4y} \end{aligned} \tag{6}$$

Thus, the velocity of the wheels and the velocity of the robot:

$$\begin{bmatrix} v_L \\ v_R \\ v_F \\ v_B \end{bmatrix} = \begin{bmatrix} 1 & -c \\ 1 & c \\ 0 & -x_{1CR} + b \\ 0 & -x_{1CR-a} \end{bmatrix} \begin{bmatrix} v_x \\ w \end{bmatrix} \tag{7}$$

When each effective radius of the wheels is assumed as $r_i = r$, the angular velocities become:

$$w_w = \begin{bmatrix} w_L \\ w_R \end{bmatrix} = \frac{1}{R} \begin{bmatrix} v_L \\ v_R \end{bmatrix} \tag{8}$$

Finally, an expression for the X velocity and the angular velocity is obtained:

$$\begin{bmatrix} v_x \\ w \end{bmatrix} = r \begin{bmatrix} \frac{w_L + w_R}{2} \\ \frac{-w_L + w_R}{c} \end{bmatrix} \tag{9}$$

4.2 Reinforcement Learning

Reinforcement Learning (RL) is one of the main three branches of Machine Learning. It is characterized by having no initial data set, since it creates its data set evaluating interactions of the agent to be trained with the environment.

4.2.1 Basic concepts

Below is a brief description of the basic concepts of Reinforcement Learning.

- Reward: it is a scalar value given depending on the interaction of the agent with the environment. It can be given once every episode or many times during the episode. Usually, it is given every time the agent performs an action.
- Agent: person or thing that takes decisions and performs actions. It can be thought of as the piece of software that is wanted to be trained to solve a problem.
- Environment: everything that is not the agent. The whole universe that exists apart from the agent.
- Actions: they are the things that an agent can do in an environment. They can be discrete or continuous.
- Observations: they are the second source of information that the agent has access to (rewards are the first one). They can be thought of as what the agent can perceive from the environment.

4.2.2 Markov Decision Processes, and the Bellman equation

Markov decision processes are the theoretical basis of reinforcement learning. A Markov Process is a process in which the current state is the only thing necessary for determining the probability of being in a different state in the next time step. The dynamics of the process are defined by a transition matrix which shows the probabilities of moving from one state to another. In a Markov Process, the history of previous states is not necessary.

When a reward is given for going from one state to another, a Markov Process becomes a Markov Reward Process. The return R in a Markov Reward Process can be calculated as :

$$R_t = r_{t+1} + \gamma r_{t+2} + ... = \sum_{r=0}^{\infty} \gamma^k r_{t+k+1} \tag{10}$$

Where r is the reward at different times t and γ is the discount factor that goes from 0 to 1 and can be used to make the present rewards more important than the future ones. It is possible to evaluate the states by using:

$$V(s) = E[R|S_t = s] \tag{11}$$

In this equation, the value V of a state s is defined as the expected return of the state. The expected value is calculated as the average return that is obtained from a state. When actions are added to a Markov Reward Process, it becomes a Markov Decision Process. In a Markov Decision Process, the Reward not only depends on the state but also on the action. Figure 5 depicts how Markov Decision Processes work in DRL.

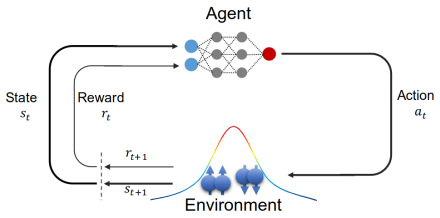


Figure 5: Illustration of a typical Markov Decision Process in DRL [Che+19]

The policy is the set of rules that commands the behavior of the agent. It is formally defined as the probability distribution over actions for every state:

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{12}$$

Q values are used to evaluate the value of states when taking a certain action.

$$Q^{\pi}(s, a) = E_{\tau_t \sim p(\tau_t)} \left[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} | s_t = s, a_t = a \right] \tag{13}$$

The equation above is used to calculate the Q value of a tuple state-action. This equation is similar to the equation for the state value, but it takes the action into account. The Q value of a terminal state is defined as zero:

$$s = \text{terminal} \Rightarrow Q(s, a) := 0, \forall a \tag{14}$$

The objective J of reinforcement learning is to maximize the expected return from an initial state:

$$J(\pi) = V^{\pi}(s_0) = E_{\tau_0 \sim p(\tau_0|\pi)} [R(\tau)] \tag{15}$$

In the above equation, τ represents the trace of states, actions and rewards. The optimal value is the one that achieves more or equal rewards than the other ones. This optimal value function is achieved by the optimal policy $\pi^{\star}$:

$$\pi^{\star}(a|s) = \arg\max_{\pi} V^{\pi}(s_0) \tag{16}$$

The benefit of using Q values instead of V is that the Q values directly determine what every action is worth:

$$a^{\star} = \arg\max_{a \in A} Q^{*}(s, a) \tag{17}$$

The optimal policy can also be found using Q values:

$$\pi^{\star}(s) = \arg\max_{a \in A} Q^{\star}(s, a) \tag{18}$$

In 1957 Richard Bellman proposed a recursive equation to solve discrete optimization problems. This equation shows how the value of sates s depends on future states s', this was called dynamic programming. Bellow the Bellman Equation is presented:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \left[\sum_{s' \in S} T_a(s, s') \left[R_a(s, s') + \gamma \cdot V^{\pi}(s') \right] \right] \tag{19}$$

4.2.3 Deep Value Based Reinforcement Learning

```
def qlearn(environment, alpha=0.001, gamma=0.9, epsilon=0.05):
    Q[TERMINAL,_] = 0 # policy
    for episode in range(max_episodes):
        s = s0
        while s not TERMINAL: # perform steps of one full episode
            a = epsilongreedy(Q[s], epsilon)
            (r, sp) = environment(s, a)
            Q[s,a] = Q[s,a] + alpha*(r+gamma*max(Q[sp])-Q[s,a])
            s = sp
     return Q
```

Figure 6: Q learning pseudo-code [Pla22]

In classic RL, the Q values are stored in a table. Figure 6 shows the pseudo-code for classic Q learning. This works because the state space is small enough to fit into the memory. When there is a high-dimensional state space that no longer fits in memory, as it is the case with most real problems, deep learning methods are used to model the policy [Pla22]. Figure 7 presents the pseudo-code to train a DQN.

```
def dqn:
    initialize replay_buffer empty
    initialize Q network with random weights
    initialize Qt target network with random weights
    set s = s0
    while not convergence:
        # DQN in Atari uses preprocessing; not shown
        epsilon-greedy select action a in argmax(Q(s,a)) # action
             selection depends on Q (moving target)
        sx,reward = execute action in environment
        append (s,a,r,sx) to buffer
        sample minibatch from buffer # break temporal correlation
        take target batch R (when terminal) or Qt
        do gradient descent step on Q # loss function uses target
             Qt network
```

Figure 7: Pseudo-code for DQN [Pla22]

4.2.4 DDQNs

Double Deep Q Networks (DDQNs) are a variant of DQNs. They are proposed as an improvement to DQN that addresses the problem of overestimation of Q values and improves training stability [VGS16]. Standard Q-learning suffers from overestimation bias, where Q-values become inflated due to using the same network for action selection and evaluation. This leads to unstable training and suboptimal policies.

DDQNs use two networks that have the same architecture, one to choose the best action according to the current policy and the other to evaluate the action. The equation for updating the Q values in a DDQN is presented bellow.

$$Q(s,a) = R + \gamma Q'(s', argmax_{a'} Q(s', a')) \tag{20}$$

Where Q is the value function intended to be learned, R is the reward obtained after taking action a in state s, γ is the discount factor, s' is the next state and Q' is the network that evaluates the action. When training these networks, the target network is periodically updated with the weights of the online network, but this is not done at every step since it would create a "moving target" for the online network to learn from.

4.3 Deep Learning

Deep learning is a branch of machine learning that uses artificial neural networks to learn from data sets. Its development has been deeply inspired by the way in which the human brain works. The neural networks are composed by layers of neurons, and each neuron performs a mathematical operation, the connections between the neurons allow the network to perform complex tasks.

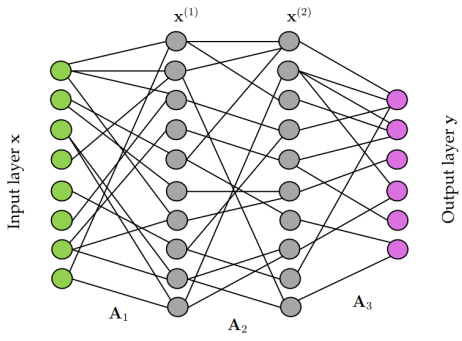


Figure 8: Neural network example [BK22]

Figure 8 shows an example of a neural network with an input layer x, an output layer y, and two hidden layers x^1 and x^2. The equations for the values obtained in each layer are:

$$\begin{aligned} x^1 &= A_1 x \\ x^2 &= A_2 x \\ y &= A_3 x^2 \end{aligned} \tag{21}$$

The output is written as:

$$y = A_3 A_2 A_1 x \tag{22}$$

For a network with n layers the output would be:

$$y = A_n A_{n-1} ... A_2 A_1 x \tag{23}$$

4.3.1 Activation functions

Activation functions take the numerical output of each layer and produce a new output. Using these activation functions in the presented example:

$$\begin{aligned} x^1 &= f_1(A_1, x), \\ x^2 &= f_2(A_2, x^2), \\ y &= f_3(A_3, x^2) \end{aligned} \tag{24}$$

For the case in which there are n layers and activation functions are used, the output would be:

$$y = f_M(A_M, ..., f_2(A_2, f_1(A_1, x)), ...) \tag{25}$$

In order to use back-propagation, the activation functions used must be differentiable. Next, the most commonly used activation functions:

$$\text{Purelin: } f(x) = x \tag{26}$$

$$\text{Sigmoid: } f(x) = \frac{1}{1 + e^{-x}} \tag{27}$$

$$\text{tanh: } f(x) = \tanh x \tag{28}$$

$$\text{Binary step: } f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{29}$$

$$\text{poslin: } f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{30}$$

4.3.2 Back-propagation

The back-propagation algorithm makes use of the compositional structure of neural networks to formulate an optimization problem for adjusting the network's weight values. Its formulation is aligned with the gradient descent optimization.

In back-propagation, the squared error (or any other measure of error) is computed in the output layer. Then, it is propagated back until it reaches the first layer. For a network with one node and one layer, the output y is:

$$y = g(z, b) = g(f(x, a), b) \tag{31}$$

Computing E as the squared error:

$$E = \frac{1}{2}(y_0 - y)^2 \tag{32}$$

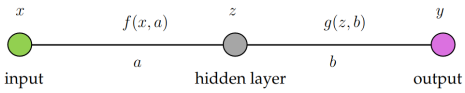


Figure 9: Back-propagation example [BK22]

To compute the change of E with respect to a, the chain rule is used:

$$\frac{\partial E}{\partial a} = -(y_0 - y)\frac{dy}{dz}\frac{dz}{da} = 0 \tag{33}$$

Now, the net parameters a and b can be updated:

$$a_{k+1} = a_k - \delta\frac{\partial E}{\partial a_k} \tag{34}$$

$$b_{k+1} = b_k - \delta\frac{\partial E}{\partial b_k} \tag{35}$$

If a linear activation function is used:

$$f(\xi, \alpha) = g(\xi, \alpha) = \alpha\xi \tag{36}$$

Then, for the example:

$$\begin{aligned} z &= ax \\ y &= bz \end{aligned} \tag{37}$$

The gradient is calculated as:

$$\begin{aligned} \frac{\partial E}{\partial a} &= -(y_0 - y)\frac{dy}{dz}\frac{dz}{da} = -(y_0 - y)bx \\ \frac{\partial E}{\partial b} &= -(y_0 - y)\frac{dy}{db} = -(y_0 - y)z = -(y_0 - y)ax \end{aligned} \tag{38}$$

For a network with n layers, the output would be:

$$\frac{\partial E}{\partial a} = -(y_0 - y)\frac{dy}{dz_n}\frac{dz_n}{d_{zn-1}}...\frac{dz_2}{dz_1}\frac{dz_1}{da} \tag{39}$$

Computing the new weights:

$$W_{k+1} = W_k - \delta\nabla E \tag{40}$$

Or, for each element:

$$w^j_{k+1} = w^j_k - \delta\frac{\partial E}{\partial w^j_k} \tag{41}$$

Using this algorithm, iterations must be performed until an acceptable value for E is reached.

4.3.3 Stochastic gradient descent algorithm

In the last section, the back-propagation algorithm was presented, in which the update on each iteration was decreasing in the opposite direction of the derivative of the squared error with respect to each element. Computing this for all elements in the training set is computationally inefficient. This is why the stochastic gradient descent algorithm is used.

The stochastic gradient descent (SGD) uses one randomly choosen data point from the training set or a reduced set of randomly chosen data points from the training set to calculate the gradient on each iteration. SGD follows the equation:

$$w_{j+1}(\delta) = w_j - \delta \nabla E_k(w_j) \tag{42}$$

Where $K \in [k_1, k_2, ..., k_p]$ denotes the p randomly chosen data points from the data set. In SGD, the learning rate δ can be variable to accelerate convergence while avoiding taking big steps that might overpass the minimum.

5 STATE OF THE ART

5.1 Navigation of UGVs

The navigation of UGVs is commonly divided in four main areas: localization, mapping, path planing and obstacle avoidance.

For path planing and obstacle avoidance, geometrical methods as Dijkstra's or A* search are known for generating optimal paths on geometric graphs that represent the environment [LLS91]. Their advantage is that they are efficient for static environments, but they struggle when facing dynamic environments with obstacles and complex geometries. Other approaches used are sampling-based planners, these use techniques like Rapidly-exploring Random Trees and Probabilistic Roadmap Planners to explore the environment and then find feasible paths (even in highly complex spaces)[LaV06]. Their disadvantage is that sometimes they may generate sub-optimal paths and that require a very careful tuning process for a good performance.

Learning based methods are a new tendency in the field of path planning and obstacle avoidance. These methods are composed by neural networks trained on data and DRL approaches. These methods allow to navigate without explicit path computations. DRL approaches have shown flexibility and adaptability to diverse environments [Mir+16] and are described as the most promising technique to achieve breakthroughs in navigation capabilities [ZZ21]. The disadvantage of these methods is that the networks trained on data require extensive data sets that are not always available and that both methods are computationally expensive.

The most basic approach used for localization is dead reckoning. This approach uses odometry data from encoders and gyroscopes to estimate the location with respect to the initial position [BF96]. It's main advantage is its low cost but when used over time it accumulates errors, and that leads to significant drifts. Due to these significant errors, it is usually combined with other sensor measurements.

Dead reckoning is not the only localization system that is usually combined with other sensors. Many localization systems combine data from sensors such as Li-DARs, cameras, and inertial units to improve accuracy; this is called sensor fusion. Sensor fusion uses algorithms such as Kalman filters to find a trade-off between the information that comes from the different sensors [Bon08].

One of the ways of localizing in the environment is using GNSS like GPS. These sensors offer good accuracy for localization, however, they may be inaccurate in areas with low signal coverage [DAB13].

The map can be given to the robot or can be done simultaneously with Localization using SLAM techniques [TX21]. SLAM can be done with sensors like Li-DARs, this is ideal for applications that require a wide-angle scanning coverage [Yue+24]; 3D Li-DARs provide a good representation of the environments but are more expansive when compared to 2D ones. Another way of mapping with slam is using cameras for Visual SLAM, this approach has the advantage of being affordable [HKK15] (as monocular cameras are not very expensive). Visual SLAM approaches have to find a trade-off between a detailed map representation that requires a lot of computational power or a representation that depends on sparse features but is more suitable for devices with limited resources [Kaz+22].

5.2 UGVs in agriculture, with a focus in Greenhouses

[Gao+18] exposes a background of the research in wheeled mobile robots used in agriculture. The four wheel chassis is described as the most common chassis structure, it usually uses differential driving or front-wheeled synchronous steering as its driving mode due to stability and simplification of the operational process. Two wheel differential driving models are described as being simpler, having lower costs and a better performance at obstacle avoidance, but usually aren't robust enough.

[Bot+22] reviews diverse robot systems in agriculture. It states that while drone solutions are standard drones with sensors, mobile platforms are specially designed for each application. It found that most ground robots in agriculture are dedicated to monitoring tasks and perform only other more complex tasks in high-value crops. It also exposes the robot Agri.Q developed by the authors for fields with significant inclines, this robot has two skid steering modules, each module has two locomotion units that drive two tires each.

The literature on navigation of mobile robots in greenhouses has been continuously developing in recent years. Mobile platforms used in greenhouses perform tasks such as measuring infrared temperature and soil moisture [Rui+16] or using image recognition technologies to count and classify fruits [SCK21] or to look for diseases [AR16]. These platforms can be completely autonomous, as they usually are in open fields, or can depend on structures of the greenhouses (generally rails) for its guidance [TD17]; while using rails to move is more expensive, using autonomous platforms is a bigger technological challenge.

[Gon+09] and [Lon+10] present navigation systems for greenhouses based on ultrasonic sensors. The main advantage of using ultrasonic sensors is that they are cheap when compared to Li-DARs or RBGD cameras but this proposed systems have the disadvantage of being unable to adapt to dynamic environments.

[AR16] proposed a design for a wheeled mobile robot for greenhouses. The robot is a line follower equipped with a camera to take pictures of the crops. The decision of using a line follower system instead of using Li-DAR or camera navigation is because an optimal cost-benefit ratio is pursued. GPS navigation based was not used because of the indoors attenuation of the GPS signal. Problems with localization of mobile platforms while using GPS are also reported in vineyards mountains [Dos+15].

5.3 Deep Reinforcement learning in navigation

The successful use of DRL to win in video games [Mni+13] aroused great interest in the scientific community due to its potential to be applied in other scenarios. Since then, it has been applied in many areas of robotics, such as manipulators [Kal+18] [And+20] [Rus+17], locomotion [Hee+17] [Pen+17] [Tan+18], trajectory tracking [Sri+22] and autonomous driving [Sal+17][SSS16]. DRL is a very promising field in robotics as it is the way to go in machine learning when there is no previous data set, which is usually the case in robotics.

In navigation, DRL has received attention due to its strong representation and experience learning abilities [ZZ21]. The training of navigation systems that use DRL for locomotion purposes is usually carried out in simulated environments as they are faster, cheaper, and safer [Tan+18].

[ZZ21] makes a systematic and comprehensive review of navigation systems in mobile robots using DRL. It exposes 3 challenges: partial observation, sparse rewards, and poor generalization. It exposes possible approaches for overcoming them. This work concludes

that though DRL navigation does not always have perfect behavior, it is the most promising approach in navigation.

[Sur+20] uses an A3C network to train the navigation of a mobile robot. Fused data from a RGBD camera and a Li-DAR, and the orientation towards the goal position were the input to the network, and the output was the linear and angular velocities for the robot. The resulting model was able to move in an unstructured indoor environment.

[Zhu+17] proposes a model for visual navigation that was able to go from simulation to reality without the need of fine-tuning. This was achieved using different targets and scenarios during the training stage. [LY22] also works in visual navigation adding a Li-DAR for obstacle detection. It compares the performance of a DQN with a DDQN, and found better results with the DDQN.

[Mir+16] proposes a deep DRL method for teaching agents to navigate in environments that are big and visually rich. Their approach of augmented DRL with auxiliary objectives leads to a more general navigation strategy and permits an end-to-end learning strategy. The method was evaluated in the deep-mind lab. [Shi+19] proposes an end-to-end method for navigation using DRL in which sparse laser-ranging measurements are taken as input to reduce the reality gap. It proposes the ICM A3C algorithm, which is based on the A3C algorithm. Experimentation performed in simulations showed that this proposed algorithm is better in navigation than the A3C.

6 OBJECTIVES

6.1 General objective

To develop the navigation of an unmanned ground robot in a tomato greenhouse by using a Deep Q network to train the model on a simulated environment so that it can properly navigate.

6.2 Specific objectives

1. To develop a virtual model of the 4WD1 four-wheeled platform that was already acquired by the DICBoT group, which accurately represents the kinematics of the platform.

2. To Train the virtual model in a simulated environment of a tomato greenhouse using Deep Reinforcement Learning for the navigation.

3. To test the ability of the navigational system in the simulated environment to find proper paths and avoid obstacles.

7 METHODOLOGY

The methodology is divided into three parts; each of the parts is directly related to the objectives of the project.

7.1 Developing the model

A virtual model of the 4WD1 platform that faithfully represents the dynamics of the real robot was created. An URDF file was used to describe the geometry of the model and its properties such as inertia and mass. This virtual model will be brought to the Gazebo simulator.

The choice of a four-wheeled (4WD1) platform stems from the need for robustness in the uneven terrain of a greenhouse environment. While two-wheeled robots may offer more agility, they are less stable on such terrain. Although six-wheeled platforms provide even better traction and stability, they are more expansive and complex, making the four-wheeled option a suitable balance between performance and cost.

The control of the four wheels was implemented using ROS2 control libraries for PID control. This libraries can be used in simulations, and can be easily transferred to real applications.

7.2 Training the model

The training stage of the model was carried out in a simulated environment of a tomato greenhouse. It is preferred to make the training inside a simulation as it is more secure and can be done faster.

It was expected to navigate between current positions and target positions relative to the environment, the coordinates of these positions were obtained using a subscriber to the Pose topic on the simulator. For DRL, a DDQN-type neural network was used as it is more stable than a single DQN and it was best suited for discrete action spaces. This was done following [VGS16].

The possible states were defined by the readings of the Li-DAR sensor, the position of the robot, and the position of the target; these parameters were also the input of the network (as in [Sur+20]). The agent was the four-wheeled robot and its space of discrete actions was speed commands. Positive rewards were given when the agent managed to reach the goal and small discounts were given for remaining in states other than the goal.

The Adam optimizer was selected for training the Double Deep Q Network (DDQN) due to its efficiency and robustness. It combines the benefits of adaptive learning rates from AdaGrad and RMSProp, allowing it to handle sparse gradients and varying feature scales effectively. Adam's bias correction and efficient computation make it ideal for reinforcement learning, where data distribution can change dynamically.

The learning rate was decreased to improve convergence and stability by allowing the model to make smaller, more precise updates as it approaches the optimal solution, reducing the risk of overshooting and improving generalization.

A batch size of 64 is commonly used because it provides a balance between noise reduction in gradient estimates and computational efficiency. Larger batch sizes reduce variance in gradient updates, leading to more stable training, while smaller batches introduce more noise but can improve generalization. It is also good that the batch size is a power of 2 as modern GPUs are better optimized to work with these values.

A network size and architecture such as [Sur+20] was used as it was already shown to lead to convergence. An epsilon greedy strategy was used to avoid local minima. Episodes were run using different starting and goal positions until the robot could navigate to the goal while avoiding obstacles.

7.3 Testing the model

To ensure objectivity and deeper insights, quantitative metrics were recorded:

- **Success Rate**: The percentage of successful navigation trials where the robot reaches the goal without collision [Sur+20].
- **Collision Rate**: The number of collisions per trial, including minor touches and complete failures [Sur+20].

The evaluation was performed in a simulated tomato greenhouse environment modeled in Ignition Gazebo. Multiple trials were conducted with different goal positions and robot starting points to test the robustness of the navigation system. A replay memory was implemented after a first evaluation to improve the results.

A successful navigation episode was defined as one when the robot reaches the goal location without collisions. The system will be deemed effective if it achieves a high success rate and consistently follows near-optimal paths with minimal deviations from the shortest path.

8 SCOPE AND RESULTS

8.1 Scope

In this project, a virtual model of the AWD1 rover that accurately represents it was created, this with the intention of carrying out the training of the navigation model within a virtual environment. A virtual model of a tomato greenhouse was also created using the Gazebo simulator. Within this virtual space, the training of the model was carried out.

The navigation system was trained by running various episodes within the virtual environment until performance was acceptable. Then it was evaluated that the virtual model could move correctly in the virtual environment, avoiding obstacles and taking optimal routes. The final model was able to perform routines that would be commonly useful in a tomato greenhouse.

For reasons of time, this project limited the evaluation of the model to be performed only on a simulation.

8.2 Results

1. Master's Thesis: Master's thesis on the topics of agro-robotics and reinforcement learning that led to the presentation of this document.

2. Model: The virtual model of the AWD1 rover was created. Although it is a simplified version of the real one, it provides a realistic representation that allows mimicking the behavior of the platform given motion commands.

3. Navigation: The navigation of a mobile robot in an agricultural environment (tomato greenhouses) was presented using reinforcement learning. This navigation model helped to determine whether the use of DRL as a navigation technique is suitable for greenhouse environments.

4. Scientific publication: The production of a paper or a presentation at an academic congress is to be generated from this project.

9 IMPACT

In a national context, this work is relevant because Colombia is an agricultural producer, in the year 2017 agriculture had a contribution of 8.3% to GDP [Wor22]. One of the crops produced in several municipalities in Colombia is the tomato, the use of greenhouses for the production of tomatoes is very common, since it helps farmers deal with external factors such as rain, changes in temperature, and pest control; and also facilitates the implementation of new technologies [PRB11]. As mentioned before, it is necessary to develop autonomous navigation of mobile robots in greenhouses in order to automate major agricultural tasks such as monitoring, planting, harvesting, and pest control. Because of this, working on the navigation of autonomous vehicles in tomato greenhouses has the potential to facilitate automation, and positively impact the economy; it may also facilitate the incorporation of practices of agriculture 4.0 and this could help Colombian farmers to improve food production and promote the development of national agriculture [MAB15].

The problem of navigation in greenhouses is contained in a bigger problem: The need to increase food production. As world population is continuously increasing (it will get to a peak of 10.4 billion during the 2080s, then it will remain constant[ES22]), it is expected that the global crop demand will drastically increase [Til+11].

The direct impact of this thesis is the development of algorithms for the navigation in agricultural environments, as well as the training of a postgraduate student in agro-robotics research. This thesis is aligned with the work that the DICBOT research group from the Mechanical Engineering School at UIS and its robotics research hotbed have been developing in the area of agro-robotics. The research group has been carrying out projects related to SLAM technologies for UGVs in agriculture, drone control for crops, recognition of diseases in leaves, among others. These have sought to develop technologies that contribute to improving productivity in agriculture.

10 Virtual model of the robot

In this section, the model of the robot is detailed. Figure 10 shows a general view of the virtual model. Gray squares show the files used, while red squares show the applications started when launching the robot.

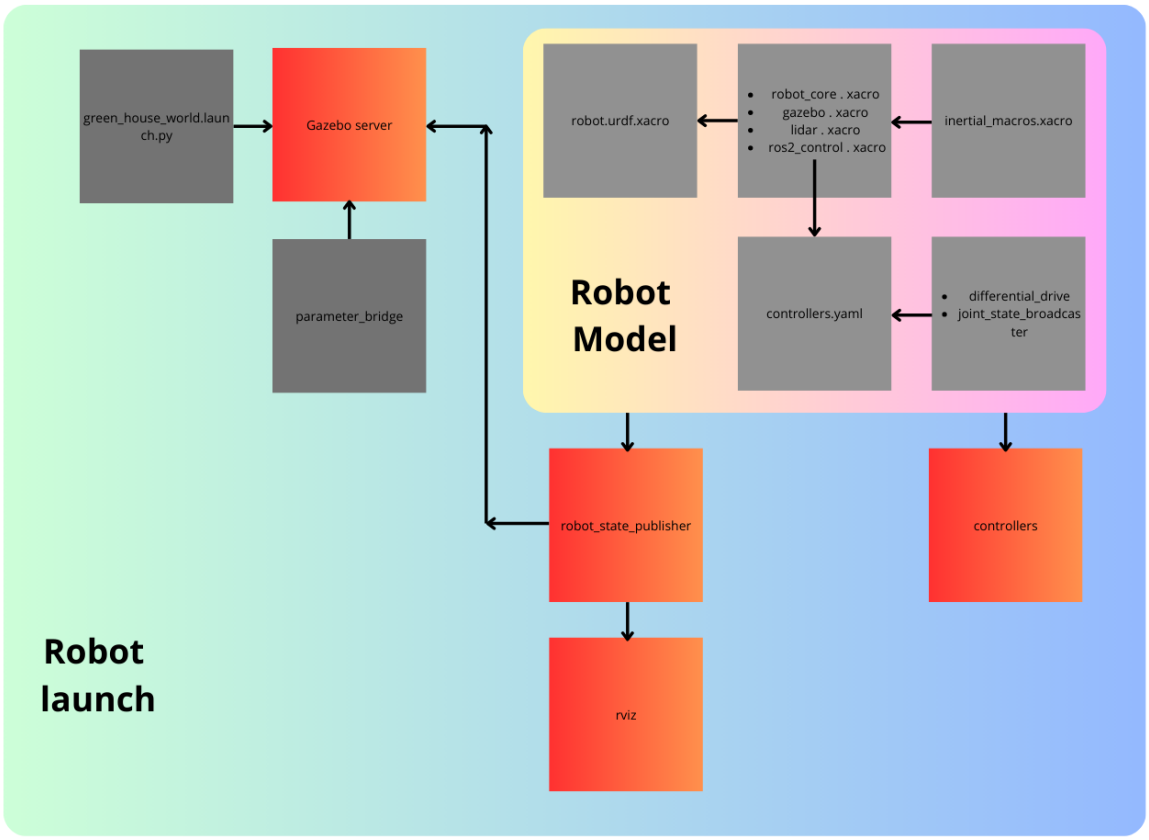


Figure 10: Structure of the virtual model

The file "robot.urdf.xacro" joins the different parts of the URDF: "robot_core.xacro", "gazebo.xacro", "lidar.xacro", and "ros2_control.xacro"; "inertial_macros.xacro" is used by some of these files to optimally define inertia properties. Controllers are defined in "controllers.yaml", the controllers used are differential drive and joint state broadcaster. When the robot model is launched, "robot_state_publisher", "rviz", "Gazebo server", and the controllers are started alongside. "Gazebo server" uses "green_house_world.launch.py" to define the geometry of the greenhouse and the "parameter_bridge" to set communications with ROS2. Figure 11 shows the greenhouse in gazebo while figure 12 shows a drawing of the greenhouse.

10.1 Hardware used and software environment

The computer used for this project had the following hardware:

- Graphics Card: NVIDIA GTX 960M
- Processor: Intel Core i7-6700HQ

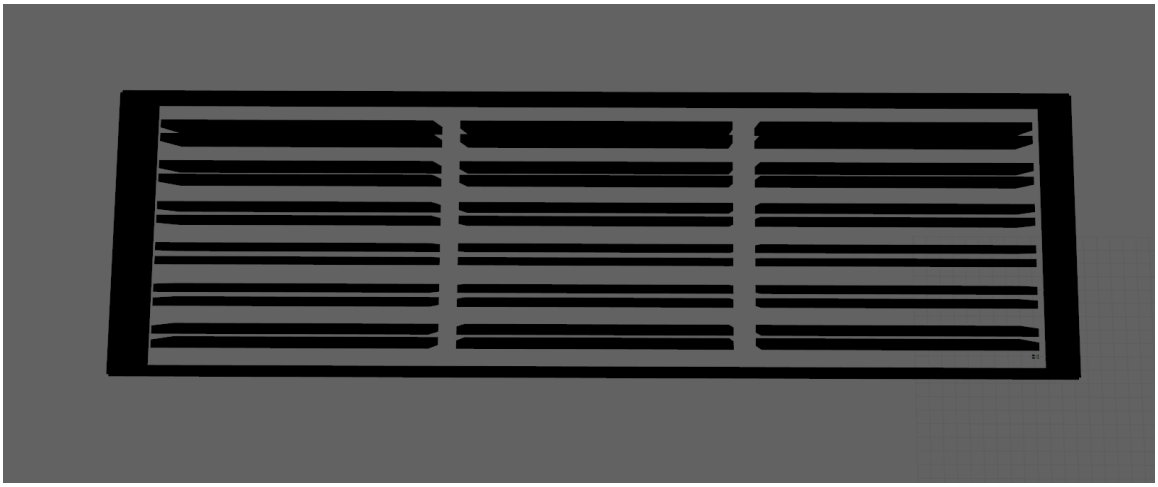


Figure 11: Structure of the greenhouse visualized in the simulator

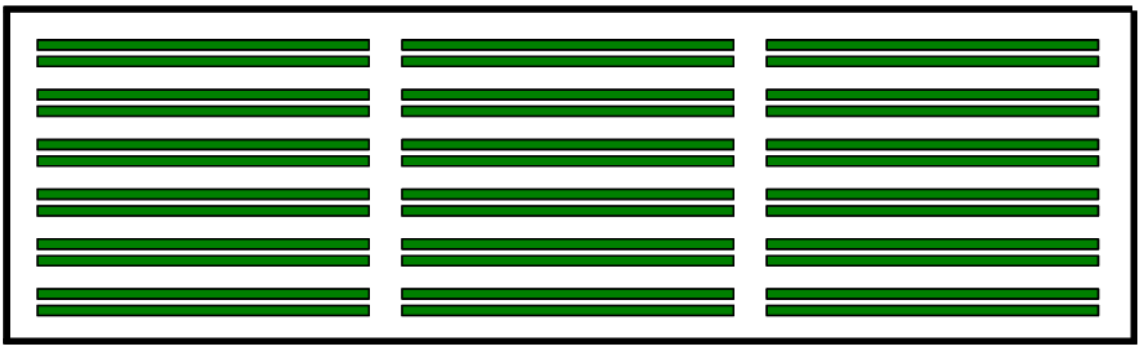


Figure 12: Drawing of the greenhouse's structure

- Memory: 24 GB DDR4 RAM
- Storage: 1 TB SSD

For the development environment, the following libraries were used.

- Operating System: Ubuntu 22.04
- Robot Operating System (ROS): ROS 2 Humble
- Simulation Platform: Ignition Gazebo 6.16.0
- Programming Language: Python 3.10.12
- Numerical Computing Library: NumPy 2.1.2
- Deep Learning Framework: PyTorch 2.4.0 with CUDA 12.2

PyTorch was selected as the deep learning framework for this research due to its pythonic syntax, robust GPU acceleration, and extensive community ecosystem. The versions of ROS2, gazebo ignition, and Ubuntu were chosen together as ROS2 libraries are designed to work with specific versions of Ubuntu, and gazebo ignition has better compatibility with ROS2.

10.2 Overview of the robot description

The robot description file, `robot.urdf.xacro`, is the central configuration file to define the structure, properties, and functionality of the robot. This file utilizes the XML-based `xacro` (XML Macros) format, which is part of the Robot Operating System (ROS). The use of `xacro` allows modularity and reusability in robot description files by enabling the inclusion of other component-specific files.

The `robot.urdf.xacro` file primarily acts as an aggregator that calls "robot_core.xacro", "gazebo.xacro", "lidar.xacro", and "ros2_control.xacro". This file defines the overall robot as a single XML element. The modular design of the robot description file enables the separation of the robot description into distinct components, making it easier to manage and modify. Each included file corresponds to a specific aspect of the robot's functionality, these sub-files will be deeply explained in the next subsections.

Figure 13 shows the coordinate frames of the model visualized in RVIZ. There is a coordinate frame on each part, one in the box, one on each of the four wheels, and one on the lidar.

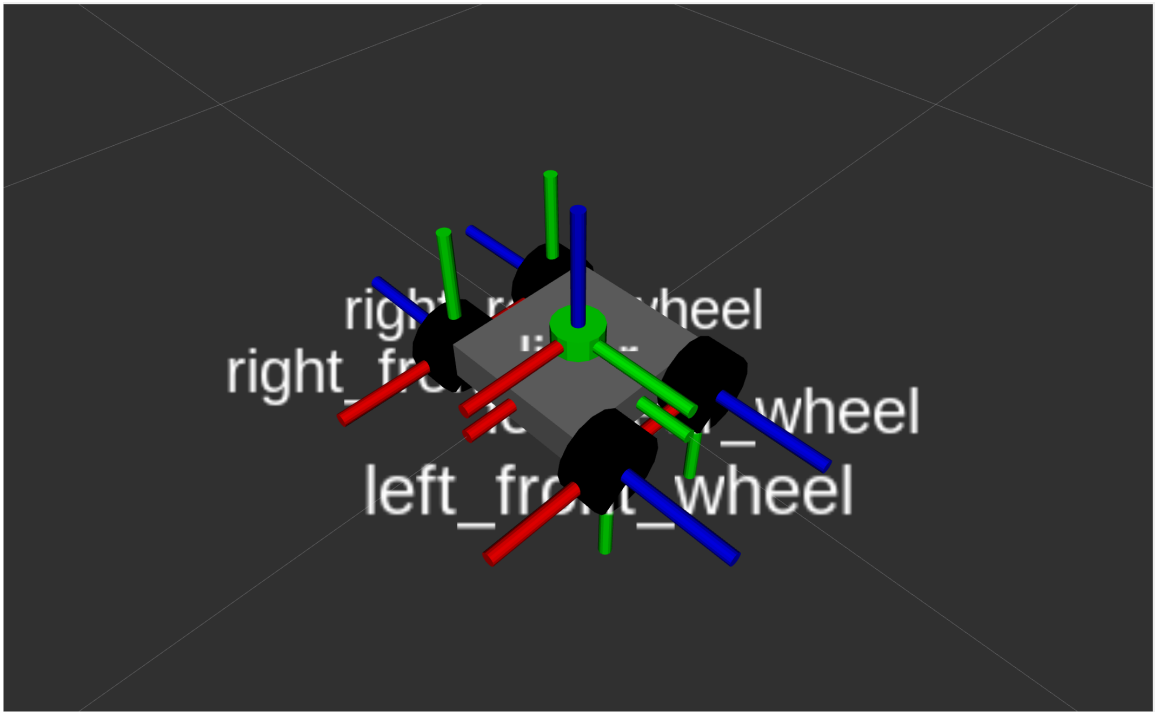


Figure 13: Model of the robot with its coordinate frames

10.3 robot_core.xacro file

The `robot_core.xacro` file describes the links and joints for the chassis and the wheels. It file begins by including a set of macros from the `inertial_macros.xacro` file, these macros simplify the definition of inertial properties for different geometric shapes, such as boxes and cylinders. The file then defines constants for dimensions, masses, and materials, followed by the structural elements of the robot.

Constants: The constants that define the dimensions of the links (in meters) and its masses (in kilograms) are presented next. These parameters were defined based on the dimensions of the real platform. Since the platform was designed with English units, their metric equivalents have many decimals.

- `chassis_length`: 0.24765 meters
- `wheel_radius`: 0.060325 meters
- `chassis_length`: 0.24765 meters
- `chassis_width`: 0.23495 meters
- `chassis_height` 0.060325 meters
- `wheel_radius` 0.060325 meters
- `wheel_width` 0.060325 meters
- `basis_wheel_x_distance` 0.0936625 meters
- `basis_wheel_y_distance` $(chassis\_width/2) + (wheel\_width/2)$
- `basis_wheel_z_distance` -0.01349375 meters
- `chassis_mass`: 1 kilogram
- `wheel_mass`: 0.1 kilograms

Materials: Two materials were defined for visual purposes:

- **Gray:** Used for the chassis.
- **Black:** Used for the wheels.

Definition of links and joints: The chassis was defined as a box-shaped link, while each of the four wheels was defined as cylindrical-shaped link. The wheels are connected to the chassis via continuous joints, allowing for free rotation. Listing 1 displays an example of how links and joints were defined. It presents the definition of the left front wheel link and the joint that connects it to the chassis; the previously shown constants are used to define the geometry while inertial properties are calculated using the inertial macros.

Listing 1: Example of the definition of a Wheel

```
<link name="left_front_wheel">
    <visual>
        <geometry>
            <cylinder radius="${wheel_radius}" length="${wheel_width
               }"/>
        </geometry>
        <material name="black"/>
    </visual>
    <collision>
        <geometry>
```

```
            <cylinder radius="${wheel_radius}" length="${wheel_width
                }"/>
        </geometry>
    </collision>
    <xacro:inertial_cylinder mass="${wheel_mass}" length="${
        wheel_width}"
                             radius="${wheel_radius}">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:inertial_cylinder>
</link>
<joint name="left_front_wheel_joint" type="continuous">
    <parent link="chassis"/>
    <child link="left_front_wheel"/>
    <origin xyz="${basis_wheel_x_distance} ${basis_wheel_y_distance}
                 ${basis_wheel_z_distance}" rpy="-${pi/2} 0 0"/>
    <axis xyz="0 0 1"/>
</joint>
```

10.4 LiDAR Configuration in lidar.xacro

These are the constants used to define the physical properties of the LiDAR:

- **Radius** (`lidar_radius`): 0.03859 m
- **Height** (`lidar_height`): 0.03500 m
- **Mass** (`lidar_mass`): 0.04600 kg
- **Distance to Optical Center** (`lidar_upper_distance_to_optical_distance`): 0.00780 m

The lidar.xacro file defines the LiDAR sensor and its integration with the robot. This includes the physical properties of its link, the connection to the chassis through the joint and sensor configuration for simulation. The LiDAR model was configured to replicate the sensor acquired by the laboratory. It was modeled as a cylindrical link colored green, this was done in a similar way in which links and joints were defined for the core. The LiDAR was attached to the chassis using a fixed joint above the chassis at:

$$z = \frac{\texttt{chassis\_height}}{2} + \frac{\texttt{lidar\_height}}{2}$$

The sensor is implemented as a gpu_lidar in Gazebo, with the following settings (technical details of the sensor were taken from the guide of the manufacturer):

- **Pose:** Offset slightly below the top of the LiDAR:

$$z = \frac{\texttt{lidar\_height}}{2} - \texttt{lidar\_upper\_distance\_to\_optical\_distance}$$

- **Update Rate:** 10 Hz
- **Topic:** Publishes data to the `scan` topic.
- **Horizontal Field of View:** 360°, with:

- **Minimum Angle:** $-3.14\,\text{rad}$
- **Maximum Angle:** $3.14\,\text{rad}$

- **Range:**
  - **Minimum:** $0.02\,\text{m}$
  - **Maximum:** $12\,\text{m}$
  - **Resolution:** $0.02\,\text{m}$

- **Noise:**
  - **Type:** Gaussian
  - **Mean:** 0.0
  - **Standard Deviation:** 0.01

10.5 Inertial Macros

In `xacro`, inertial macros are utilized to calculate and define the physical properties of robot components accurately. These macros simplify the computation of the inertial matrix for common geometric shapes. In this project, two macros were used, inertial_box and inertial_cylinder.

`inertial_box` Macro

- **Parameters:**
  - `mass`: Mass of the box.
  - `x`, `y`, `z`: Dimensions of the box along each axis.
  - `*origin`: Optional parameter for specifying the origin of the inertial frame.

- **Inertia Tensor:** The inertia matrix elements are computed using:

$$I_{xx} = \frac{1}{12} \cdot m \cdot (y^2 + z^2), \quad I_{yy} = \frac{1}{12} \cdot m \cdot (x^2 + z^2), \quad I_{zz} = \frac{1}{12} \cdot m \cdot (x^2 + y^2)$$

  Off-diagonal terms (I_{xy}, I_{xz}, I_{yz}) are set to zero assuming symmetry.

`inertial_cylinder` Macro

- **Parameters:**
  - `mass`: Mass of the cylinder.
  - `length`: Height of the cylinder along its axis.
  - `radius`: Radius of the circular base.
  - `*origin`: Optional parameter for specifying the origin of the inertial frame.

- **Inertia Tensor:** The components of the inertia matrix are computed as:

$$I_{xx} = I_{yy} = \frac{1}{12} \cdot m \cdot (3r^2 + l^2), \quad I_{zz} = \frac{1}{2} \cdot m \cdot r^2$$

  where r is the radius and l is the length of the cylinder. Off-diagonal terms (I_{xy}, I_{xz}, I_{yz}) are zero.

10.6 ros2_control.xacro

The ros2_control.xacro file defines the robot's control interfaces for use with ros2 control, enabling communication between the robot's hardware and controllers. This configuration uses the IgnitionSystem plugin, which integrates ros2 control with Ignition Gazebo.

Joint Configuration

The file defines control interfaces for four wheel joints:

- `left_front_wheel_joint`
- `left_rear_wheel_joint`
- `right_front_wheel_joint`
- `right_rear_wheel_joint`

Each joint specifies the following:

- **Command Interfaces:**
  - `velocity`: Allows setting the target velocity of the joint.
  - Parameters:
    - `min`: Minimum velocity command (-80).
    - `max`: Maximum velocity command (80).
- **State Interfaces:**
  - `position`: Provides the joint's current position.
  - `velocity`: Provides the joint's current velocity.

The velocity command interface enables the robot to execute velocity-based movement, while the position and velocity state interfaces provide feedback to ensure proper operation. Maximum and minimum velocities were defined based on the maximum velocity of the motors of the real platform, they are measured in RPMs.

10.7 Controllers configuration

The controllers.yaml file specifies the configuration of the control components in the `ros2_control` framework. These components include both the controller manager and specific controllers such as the `diff_drive_controller`, which is essential for differential-drive robots.

The YAML file begins by defining the parameters for the `controller_manager`, including the update rate, the simulation time setting, and the controllers to load:

- `update_rate: 30`: Sets the frequency (in Hz) at which the controller manager updates the controllers.
- `use_sim_time: true`: Enables simulation time, which synchronizes the controllers with the simulation environment.

Two primary controllers are loaded:

- `diff_cont`: A differential-drive controller, responsible for driving the robot.
- `joint_broad`: A joint state broadcaster that publishes joint states (e.g., position, velocity) for visualization and monitoring.

The `diff_drive_controller` is configured under the `diff_cont` namespace with the following key parameters:

- `publish_rate: 50.0`: Sets the frequency (in Hz) at which the controller publishes its data.
- `base_frame_id: chassis`: Defines the reference frame of the robot's base.
- `left_wheel_names` and `right_wheel_names`: Specify the joint names for the left and right wheels, respectively. In this example:
  - Left wheels: `left_front_wheel_joint`, `left_rear_wheel_joint`.
  - Right wheels: `right_front_wheel_joint`, `right_rear_wheel_joint`.
- `wheel_separation: 0.3556`: Indicates the distance between the left and right wheels (in meters).
- `wheel_radius: 0.060325`: Specifies the radius of each wheel (in meters).
- `wheels_per_side: 2`: Denotes the number of wheels on each side of the robot. In this case it is 2 wheels per side as it is a 4 wheeled platform.
- `use_stamped_vel: false`: Configures whether the velocity command includes a timestamp.

10.7.1 Functionality of the `diff_drive_controller`

The diff_drive_controller is a plugin within the ros2_control framework that implements differential drive kinematics for mobile robots. This controller calculates the necessary wheel velocities to achieve the desired linear and angular velocity commands for the robot. Differential drive robots move based on the relative motion of their wheels:

- Linear velocity (v): Achieved by both wheels moving at the same speed.
- Angular velocity (ω): Achieved by wheels moving at different speeds.

The controller translates the robot's desired velocity (v, ω) into individual wheel velocities using the following equations:

$$v_{\text{left}} = v - \frac{\omega \cdot d}{2}$$

$$v_{\text{right}} = v + \frac{\omega \cdot d}{2}$$

where:

- v_{left} and v_{right}: Velocities of the left and right wheels, respectively.
- d: Wheel separation distance.
- v and ω: Linear and angular velocities of the robot.

11 Training the DDQN model

This section presents the process followed for training the agent.

11.1 The Neural Network

The neural network used for navigation in this study is a custom-built Double Deep Q Network (DDQN) designed to process both goal-related information and LiDAR sensor data, generating Q-values as outputs for action selection. Figure 14 shows information about the number of parameters in the layers of the network. Figure 15 shows the architecture of the network, while blue blocks show the layers of the network, gray ones show information about the gradient computation.

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
Q_network                                --
├─Sequential: 1-1                        --
│    └─Linear: 2-1                       256
│    └─ReLU: 2-2                         --
│    └─Linear: 2-3                       8,320
│    └─ReLU: 2-4                         --
├─Conv1d: 1-2                            544
├─Conv1d: 1-3                            18,496
├─Conv1d: 1-4                            32,896
├─Sequential: 1-5                        --
│    └─Linear: 2-5                       2,654,464
│    └─ReLU: 2-6                         --
│    └─Linear: 2-7                       32,896
│    └─ReLU: 2-8                         --
│    └─Linear: 2-9                       8,256
│    └─ReLU: 2-10                        --
│    └─Linear: 2-11                      325
==========================================================================
Total params: 2,756,453
Trainable params: 2,756,453
Non-trainable params: 0
==========================================================================
```

Figure 14: Network information

The network has two main processing streams:

1. Goal Processing Stream: The goal stream takes as input a 3-dimensional tensor, representing the robot's distance to the goal and orientation, as well as "orientation to local goal" which represents the orientation towards the exit of a row or the right row to go in a corridor. This input is passed through two fully connected layers:

- The first layer has 64 neurons and uses ReLU activation.
- The second layer has 128 neurons and also applies ReLU activation.

- These layers create a representation of goal-related features that contribute to decision-making.

2. LiDAR Processing Stream: LiDAR data, representing the surroundings, is processed through a series of convolutional layers:

- The first convolutional layer has 32 filters, a kernel size of 16, a stride of 1, and padding of 1.
- The second layer has 64 filters with a kernel size of 9.
- The third layer has 128 filters with a kernel size of 4.

Each layer applies ReLU activation, producing feature maps that capture spatial patterns from LiDAR inputs. The final output is flattened for concatenation with the goal features.

3. Feature Combination and Q-value Prediction: After processing the goal and LiDAR inputs separately, the network concatenates the two feature representations. The combined feature vector is then passed through several FC layers to predict Q-values:

- The first FC layer has 256 neurons, followed by ReLU activation.
- This is followed by layers with 128, and 64 neurons, each with ReLU activation.
- The final layer outputs Q-values for each possible action, guiding the robot's decision-making.

This architecture effectively integrates both spatial and goal-oriented information, enhancing the network's ability to learn navigation strategies in the greenhouse environment. The Convolutional layers capture spatial information from the LiDAR, while the fully connected layers process goal-related inputs, enabling a comprehensive response for complex navigation tasks.

11.2 The Replay Memmory

The replay memory is a crucial component of the Double Deep Q Network (DDQN) architecture, designed to store and manage past experiences, enabling the agent to learn from them efficiently. In this work, a custom implementation of the replay memory is employed, incorporating both standard memory and a separate memory dedicated to collision events.

Replay Memory Structure and Initialization

The replay memory is implemented as a Python class, which stores transitions as named tuples. A transition is defined as:

- `goal_tensor`: The goal-related observation at the current time step.
- `lidar_tensor`: The LiDAR observation at the current time step.
- `action`: The action taken by the agent at the current time step.
- `next_goal_tensor`: The goal-related observation at the next time step.
- `next_lidar_tensor`: The LiDAR observation at the next time step.

- `reward`: The scalar reward received after executing the action.

The replay memory is initialized with the following parameters:

- `CAPACITY`: The maximum number of transitions the memory can store.
- `device`: The computational device (CPU or GPU) used for storing tensors.
- `collision_reward`: The reward value indicating a collision event.
- `collision_capacity`: An optional parameter to define the size of the collision memory, defaulting to half the capacity of the regular memory if not specified.

The class maintains two separate memories:

- `memory`: The standard memory for storing transitions.
- `collision_memory`: A dedicated memory for storing transitions where the reward corresponds to a collision.

Memorization of Transitions

The `memorize` method is responsible for storing transitions into the appropriate memory. The inputs to this method include:

- `obs`: A dictionary containing the current `goal_obs` and `lidar_obs`.
- `action`: The action taken at the current time step.
- `next_obs`: A dictionary containing the next `goal_obs` and `lidar_obs`, or `None` for terminal states.
- `reward`: The scalar reward received.

The method performs the following steps:

1. Converts observations, actions, and rewards into PyTorch tensors.
2. Creates a transition object containing the current and next observations, action, and reward.
3. Stores the transition in the `collision_memory` if the reward matches the `collision_reward`. Otherwise, it stores the transition in the regular `memory`.
4. Ensures that both memories function as circular buffers, overwriting the oldest transition when full.

Sampling from a prioritized Replay Memory

The prioritized replay memory, first introduced by [Sch+16] presented the concept of sampling more frequently the samples that were the most important to the learning process based on its temporal difference. In this work, a similar approach was taken in which the samples of collision are prioritized in the sampling to ensure an specified number of collision samples is taken for each mini-batch. The `sample` method retrieves a batch of transitions for training. The parameters include:

- `batch_size`: The total number of samples to retrieve.
- `collision_batch_size`: The number of samples specifically drawn from the `collision_memory` (default is 32).

The method splits the sampling process into two parts:

- Samples `collision_batch_size` transitions from the `collision_memory`.
- Samples the remaining transitions from the regular `memory`.

The sampled transitions are combined into a single list for use in training.

Saving and Loading Replay Memory

To enable persistence, the replay memory can be saved to a file using the `save_memory` method. This saves the memory contents, capacities, indices, and device information into a file. The `load_memory` method restores the memory state from a file, enabling the continuation of training from a previous session.

11.3 Agent Class Implementation

The `Agent` class is a critical component in the Double Deep Q-Network (DDQN) implementation. It serves as the interface between the environment and the Q-networks, managing the decision-making process, the memory replay mechanism, and network updates. The following subsections describe each component and functionality of the `Agent` class in detail.

11.3.1 Constructor: Initialization of the Agent

The constructor initializes the `Agent` class with key hyperparameters and configurations necessary for DDQN training.

The parameters of the constructor are described below:

- `lidar_input_size`: The size of the LiDAR input tensor.
- `num_actions`: The number of discrete actions available to the agent.
- `lr`: Learning rate for the optimizer.
- `initial_epsilon`: Initial value for ϵ, the exploration rate.
- `gamma`: The discount factor for future rewards.
- `memory_capacity`: Capacity of the replay memory.
- `batch_size`: The number of samples per minibatch during training.
- `collision_reward`: Reward value associated with collisions.

The constructor performs the following tasks:

- Initializes key hyperparameters such as `gamma`, `batch_size`, and `epsilon`.
- Instantiates the main and target Q-networks using the `Q_network` class, setting both to use either the GPU (if available) or the CPU.

- Configures the loss function as Mean Squared Error (MSE) using `torch.nn.MSELoss`.
- Sets up the optimizer using Adam optimization with the specified learning rate `lr`.
- Configures a learning rate scheduler (`StepLR`) to decay the learning rate every 500 steps.
- Initializes the replay memory using the `replay_memory` class.

11.3.2 Action Selection

The `get_action` method determines the action to take based on the current observation. It implements an ϵ-greedy policy, balancing exploration and exploitation:

- With probability ϵ, the agent selects a random action.
- Otherwise, it uses the main Q-network to predict the Q-values for the given state and selects the action with the highest Q-value.

The method is defined as in Listing 2:

Listing 2: Action Selection Function

```
def get_action(self, obs):
    if np.random.rand() < self.epsilon:
        action = np.random.randint(self.num_actions)
    else:
        goal_tensor = obs['goal_obs'].clone().to(self.device)
        lidar_tensor = obs['lidar_obs'].clone().to(self.device)
        with torch.no_grad():
            self.main_q_network.eval()
            q = self.main_q_network(goal_tensor, lidar_tensor)
            action = int(torch.max(q, dim=1).indices[0])
    return action
```

11.3.3 Replay Mechanism

The `replay` method is responsible for training the main Q-network. It consists of three main steps:

1. `make_minibatch`: Samples a minibatch of transitions from the replay memory and preprocesses them for training.
2. `get_expected_state_action_values`: Computes the expected Q-values for the selected actions.
3. `update_main_q_network`: Updates the main Q-network's weights using the computed loss.

Minibatch Preparation The `make_minibatch` method preprocesses a minibatch of transitions:

- Extracts tensors for goals, LiDAR data, actions, rewards, and next states.
- Handles final states by masking them and ensuring appropriate dimensions.

Expected Q-Value Computation The `get_expected_state_action_values` method calculates the target Q-values using the Bellman equation:

$$Q_{\text{target}} = R + \gamma \max_{a'} Q_{\text{target}}(s', a') \tag{43}$$

where s' represents the next state.

Network Update The `update_main_q_network` method minimizes the loss between the predicted and target Q-values:

$$\text{Loss} = \text{SmoothL1Loss}(Q_{\text{predicted}}, Q_{\text{target}}) \tag{44}$$

11.3.4 Target Network Update

The `update_target_q_network` method synchronizes the weights of the target Q-network with the main Q-network.

11.3.5 Model Persistence

The `save_model` and `load_model` methods allow saving and loading the Q-network's weights, ensuring the agent's progress can be preserved and resumed.

11.4 ROS 2 Node Implementation

11.4.1 The DDQN Navigation Node

The `train_ddqn_navigation` class extends the ROS 2 `Node` base class, implementing the interface between the reinforcement learning agent and the robotic simulation. The agent class, and the replay mechanism already explained in this document are called in this node. Its functionalities include:

Initialization The constructor initializes the following:

- `Agent`: An instance of the `Agent` class is created with user-defined parameters.
- Communication interfaces:
  - A publisher for velocity commands.
  - Subscribers for pose and lidar data.
- A timer to periodically execute the agent's navigation logic.

Lifecycle Management The `create_timer_publisher_and_subscribers` and `destroy_timer_publish` methods manage the lifecycle of communication interfaces, ensuring resources are appropriately allocated and released.

Episode Initialization The `new_episode` method resets the robot's position, initializes the goal, and updates parameters such as ϵ. Additionally, it ensures proper synchronization with the simulation environment by verifying pose and lidar data reception. On each new episode the robot is spawned in a random position in the environment and the goal is also a random point (this random points are set such that always occupy free space).

Pose Resetting The `set_pose` method constructs and executes a service call to reset the robot's pose in the simulation. This operation relies on the Ignition Gazebo service for setting poses, ensuring precise initialization for each episode.

11.4.2 Observation, Action, and Reward Processing

This section details the core methods employed to observe the robot's state, process observations, determine actions, and compute rewards within the reinforcement learning framework.

Observing the Environment The `observe` method collects and processes sensor data, transforming it into a suitable format for the DDQN algorithm. This includes:

- Processing the robot's pose to compute angular and linear distances to the goal.
- Normalizing distances to a range of $[-1, 1]$ for compatibility with the neural network.
- Processing LiDAR readings to create a scaled tensor representing the robot's surroundings.
- Constructing an observation dictionary containing:
  - `lidar_obs`: A tensor of LiDAR measurements.
  - `goal_obs`: A tensor of goal-related observations (e.g., angular and linear distances).

Pose Processing The `process_pose` method calculates essential metrics related to the robot's position and orientation:

1. **Goal Distance:** Computes the Euclidean distance to the goal and normalizes it for network input. Distances are capped at a maximum value to focus the Q-values on the local neighborhood.
2. **Local Goal Distance:** Determines the distance to an intermediate local goal and scales it to the range $[-1, 1]$.
3. **Angular Distance:** Calculates the angular difference between the robot's orientation and the direction to the local goal, normalized to $[-1, 1]$.

LiDAR Processing The `process_lidar_readings` method processes the robot's LiDAR data:

- Converts raw LiDAR readings to a NumPy array.
- Handles infinite values by replacing them with a predefined maximum range.
- Linearly scales the values to the range $[-1, 1]$, enabling direct input into the neural network.
- Computes the minimum range value to detect potential collisions.

Action Execution The `process_action_and_send_it` method translates discrete actions into velocity commands for the robot:

- Implements six discrete actions, including forward movement, rotations, and combinations of linear and angular velocities.
- Publishes velocity commands via a ROS 2 `Twist` message to control the robot in the simulation environment.

Reward Calculation The `get_reward_and_check_if_done` method computes the reward signal based on the robot's state:

- **Collision Handling:** If the minimum LiDAR range is below a threshold, the robot is considered to have collided, and the episode terminates with a negative reward.
- **Goal Achievement:** If the linear distance to the goal is within a tolerance, the robot is considered to have reached the goal, and the episode terminates with a maximum reward.
- **Intermediate Rewards:** For non-terminal states, a small negative reward proportional to the angular distance to the local goal is assigned, encouraging the robot to align with the goal direction.

Timer Callback Implementation The `timer_callback` function serves as the central loop in the reinforcement learning pipeline, orchestrating the observation, action, reward, and learning processes. This subsection outlines its functionality and key components in detail. The routine for training episodes with the timer call_back its shown in Figure 16, it consists of:

1. Verify initialization and ensure all systems are ready before proceeding with the first step.
2. Observe the environment, including the robot's pose and LiDAR data, to construct the observation tensors.
3. Determine the action using the DDQN agent's policy.
4. Send the selected action as velocity commands to the robot.
5. Compute the reward and determine if the episode has terminated.
6. Store the transition in the replay memory and update the Q-network.
7. Manage state transitions and track progress by incrementing the step count.

The following pseudocode summarizes its functionality:

```
if step == 0:
    if not verify_initialization():
        return
obs, ang_dist, min_range, lin_dist = observe()
action = agent.get_action(obs)
```

```
process_action_and_send_it(action)
if step > 0:
    reward = get_reward_and_check_if_done(ang_dist, min_range, lin_dist)
    episode_reward += reward
    agent.replay_memory.memorize(last_obs, last_action, obs, reward)
    if terminal_state:
        agent.replay_memory.memorize(obs, action, None, 0)
    agent.replay()
if not is_done:
    last_action = action
    last_obs = obs
    step += 1
else:
    future.set_result('Terminal state')
```

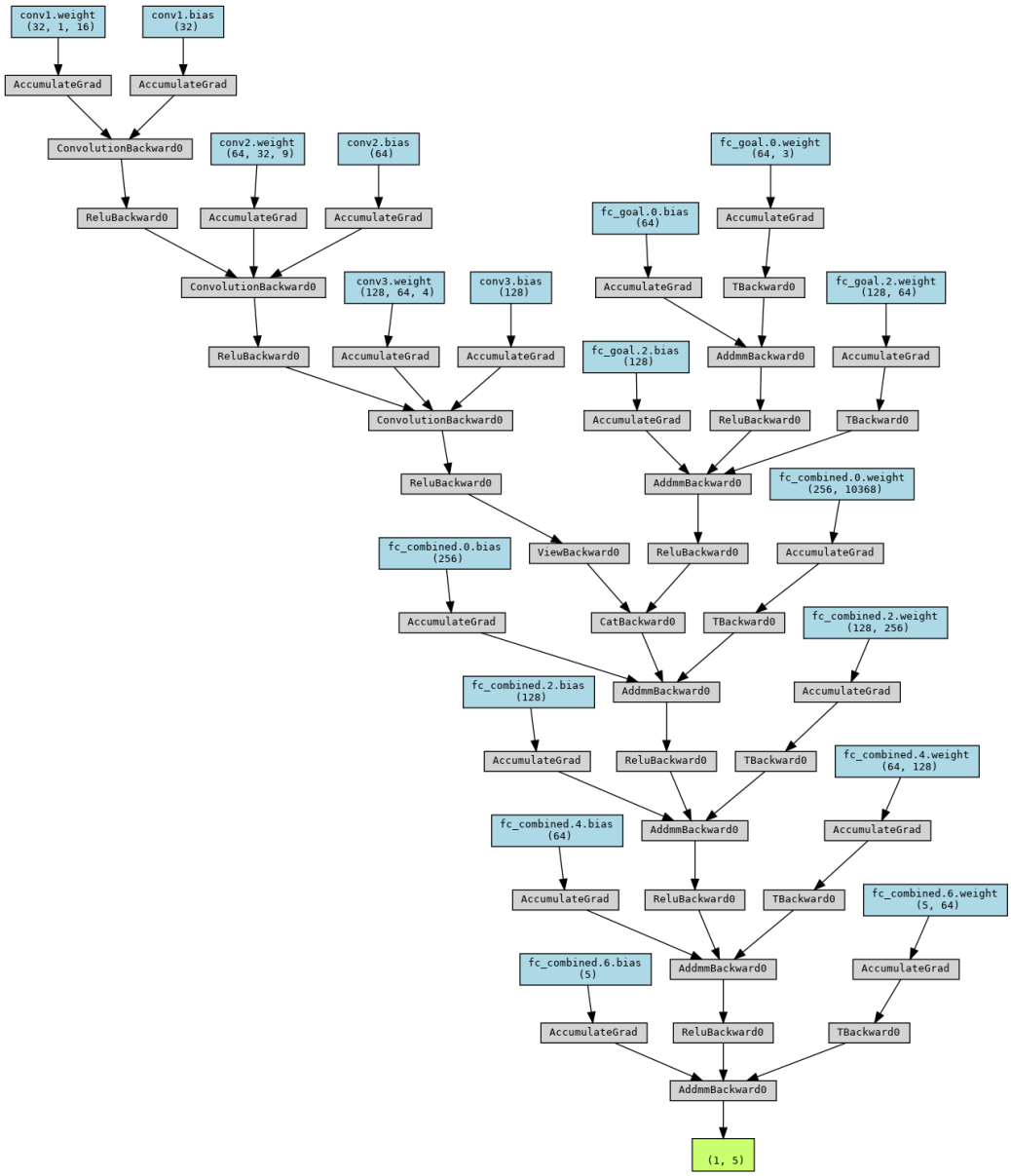


Figure 15: Architecture of the Q - network model

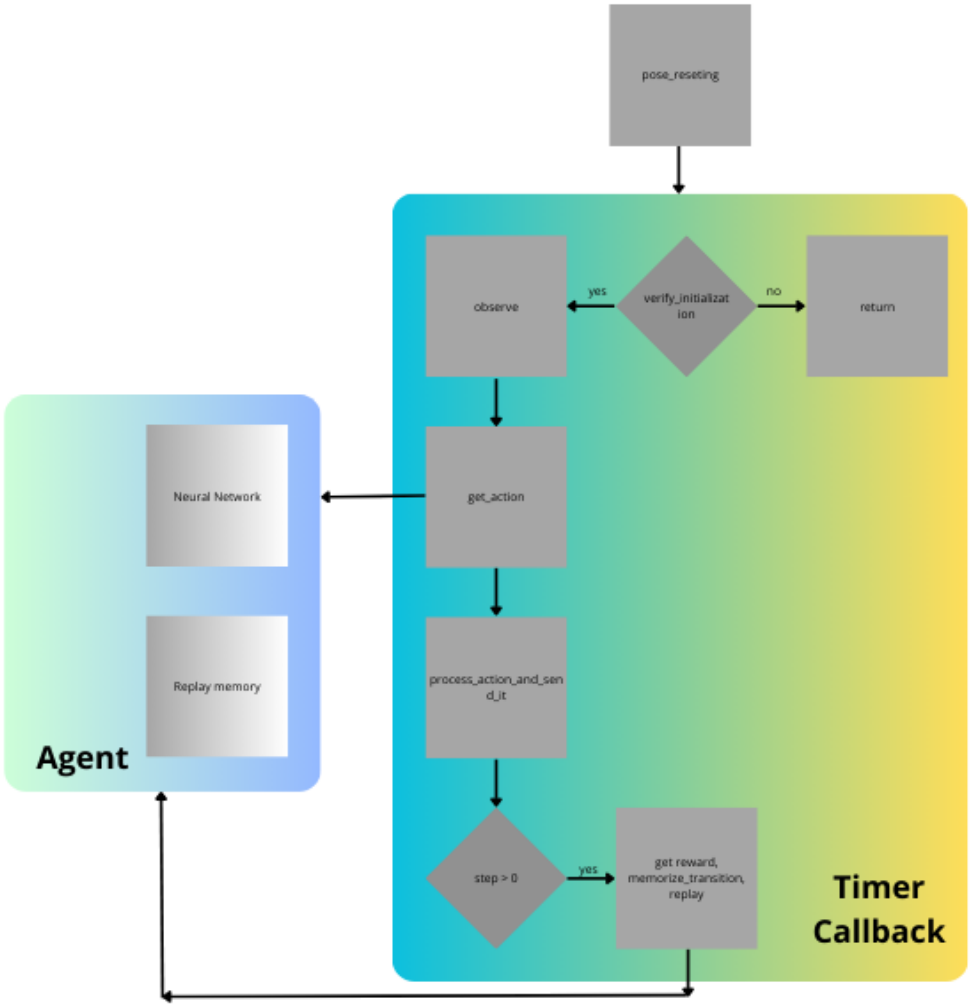


Figure 16: Flow during episodes

12 Evaluation of the DDQN Navigation Model

To evaluate the performance of the DDQN navigation model in a simulated tomato greenhouse environment, the training script `train_ddqn_navigation.py` was adapted to create a new script, `ddqn_navigation.py`. This new script is optimized for evaluating a pre-trained model, focusing on the performance metrics: goal achievement, and collision avoidance.

The primary modifications involved removing training-specific components, such as replay memory updates and target network updates, and integrating mechanisms to evaluate the model's behavior over multiple episodes. This section outlines the critical changes made to the original script to enable evaluation.

12.1 Key Modifications

12.1.1 Agent Initialization

In the original script, the agent initializes both a *main* and a *target* Q-network for Double Deep Q-Learning (DDQN). The evaluation script only requires the main Q-network for inference. Consequently, the target network and its associated updates were removed.

12.1.2 Model Loading

Instead of training the network from scratch, the evaluation script loads a pre-trained model. The function `load_model()` was added to the `Agent` class to facilitate this.

12.1.3 Action Selection

In the training script, actions were selected either through exploration (random action) or exploitation (based on the Q-network output), with an epsilon-greedy policy. For evaluation, the action selection is purely exploitative, as shown in Listing 3.

Listing 3: Action Selection for Evaluation

```
def get_action(self, obs):
    goal_tensor = obs['goal_obs'].clone().to(self.device)
    lidar_tensor = obs['lidar_obs'].clone().to(self.device)
    with torch.no_grad():
        self.main_q_network.eval()
        q = self.main_q_network(goal_tensor, lidar_tensor)
        action = int(torch.max(q, dim=1).indices[0])
    return action
```

12.1.4 Trajectory Logging

To analyze the navigation behavior, the robot's trajectory is logged during each episode. The function `save_trajectory_to_file()` was implemented to save the robot's position at each timestep.

12.1.5 Performance Metrics

The evaluation script logs the number of episodes that ended with successful goal achievement and those that resulted in collisions. These metrics provide insight into the robustness of the trained model.

12.2 Evaluation Results

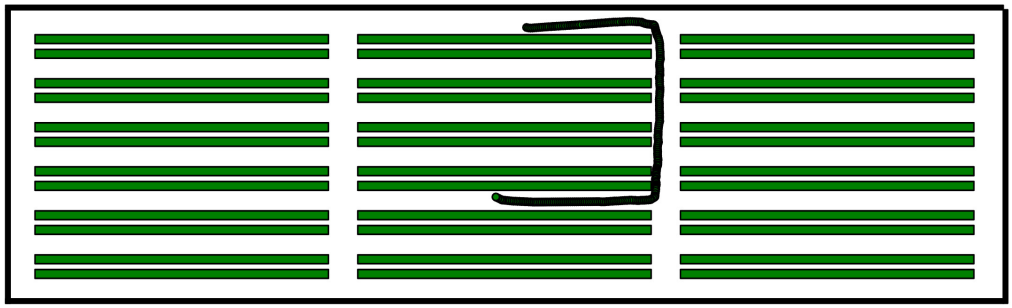
The performance of the DDQN-based navigation model was evaluated in a simulated tomato greenhouse environment. The evaluation consisted of 30 episodes, each starting with the robot placed at a random position and tasked with navigating to a specified goal while avoiding obstacles. The evaluation metrics included the number of episodes that successfully ended at the goal and the number of episodes that terminated due to collisions with obstacles. Table 1 summarizes the results of the evaluation.

Table 1: Evaluation Results of the DDQN Navigation Model

Metric	**Before Prioritized Replay Memory**	**After**
Episodes Ended at Goal	19	27
Episodes Ended with Collision	11	3

The results demonstrate a significant improvement in navigation performance after implementing the prioritized replay memory. Specifically, the number of successful episodes, which the robot reached the goal, increased from 19 to 27. At the same time, the number of episodes that ended with a collision decreased from 11 to 3. This final result is comparable with the one obtained by [Sur+20] of 95% success rate, [Sur+20] also implemented 2D lidar navigation but instead of using a DDQN, an Asynchronous Advantage Actor-Critic network was used.

Figure 17: Example of a trajectory that successfully ended at the goal

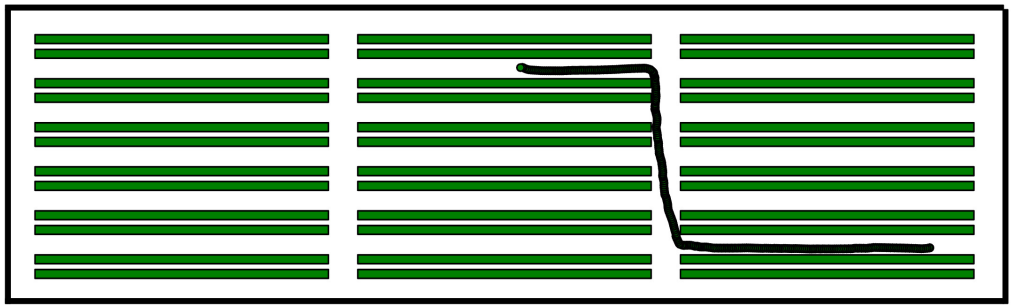


12.3 A qualitative analysis of the results

During the evaluation, only the neural network was used to select the velocity commands for the robot model. In real life performance, it would be more adequate to combine the neural network with a security system programmed to stop the robot when it perceives that there will be a collision; or a more refined version that takes the robot as far as possible from obstacles and then passes the lead back to the neural network. This was not implemented in this work, since the intention was only to test the ability of DRL to navigate, and the implementation of this security system will possibly lead to no collisions.

It was observed that the trajectories that ended in collisions were those whose starting point was quite close to the exterior walls of the greenhouse. This might be due to the lack

Figure 18: Example of a trajectory that successfully ended at the goal



of training samples in these scenarios, since from early stages of the training the robot learns to avoid being quite close to the walls, the proportion of training samples that represent this situation becomes smaller and smaller as the training goes on.

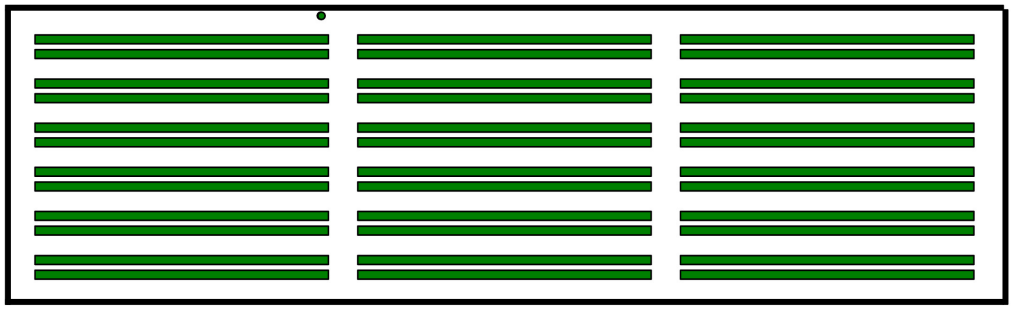


Figure 19: Example of a trajectory that ended with a collision

Figures 17 and 18 show two trajectories that were successfully executed while figures 19 and 20 show a trajectory that ended with a collision right after starting the episode; these figures were generated by storing the coordinates of the robot at each time step and then plotting them on the drawing of the greenhouse.

The trajectories taken by the robot were mostly optimal. When observing the routes in a global way, the proposed system always chooses the corridors that lead to the goal in the shortest way. It was observed that in some trajectories, specially when turning around a corner, the robot passed dangerously close to the obstacles. This behavior is thought to be due to the reward function, which penalizes taking extra time to get to the goal positions, but does not penalize taking risky actions. i.e. in the trajectory shown in figure 18, the robot moves between rows until reaching the vertical corridor, once there moves diagonally towards the the rows where its goal destination its located; when moving diagonally it passes close to the borders of the rows. A possible way to optimize this, would be to experiment with the reward function, adding a negative term proportional to the minimum distance

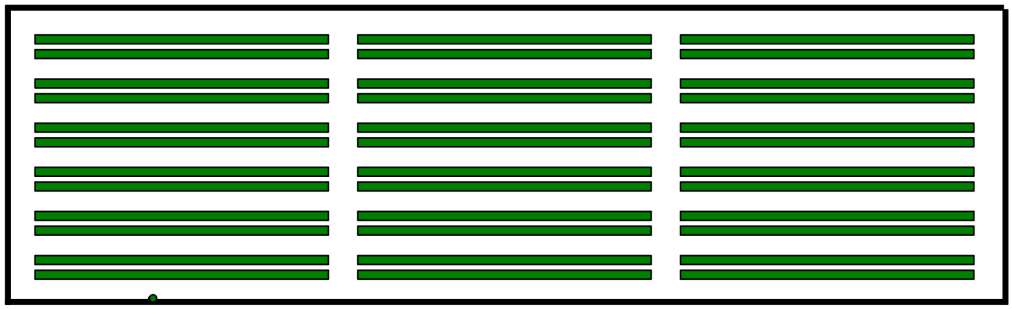


Figure 20: Example of a trajectory that ended with a collision

meassured by the LiDAR.

13 Conclusions

1. **Successful implementation of the DDQN for navigation:** The DDQN successfully trained the robot to navigate toward a goal while avoiding obstacles in the greenhouse environment. With a success rate of 90%, the system demonstrated robust performance in path planning and collision avoidance, with the reward structure encouraging efficient navigation.

2. **Kinematic modeling of the 4WD1 platform:** The virtual model of the 4WD1 four-wheeled platform was successfully developed, ensuring that its kinematics were accurately represented in the simulated environment.

3. **Simulation results and generalization:** The robot's ability to generalize to various spawn points and obstacle configurations was demonstrated through repeated simulations during the testing episodes. The results showed that the navigation system could effectively transfer learning from one scenario to another, making it adaptable to different conditions. This only failed when the agent was spawned close to the exterior walls.

13.1 Challenges and future work

Although the system showed promising results, the following challenges are proposed to be treated in future work:

- Real-world testing will be crucial to assess the system's robustness outside the simulated environment.

- Experiment with a term in the reward function to prevent the robot from getting close to obstacles based on the minimum distance measured by the LiDAR.

- Since many tasks performed by the robot would depend on cameras, the spatial information obtained from them could be used along with the LiDAR. This could be a way to overcome the limitation of the 2D LiDAR to detect objects that are not at the height of its sensor.

References

[And+20] OpenAI: Marcin Andrychowicz et al. "Learning dexterous in-hand manipulation". In: *The International Journal of Robotics Research* 39.1 (2020), pp. 3–20.

[AR16] KR Aravind and P Raja. "Design and simulation of crop monitoring robot for green house". In: *2016 International Conference on Robotics: Current Trends and Future Challenges (RCTFC)*. IEEE. 2016, pp. 1–6.

[BF96] Johann Borenstein and Liqiang Feng. "Measurement and correction of systematic odometry errors in mobile robots". In: *IEEE Transactions on robotics and automation* 12.6 (1996), pp. 869–880.

[BK22] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

[Bon08] Josh Bongard. *Probabilistic robotics. sebastian thrun, wolfram burgard, and dieter fox.(2005, mit press.) 647 pages*. 2008.

[Bot+22] Andrea Botta et al. "A review of robots, perception, and tasks in precision agriculture". In: *Applied Mechanics* 3.3 (2022), pp. 830–854.

[Che+19] Feng Chen et al. "Extreme spin squeezing from deep reinforcement learning". In: *Physical Review A* 100.4 (2019), p. 041801.

[DAB13] Nabil M. Drawil, Haitham M. Amar, and Otman A. Basir. "GPS Localization Accuracy Classification: A Context-Based Approach". In: *IEEE Transactions on Intelligent Transportation Systems* 14.1 (2013), pp. 262–273. DOI: 10.1109/TITS.2012.2213815.

[Dos+15] Filipe Neves Dos Santos et al. "Towards a reliable monitoring robot for mountain vineyards". In: *2015 IEEE international conference on autonomous robot systems and competitions*. IEEE. 2015, pp. 37–43.

[ES22] United Nations: Department of Economic and Population Division Social Affairs. *World Population Prospects 2022: Ten Key Messages.* University of Melbourne, 2022.

[Gao+18] Xinyu Gao et al. "Review of Wheeled Mobile Robots' Navigation Problems and Application Prospects in Agriculture". In: *IEEE Access* 6 (2018), pp. 49248–49268. DOI: 10.1109/ACCESS.2018.2868848.

[Gon+09] R González et al. "Navigation techniques for mobile robots in greenhouses". In: *Applied Engineering in Agriculture* 25.2 (2009), pp. 153–165.

[Hee+17] Nicolas Heess et al. "Emergence of locomotion behaviours in rich environments". In: *arXiv preprint arXiv:1707.02286* (2017).

[HKK15] Arthur Huletski, Dmitriy Kartashov, and Kirill Krinkin. "Evaluation of the modern visual SLAM methods". In: *2015 Artificial Intelligence and Natural Language and Information Extraction, Social Media and Web Search FRUCT Conference (AINL-ISMW FRUCT)*. IEEE. 2015, pp. 19–25.

[Kal+18] Dmitry Kalashnikov et al. "Scalable deep reinforcement learning for vision-based robotic manipulation". In: *Conference on Robot Learning*. PMLR. 2018, pp. 651–673.

[Kaz+22] Iman Abaspur Kazerouni et al. “A survey of state-of-the-art on visual SLAM”. In: *Expert Systems with Applications* 205 (2022), p. 117734.

[KP04] Krzysztof Kozłowski and Dariusz Pazderski. “Modeling and control of a 4-wheel skid-steering mobile robot”. In: *International journal of applied mathematics and computer science* 14.4 (2004), pp. 477–496.

[LaV06] SM LaValle. “Planning Algorithms”. In: *Cambridge University Press google schola* 2 (2006), pp. 3671–3678.

[LLS91] Jean-Claude Latombe, Anthony Lazanas, and Shashank Shekhar. “Robot motion planning with uncertainty in control and sensing”. In: *Artificial Intelligence* 52.1 (1991), pp. 1–47.

[Lon+10] D Longo et al. “An autonomous electrical vehicle based on low-cost ultrasound sensors for safer operations inside greenhouses”. In: *Proceedings of International Conference Ragusa, SHWA'2010*. 2010, pp. 437–443.

[LY22] Min-Fan Ricky Lee and Sharfiden Hassen Yusuf. “Mobile robot navigation using deep reinforcement learning”. In: *Processes* 10.12 (2022), p. 2748.

[MAB15] José Ignacio Rodríguez Molano, Yeimy Andrea Montoya Alvarez, and Leonardo Emiro Contreras Bravo. “Colombian agriculture: approaching agriculture 4.0”. In: *Revista de Ingeniería Solidaria* 18.2 (2015).

[Mir+16] Piotr Mirowski et al. “Learning to navigate in complex environments”. In: *arXiv preprint arXiv:1611.03673* (2016).

[Mni+13] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[OMS21] Luiz FP Oliveira, António P Moreira, and Manuel F Silva. “Advances in agriculture robotics: A state-of-the-art review and challenges ahead”. In: *Robotics* 10.2 (2021), p. 52.

[Pen+17] Xue Bin Peng et al. “Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning”. In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), pp. 1–13.

[Pla22] Aske Plaat. *Deep reinforcement learning*. Vol. 10. Springer, 2022.

[PRB11] Adriana Perilla, Luis Felipe Rodríguez, and Lilia Teresa Bermúdez. “Estudio técnico económico del sistema de producción de tomate bajo invernadero en Guateque, Suitenza y Tenza (Boyacá)”. In: *Revista colombiana de ciencias hortícolas* 5.2 (2011), pp. 220–232.

[Rui+16] Alberto Ruiz-Larrea et al. “A UGV approach to measure the ground properties of greenhouses”. In: *Robot 2015: Second Iberian Robotics Conference: Advances in Robotics, Volume 2*. Springer. 2016, pp. 3–13.

[Rus+17] Andrei A Rusu et al. “Sim-to-real robot learning from pixels with progressive nets”. In: *Conference on robot learning*. PMLR. 2017, pp. 262–270.

[Sal+17] Ahmad EL Sallab et al. “Deep reinforcement learning framework for autonomous driving”. In: *arXiv preprint arXiv:1704.02532* (2017).

[Sch+16] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: `1511.05952 [cs.LG]`. URL: `https://arxiv.org/abs/1511.05952`.

[SCK21] Dasom Seo, Byeong-Hyo Cho, and Kyoung-Chul Kim. "Development of monitoring robot system for tomato fruits in hydroponic greenhouses". In: *Agronomy* 11.11 (2021), p. 2211.

[Shi+19] Haobin Shi et al. "End-to-end navigation strategy with deep reinforcement learning for mobile robots". In: *IEEE Transactions on Industrial Informatics* 16.4 (2019), pp. 2393–2402.

[Sri+22] Sandeep Srikonda et al. "Deep Reinforcement Learning for Autonomous Dynamic Skid Steer Vehicle Trajectory Tracking". In: *Robotics* 11.5 (2022), p. 95.

[SSS16] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. "Safe, multi-agent, reinforcement learning for autonomous driving". In: *arXiv preprint arXiv:1610.03295* (2016).

[Sur+20] Hartmut Surmann et al. "Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments". In: *arXiv preprint arXiv:2005.13857* (2020).

[Tan+18] Jie Tan et al. "Sim-to-real: Learning agile locomotion for quadruped robots". In: *arXiv preprint arXiv:1804.10332* (2018).

[TD17] Héctor Iván Tangarife and Andrés Escobar Díaz. "Robotic applications in the automation of agricultural production under greenhouse: A review". In: *2017 IEEE 3rd Colombian Conference on Automatic Control (CCAC)*. IEEE. 2017, pp. 1–6.

[Til+11] David Tilman et al. "Global food demand and the sustainable intensification of agriculture". In: *Proceedings of the national academy of sciences* 108.50 (2011), pp. 20260–20264.

[TX21] Hamid Taheri and Zhao Chun Xia. "SLAM; definition and evolution". In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104032.

[VGS16] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[Wor22] WorldBank. *Agricultura, valor agregado %delPIB Colombia*. Accessed on september 09, 2023. 2022. URL: https://datos.bancomundial.org/indicator/NV.AGR.TOTL.ZS?end=2022&locations=CO&start=1965.

[Yue+24] Xiangdi Yue et al. "LiDAR-based SLAM for robotic mapping: state of the art and new frontiers". In: *Industrial Robot: the international journal of robotics research and application* (2024).

[Zha+18] Kaichena Zhang et al. "Robot Navigation of Environments with Unknown Rough Terrain Using deep Reinforcement Learning". In: *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. 2018, pp. 1–7. DOI: 10.1109/SSRR.2018.8468643.

[Zhu+17] Yuke Zhu et al. "Target-driven visual navigation in indoor scenes using deep reinforcement learning". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3357–3364.

[ZZ21] Kai Zhu and Tao Zhang. "Deep reinforcement learning based mobile robot navigation: A review". In: *Tsinghua Science and Technology* 26.5 (2021), pp. 674–691.