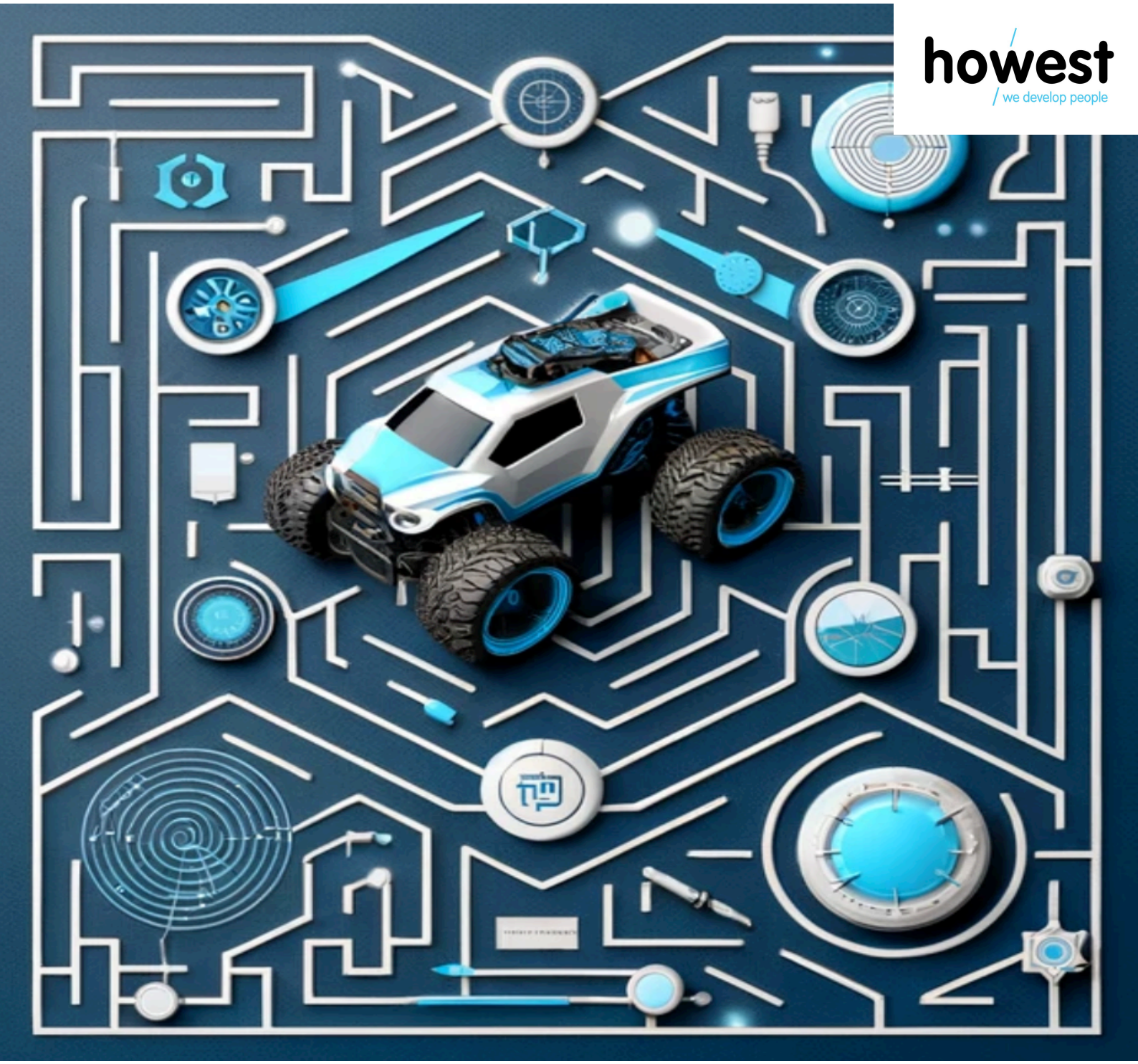
howest
/ we develop people



EXPLORING THE FEASIBILITY OF SIM2REAL TRANSFER IN REINFORCEMENT LEARNING

APPLICATION IN MAZE NAVIGATION

INTERNAL PROMOTOR: GEVAERT WOUTER
EXTERNAL PROMOTOR: SAM DEBEUF

RESEARCH CONDUCTED BY
LUCAS DRIESSENS

FOR OBTAINING A BACHELOR'S DEGREE IN
MULTIMEDIA & CREATIVE TECHNOLOGIES

HOWEST | 2023-2024

Abstract

This research explored the feasibility of transferring a trained reinforcement learning (RL) agent from a simulation to the real world, focusing on maze navigation. The primary objective was to determine if and how an RL agent, specifically a Double Deep Q-Network (DDQN), could successfully navigate a physical maze after being trained in a virtual environment.

First, suitable virtual environments for training an RL agent were explored and the most effective reinforcement learning techniques for this application were evaluated. The study then addressed the challenges in translating simulation-trained behaviors to real-world performance, such as sensor data interpretation and movement replication.

Results showed that the DDQN agent, trained in a simulated maze, could navigate a physical maze with some challenges in sensor data interpretation and, more importantly, movement replication.

This study contributes to AI and robotics by providing insights and methodologies for Sim2Real transfer in RL, with potential applications extending beyond robotics to other fields where simulation-based training is beneficial.

Preface

This thesis, titled "Exploring the Feasibility of Sim2Real Transfer in Reinforcement Learning," is the final project of my studies in Multimedia & Creative Technology at Howest, University of Applied Sciences. The main question it tackles is: "Can a trained RL agent be successfully transferred from a simulation to the real world?" This question highlights my curiosity about how virtual simulations can be used in real-life applications and addresses a significant challenge in AI: making systems that can work in real-world conditions.

My interest in Sim2Real transfer started during the 'Advanced AI' classes and the 'Research Project' module. Learning about reinforcement learning and seeing how simulated environments can mimic complex real-world behaviors got me excited to explore their practical uses. This thesis explores the theory behind Sim2Real transfer and tests its feasibility through various experiments aimed at improving the process and making it more reliable.

The research combines theoretical studies with practical experiments. The theoretical part provides a solid background, while the experiments test how well RL agents perform in different controlled scenarios. By evaluating these agents, the research aims to find the best strategies for successfully transferring them from simulations to real-world applications.

Lucas Driessens
01-06-2024

Acknowledgements

I extend my heartfelt thanks to my supervisor, Gevaert Wouter, for his invaluable guidance and insights. His mentorship has been fundamental to my growth. I also thank Amaury Van Naemen for his technical support with 3D printing, which was crucial for my experiments.

I would also like to thank Sam DeBeuf for not only being my internship supervisor but also my external promotor for this thesis, and for allowing me the opportunity to conduct more research within my internship. Through this internship, I gained valuable soft skills, such as learning to accept feedback, considering different solutions, and being flexible in problem-solving.

Additional appreciation goes to the faculty and staff at Howest, whose commitment to fostering an innovative educational environment has profoundly influenced my development. Their dedication to excellence and support for student initiatives have been instrumental in shaping my academic journey.

Table of Contents

List of Figures

1 List of Abbreviations

1. **AC** - Actor-Critic
2. **AI** - Artificial Intelligence
3. **DDQN** - Double Deep Q-Network
4. **DQN** - Deep Q-Network
5. **ESP32** - Espressif Systems 32-bit Microcontroller
6. **HC-SR04** - Ultrasonic Distance Sensor
7. **MPU6050** - Motion Processing Unit (Gyroscope + Accelerometer)
8. **MSE** - Mean Squared Error
9. **OTA** - Over the Air Updates
10. **PPO** - Proximal Policy Optimization
11. **PWM** - Pulse-Width Modulation
12. **RC** - Remote Controlled
13. **RCMazeEnv** - RC Maze Environment (Custom Virtual Environment for RL Training)
14. **RL** - Reinforcement Learning
15. **SAC** - Soft Actor-Critic
16. **Sim2Real** - Simulation to Reality Transfer
17. **TRPO** - Trust Region Policy Optimization
18. **3D** - Three-Dimensional
19. **2WD** - 2-Wheel Drive
20. **4WD** - 4-Wheel Drive

2 Glossary of Terms

1. **Artificial Intelligence (AI)**: The simulation of human intelligence processes by machines, especially computer systems, enabling them to perform tasks that typically require human intelligence.
2. **Bellman Equation**: A fundamental recursive equation in dynamic programming and reinforcement learning that provides a way to calculate the value of a policy.
3. **Domain Adaptation**: A set of methods used to adapt a model trained in one domain (e.g., simulation) to perform well in a different but related domain (e.g., real-world).
4. **Domain Randomization**: A technique used in reinforcement learning to bridge the gap between simulation and reality by varying the parameters of the simulated environment to improve the robustness of the learned models when applied to real-world tasks.
5. **Double Deep Q-Network (DDQN)**: An enhancement of the Deep Q-Network (DQN) algorithm that addresses the overestimation of action values, thus improving learning stability and performance.
6. **Epsilon Decay**: A technique in reinforcement learning that gradually decreases the rate of exploration over time, allowing the agent to transition from exploring the environment to exploiting known actions for better outcomes.
7. **Experience Replay**: A technique in reinforcement learning where past experiences (state, action, reward, next state) are stored and randomly sampled to break the correlation between consecutive samples, improving the stability and performance of the learning algorithm.
8. **Fixed Q-Targets**: In the context of DQN, this refers to using a separate network to generate target values for training, which are held fixed for a number of steps to improve training stability.
9. **Markov Decision Process (MDP)**: A mathematical framework for modeling decision-making situations where outcomes are partly random and partly under the control of a decision-maker. It is characterized by states, actions, transition probabilities, and rewards.
10. **Mean Squared Error (MSE)**: A loss function used in regression models to measure the average squared difference between the estimated values and the actual value, useful for training models by minimizing error.
11. **Motion Processing Unit (MPU6050)**: A sensor device combining a MEMS (Micro-Electro-Mechanical Systems) gyroscope and a MEMS accelerometer, providing comprehensive motion processing capabilities.
12. **Over the Air Updates (OTA)**: Remotely updates software or firmware, enabling seamless upgrades without physical access.
13. **Policy Gradient Method**: A class of reinforcement learning algorithms that optimize the policy directly by computing gradients of the expected reward with respect to the policy parameters.
14. **Policy Network**: In reinforcement learning, a neural network model that directly maps observed environment states to actions, guiding the agent's decisions based on the current policy.

15. **Proximal Policy Optimization (PPO)**: A policy gradient method for reinforcement learning that simplifies and improves upon the Trust Region Policy Optimization (TRPO) approach.
16. **Pulse-Width Modulation (PWM)**: A method used to control the amount of power delivered to a device by varying the duty cycle of the signal, commonly used in robotics to control motor speed and direction.
17. **Q-agent**: Based on the Q-learning algorithm, it is a model-free algorithm that learns to estimate the values of actions at each state without requiring a model of the environment.
18. **RC Car**: A remote-controlled car used as a practical application platform in reinforcement learning experiments, demonstrating how algorithms can control real-world vehicles.
19. **Raspberry Pi (RPI)**: A small, affordable computer used for various programming projects, including robotics and educational applications.
20. **Reinforcement Learning (RL)**: A subset of machine learning where an agent learns to make decisions by taking actions within an environment to achieve specified goals, guided by a system of rewards and penalties.
21. **Sensor Fusion**: The process of combining sensory data from multiple sources to produce a more accurate and reliable understanding of the environment.
22. **Sim2Real Transfer**: The practice of applying models and strategies developed within a simulated environment to real-world situations, crucial for bridging the gap between theoretical research and practical application.
23. **Target Network**: Utilized in the DDQN framework, a neural network that helps stabilize training by providing consistent targets for the duration of the update interval.
24. **Ultrasonic Distance Sensor (HC-SR04)**: Measures distance using ultrasonic waves, used in robotics for obstacle detection and navigation.
25. **Virtual Environment**: A simulated setting designed for training reinforcement learning agents, offering a controlled, risk-free platform for experimentation and learning.
26. **Wheel Slippage**: Loss of traction causing wheels to spin without moving the vehicle forward, common on uneven terrain.
27. **3D (Three-Dimensional)**: Refers to objects or environments that have three dimensions (length, width, and height), commonly used in computer graphics and simulations.
28. **2WD (2-Wheel Drive)**: A vehicle configuration where power is delivered to two wheels, typically the front or rear wheels, providing propulsion and steering control.
29. **4WD (4-Wheel Drive)**: A vehicle configuration where power is delivered to all four wheels, enhancing traction and stability, particularly in off-road or challenging terrains.

3 Introduction

3.1 Navigating the Maze: Sim-to-Real Transfer in Reinforcement Learning

This thesis explores the intersection of reinforcement learning (RL) and Sim2Real transfer, specifically focusing on the challenges and feasibility of transferring a DDQN-trained agent from a simulated environment to a physical maze. The primary research question is: "Is it possible to transfer a trained RL-agent from a simulation to the real world? (case: maze)"

3.2 Sim-to-Real Transfer: Bridging the Gap

Sim-to-real transfer involves translating learned behaviors from simulated environments to effective actions in the real world. Rusu et al. demonstrated the effectiveness of progressive networks in bridging this gap, particularly in robot learning from pixels, highlighting the importance of adaptable architectures in overcoming environmental discrepancies[6].

3.3 The Maze Navigation Challenge: RC Cars and Algorithms

This study focuses on maze navigation using a remote-controlled (RC) car equipped with sensors. The car learns optimal paths, avoids dead ends, and optimizes its trajectory in a simulated maze. The key question is whether this digital training can translate seamlessly to navigating a physical maze, where real-world challenges like friction and uneven terrain await.

3.4 The Expedition: Three Key Steps

1. **Simulator Design:** Creating a simulation environment that mirrors the real-world maze, allowing the agent to learn navigation strategies in a controlled setting. This involves designing the maze layout, defining sensor inputs, and implementing the reward system.
2. **Transfer Learning Strategies:** Implementing transfer learning techniques to adapt the agent's behavior from simulation to reality. This involves adjusting the agent's policies, reward functions, and exploration strategies to account for the discrepancies between the two environments.

3. **Sensor Calibration:** Calibrating the RC car's sensors, and motors to match their virtual counterparts. This involves exploring how sensors act in the simulated environment versus the real world, adjusting for noise, latency, and environmental factors.

3.5 Beyond Mazes: A Broader Canvas

While this research focuses on maze navigation, its implications extend far beyond. The principles of Sim2Real transfer can be applied to autonomous drones in urban landscapes, self-driving cars avoiding pedestrians, or medical robots operating in cluttered hospital rooms. Sim2Real transfer is the key to making these scenarios feasible.

So, buckle up (or tighten your wheel nuts), as we embark on this thrilling expedition. In the following chapters, I will get into how I arrived at these results. We will start with a literature review and the methodology, followed by the results and challenges encountered. Finally, I will discuss reflections and provide advice for future researchers embarking on a similar journey. Last but not least, at the end of the document there are installation instructions to replicate the setup.

4 Research Questions

This investigation centers around the question: "Is it possible to transfer a trained RL-agent from a simulation to the real world? (case: maze)" To address this question, I explored various aspects of RL training and implementation:

1. **Which virtual environments exist to train a virtual RC-car?**: determine which virtual environments are most effective for RL training.
2. **Which reinforcement learning techniques are best suited for this application?**: Identifying RL techniques suitable for autonomous navigation.
3. **Can the simulation be transferred to the real world? Explore the difference between how the car moves in the simulation and in the real world.**: Assessing how well the agent adapts to real-world dynamics.
4. **Does the simulation have any useful contributions? In terms of training time or performance.**: Evaluating training effectiveness and optimizing performance through simulation.
5. **How can the trained model be transferred to the real RC car? How do we need to adjust the agent and the environment for it to translate to the real world?**: Discussing necessary adjustments for real-world application.

5 Literature Review and Methodology.

5.1 Background on Reinforcement Learning and Reinforcement Learning Algorithms

5.1.1 Background on Reinforcement Learning

The challenge of Sim2Real transfer is pivotal in the deployment of autonomous systems, influencing applications ranging from robotic navigation to self-driving vehicles[15][16]. Recent advancements in RL, such as the introduction of Proximal Policy Optimization[3] and Soft Actor-Critic algorithms[23], have shown promise in various domains. However, the discrepancy between simulated and real environments, often referred to as the 'reality gap'[14], poses a major hurdle.

Several approaches have been proposed to bridge this gap. This involves training models on a variety of simulated environments with different parameters to improve their robustness[4]. Another promising technique is domain adaptation, which seeks to align the simulated and real-world data distributions[5]. Despite these advancements, challenges remain, particularly in ensuring the transferability of learned behaviors in complex, dynamic environments[14].

This thesis builds on these foundations by exploring the feasibility of transferring RL agents trained in a simulated maze environment to a real-world RC car setup. By leveraging the Double Deep Q-Network (DDQN) architecture, known for its reduced overestimation bias[16], this study aims to enhance the reliability of Sim2Real transfer in maze navigation tasks. The chosen approach addresses the limitations of prior methods by integrating robust policy development and comprehensive sensor calibration, providing a novel contribution to the field.

Reinforcement Learning (RL) employs a computational approach where agents learn to optimize their action sequences through trials and errors, engaging with their environment to maximize rewards over time. This learning framework is built upon the foundation of Markov Decision Processes (MDP)[15] , which includes:

- **States** (S): A definitive set of environmental conditions.
- **Actions** (A): A comprehensive set of possible actions for the agent.
- **Transition Probabilities** ($P(s_{t+1}|s_t, a_t)$): The likelihood of moving from state s_t to state s_{t+1} after the agent takes action a_t at time t.
- **Rewards** ($R(s_t, a_t)$): The reward received when transitioning from state s_t to state s_{t+1} due to action a_t.

The principles of Reinforcement Learning, particularly the dynamics of Markov Decision Processes involving states S, actions A, transition probabilities $P(s_{t+1}|s_t, a_t)$, and rewards $R(s_t, a_t)$, form the foundation of how agents learn from and interact with their environment to optimize decision-making over time. This understanding is crucial in the development of autonomous vehicles, improving navigational strategies, decision-making capabilities, and adaptation to real-time environmental changes. The seminal work by R.S. Sutton and A.G. Barto significantly elucidates these principles and complexities of RL algorithms[15].

5.1.2 Background on Double Deep Q-Network (DDQN)

The Double Deep Q-Network (DDQN) is an enhancement of the Deep Q-Network (DQN), a pivotal algorithm in the field of deep reinforcement learning that integrates deep neural networks with Q-learning. DQN itself was a significant advancement as it demonstrated the capability to approximate the Q-value function, which represents the expected reward for taking an action in a given state, using high-capacity neural networks[17][18].

Evolution from DQN to DDQN

DQN Challenges: While DQN substantially improved the stability and performance of Q-learning, it was susceptible to significant overestimations of Q-values due to the noise inherent in the approximation of complex functions by deep neural networks. This overestimation could lead to suboptimal policies and slower convergence during training.

DDQN Solution: Introduced by Hado van Hasselt et al.,[16] DDQN addresses the overestimation problem of DQN by decoupling the action selection from the target Q-value generation—a technique termed "double learning." In traditional DQN, a single neural network is used both to select the best action and to evaluate its value. DDQN modifies this by employing two networks:

- The **current network** determines the action with the highest Q-value for the current state.
- A separate **target network**, which is a delayed copy of the current network, is used to estimate the Q-value of taking that action at the next state.

The Decoupling Effect

This separation ensures that the selection of the best action is less likely to overestimate Q-values, as the estimation is made using a different set of weights, thus reducing bias in the learning process. The target network's parameters are updated less frequently (often after a set number of steps), which further enhances the algorithm's stability.

Impact and Applications

DDQN has been shown to achieve better performance and faster convergence in complex environments compared to DQN. It is particularly effective in scenarios where precise action evaluation is crucial, such as in video games and robotic navigation tasks. The improved reliability and accuracy of DDQN make it a valuable model for studying reinforcement learning in controlled environments where stability and efficiency are critical.

5.1.3 Background on Deep Q-Network (DQN)

The Deep Q-Network (DQN) algorithm represents a significant breakthrough in reinforcement learning by combining traditional Q-learning with deep neural networks. This approach was popularized by researchers with their notable success in training agents that could perform at human levels across various Atari games[27].

Core Mechanism: DQN uses a deep neural network to approximate the Q-value function, which is the expected reward obtainable after taking an action in a given state and following a certain policy thereafter. The neural network inputs the state of the environment and outputs Q-values for each possible action, guiding the agent's decisions.

Innovations Introduced:

- **Experience Replay**: DQN utilizes a technique called experience replay, where experiences collected during training are stored in a replay buffer. This allows the network to learn from past experiences, reducing the correlations between sequential observations and smoothing over changes in the data distribution.
- **Fixed Q-Targets**: To further stabilize training, DQN employs a separate target network, whose weights are fixed for a number of steps and only periodically updated with the weights from the training network[16]

DQN Advantages and Applications

DQN's ability to handle high-dimensional sensory inputs directly with minimal domain knowledge makes it highly versatile and effective in complex environments such as video games, where it can learn directly from pixels.

5.1.4 Background on Q-agent (Q-learning)

Q-agent, based on the Q-learning algorithm, is one of the most fundamental types of reinforcement learning methods. It is a model-free algorithm that learns to estimate the values of actions at each state without requiring a model of the environment[21].

Simplicity and Versatility: Q-learning works by updating an action-value lookup table called the Q-table, which stores Q-values for each state-action pair. These values are updated using the Bellman equation during each step of training based on the reward received and the maximum predicted reward for the next state.

Challenges: While simple and effective for smaller state spaces, Q-learning's reliance on a Q-table becomes impractical in environments with large or continuous state spaces, where the table size would become infeasibly large.

Q-learning Applications

Q-learning has been foundational in teaching agents in environments with discrete, limited state spaces, such as simple mazes or decision-making scenarios with clear, defined states and actions.

5.1.5 Background on Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method for reinforcement learning that simplifies and improves upon the Trust Region Policy Optimization (TRPO) approach. PPO has become popular due to its effectiveness and ease of use[3].

Optimization Technique: PPO seeks to take the largest possible improvement step on a policy while avoiding

too large updates that might lead to performance collapse. It achieves this through an objective function that includes a clipped term, penalizing changes to the policy that move it too far from the previous policy.

Advantages: PPO is robust to a variety of hyperparameters and can be used in both continuous and discrete action spaces. It has shown great success in environments ranging from simulated robotics to complex game environments.

PPO Applications

PPO is favored in many modern RL applications due to its balance between efficiency, ease of implementation, and strong empirical performance.

5.1.6 Background on Actor-Critic (AC)

Actor-Critic methods form a broad class of algorithms in reinforcement learning that combine both policy-based (actor) and value-based (critic) approaches[23].

Dual Components:

- **Actor**: Responsible for selecting actions based on a policy.
- **Critic**: Estimates the value function (or Q-value), which is used to evaluate how good the action taken by the actor is.

Advantages: By separating the action selection and evaluation, actor-critic methods can be more efficient than conventional policy-gradient methods. They reduce the variance of the updates and typically converge faster.

Actor-Critic Applications

Actor-Critic algorithms are versatile and can be applied to both discrete and continuous action spaces. They have been effectively used in applications that require balancing exploration of the environment with the exploitation of known rewards, such as in robotics and complex game environments.

5.2 Methodology

This section explores the Reinforcement Learning Maze Navigation (RCMazeEnv) method using a Double Deep Q-Network (DDQNAgent). It covers the maze environment setup, DDQN agent design, and training algorithm, providing a comprehensive overview of the experimental framework.

5.2.1 Environment Setup (RCMazeEnv)

RCMazeEnv is a custom 12x12 cell maze environment built on OpenAI Gym, designed to test and train agents in navigating complex mazes. The environment is represented by a grid, with '1' indicating walls and '0' indicating paths. The agent starts at position (1,1) and aims to reach the goal at position (10,10), utilizing sensor inputs for navigation.

Maze Structure

The maze structure is predefined with specific wall and path placements to create various challenges for the agent. This setup allows for controlled testing of the agent's navigation algorithms. The walls ('1') act as walls that the agent must avoid, while the paths ('0') are the traversable spaces the agent can move through.

Maze Layout (12x12 Grid) with the start marked as s and the goal marked as X:

1	1	1	1	1	1	1	1	1	1	1	1
1	S	0	0	0	0	0	0	0	0	0	1
1	0	1	1	1	1	1	1	0	1	1	1
1	0	1	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	1	1	1	1	0	1
1	0	1	0	1	1	0	0	0	0	0	1
1	0	0	0	1	1	0	1	1	1	1	1
1	0	1	0	0	0	0	1	0	1	1	1
1	0	1	1	0	1	0	0	0	0	0	1
1	0	1	0	0	1	1	1	1	1	0	1
1	0	0	0	1	1	0	0	0	0	X	1
1	1	1	1	1	1	1	1	1	1	1	1

Agent and Sensors

The agent in RCMazeEnv is equipped with three sensors that provide readings in the front, left and right directions. These sensors are critical for the agent's navigation, offering real-time data about its proximity to the nearest walls. The sensor readings are used to update the agent's understanding of its immediate environment, enabling it to make informed decisions to avoid collisions and find the optimal path to the goal.

- **Front Sensor**: Measures the distance to the nearest wall directly ahead of the agent.
- **Left Sensor**: Measures the distance to the nearest wall to the agent's left.
- **Right Sensor**: Measures the distance to the nearest wall to the agent's right.

State Space Representation

The state space ($\mathscr{S}$) of the environment is a comprehensive representation that includes:

- **Position**: The current coordinates of the agent within the maze, denoted as (x, y).
- **Orientation**: The direction the agent is facing, which can be one of four possible orientations: north, east, south, or west.
- **Sensor Readings**: The distances to the nearest walls as detected by the front, left, and right sensors, represented as $\{s_{\text{front}}, s_{\text{left}}, s_{\text{right}}\}$.

Reward Function and Termination Conditions

In the context of maze navigation, designing an effective reward function is the bread and butter of any reinforcement learning task. The reward function guides the agent's behavior by providing feedback on its actions, encouraging desirable behaviors, and discouraging undesirable ones. The reward function in RCMazeEnv is designed to promote efficient navigation towards the goal while avoiding collisions and revisiting previously explored positions.

Reward Function Components

1. **Collision or Out of Bounds Penalty** ($R_{\textbf{collision}}$):
   - If the sensor detects a collision or out-of-bounds condition (front, left, or right), a penalty is applied:

$$R_{\text{collision}} = -20 \quad \text{if any sensor reading in \{front, left, right\} is 0}$$

2. **Goal Achievement Bonus** ($R_{\textbf{goal}}$):
   - Reward for reaching the goal:

$$R_{\text{goal}} = \begin{cases} +500 & \text{if goal is reached within 1000 steps} \\ +300 & \text{if goal is reached but exceeds 1000 steps} \end{cases}$$

3. **Proximity Reward** ($R_{\textbf{proximity}}$):
   - Reward based on proximity to the goal, where d_{goal} is the Euclidean distance to the goal:

$$R_{\text{proximity}} = \frac{50}{d_{\text{goal}} + 1}$$

4. **Progress Reward** ($R_{\textbf{progress}}$):
    - Reward or penalty based on movement towards or away from the goal:

$$R_{\text{progress}} = \begin{cases} +50 & \text{if moving closer to the goal} \\ -25 & \text{if moving farther from the goal} \end{cases}$$

5. **Revisit Penalty** ($R_{\textbf{revisit}}$):
    - Penalty for revisiting the same position:

$$R_{\text{revisit}} = -10 \quad \text{if revisiting a previously visited position}$$

6. **Efficiency Penalty** ($R_{\textbf{efficiency}}$):
    - Penalty for each step taken to encourage efficient navigation:

$$R_{\text{efficiency}} = -5 \quad \text{per step}$$

Combined Reward Function

The overall reward function combines these components to compute the total reward for a given state and action:

$$R_{\text{total}} = R_{\text{collision}} + R_{\text{goal}} + R_{\text{proximity}} + R_{\text{progress}} + R_{\text{revisit}} + R_{\text{efficiency}}$$

Termination Conditions

The termination conditions determine when the environment has reached a "done" or "ended" state. These conditions are:

$$\text{terminate}(steps, position) = \begin{cases} \text{true, "Exceeded max steps"} & \text{if } steps > 3000 \\ \text{true, "Goal reached"} & \text{if } position = (10, 10) \end{cases}$$

By combining these elements, I found that the agent could not only successfully navigate the maze but also learn the optimal path to the goal exceptionally quickly. Although some may find this overkill, I believe that an excessive reward function is better than a sparse one, as it provides the agent with more information about its environment and encourages more nuanced behavior.

Sensor Integration and Real-World Application

RCMazeEnv supports both virtual and real-world sensor data:

- **Virtual Sensors**: In the simulated environment, virtual sensors provide synthetic distance measurements to the nearest walls, computed based on the agent's position and orientation within the maze.
- **Real Sensors**: In the physical setup, the agent uses actual ultrasonic sensors to obtain distance measurements. These real-world readings are mapped to the virtual environment to maintain consistency in the agent's behavior and decision-making process.

Web Application Interface

A web application was developed to serve as a control interface for the RC car, allowing real-time monitoring and allowing me to intervene if needed. The web app provides a visual representation of the maze, the agent's position, and sensor readings, enabling users to observe the agent's behavior and performance during training and testing.

Web App:

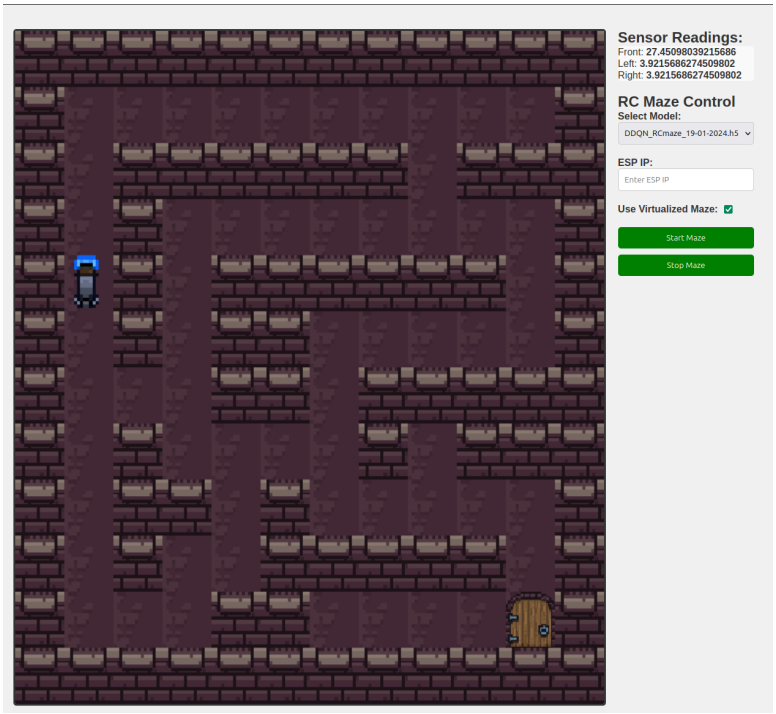


Figure 1: Web App (Image created by author)

5.2.2 Agent Design (DDQNAgent)

The agent uses a Double Deep Q-Network (DDQN) architecture to learn the optimal policy π^*. DDQN is an enhancement over the standard DQN, aiming to reduce overestimation of Q-values by separating action selection from evaluation[16]

- **Policy Network**: Estimates the Q-value $Q(s, a; \theta)$ for taking action a in state s, with weights θ. This network selects actions based on the current policy.
- **Target Network**: Independently parameterized by weights θ^-, it estimates the target Q-value for updating the policy network. The target network mirrors the policy network's architecture but updates less frequently to provide stable target values.

The DDQN update equation modifies the Q-function:

$$Y_t^{DDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a}{\text{argmax}}\, Q(S_{t+1}, a; \theta); \theta^-\right)$$

Where:

- R_{t+1} is the reward received after taking action a in state s.
- γ is the discount factor.
- $\underset{a}{\text{argmax}}\, Q(S_{t+1}, a; \theta)$ selects the action using the policy network.
- $Q\left(S_{t+1}, a; \theta^-\right)$ evaluates the action using the target network.

This approach reduces overestimation by separating the max operation in the target, mitigating overoptimism observed in Q-learning[16].

The action space $\mathscr{A}$ and other agent setup details remain consistent. DDQN significantly improves stability and performance by addressing Q-value overestimation, although its effectiveness varies depending on the task compared to traditional DQN approaches[4].

5.2.3 Training Process

The training process involves using experience replay, where transitions (s, a, r, s') are stored in a replay buffer denoted as D. My objective is to train a Double Deep Q-Network (DDQN) by minimizing the loss function $L(\theta)$. This loss function quantifies the discrepancy between the current Q-values and the target Q-values:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(Y_t^{DDQN} - Q(s, a; \theta)\right)^2\right]$$

Where:

- Y_t^{DDQN} is the target Q-value computed as:

$$Y_t^{DDQN} = \begin{cases} r & \text{if done} \\ r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta); \theta^-) & \text{if not done} \end{cases}$$

- s represents the current state.
- a corresponds to the action taken.
- r denotes the received reward.
- s' signifies the subsequent state.
- θ^- refers to the weights of the target network.
- γ represents the discount factor.

To enhance training stability, we periodically update the target network's weights with those of the policy network. Additionally, I employ an epsilon-greedy strategy for action selection. Initially, I prioritize exploration, gradually reducing exploration as training progresses with a decay rate. This balance between exploration and exploitation contributes to the DDQN's overall performance[15].[16].

5.2.4 Detailed Steps in Code

These steps explain how I implemented and applied the training process for the DDQN agent in the RCMazeEnv environment:

1. **Replay Memory Sampling**:
   - I sampled a minibatch of transitions (s, a, r, s') from the replay buffer D.
2. **Predict Q-values for Current States**:
   - I computed the Q-values for the current states for all actions using the policy network:
   $$Q(s, a; \theta)$$
3. **Predict Q-values for Next States Using Policy Network**:
   - I calculated the Q-values for the next states for all actions using the policy network:
   $$Q(s', a; \theta)$$
4. **Select Best Action for Next State**:
   - I determined the best action for the next state by selecting the action with the highest Q-value:
   $$a^* = \underset{a}{\operatorname{argmax}}\, Q(s', a; \theta)$$

5. **Predict Q-values for Next States Using Target Network**:
    - I predicted the Q-values for the next states using the target network:

$$Q(s', a; \theta^-)$$

6. **Compute Target Values**:
    - I computed the target Q-values using the target network and the selected best action:

$$Y_t^{DDQN} = \begin{cases} r & \text{if done} \\ r + \gamma Q(s', a^*; \theta^-) & \text{if not done} \end{cases}$$

7. **Update Policy Network by Minimizing Loss**:
    - I updated the policy network by minimizing the loss function, which is the mean squared error between the target Q-values and the predicted Q-values:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(Y_i - Q(s_i, a_i; \theta)\right)^2$$

8. **Periodically Update Target Network**:
    - I periodically updated the weights of the target network to match those of the policy network:

$$\theta^- = \theta$$

5.3 Expanding on Real-World Testing

In this study, I conducted experiments indoors to closely replicate the theoretical conditions. The tests were performed on a hard cloth surface to minimize ground-related issues and ensure a consistent testing environment. This step was crucial because during real-world testing, the RC car encountered challenges on uneven surfaces.

However, the exploration wasn't limited to indoor setups alone. I also aimed to assess the adaptability and resilience of my proposed solutions in outdoor environments. Taking the experiments outdoors posed significant challenges due to the differences in ground conditions. Outdoor landscapes are diverse and unpredictable, which exposed limitations in my current method's ability to handle such variations. This highlighted the need for further research and improvements in the methods used, such as the hardware limitations.

The outdoor tests were done in a wooden maze, as shown below:

Figure 2: Real life Maze Build (Image created by author)

6 Addressing Research Questions

6.1 1. Which Virtual Environments Exist to Train a Virtual RC-Car?

Selecting the right virtual environment is crucial for effective RL training of a virtual RC car. Several platforms are available, including Unity 3D, AirSim, CARLA, OpenAI Gym, and ISAAC Gym. For this project, I chose OpenAI Gym due to its flexibility in creating custom environments and compatibility with Python. This choice supports seamless integration with advanced AI coursework and facilitates effective Sim2Real transfer practices[1].

Unity 3D, through its ML-Agents toolkit, offers highly realistic simulations and a user-friendly interface, making it a popular choice for training RL agents in various scenarios, including autonomous vehicle navigation[26]. However, its complexity and the need for substantial computing resources can pose challenges for beginners.

AirSim, developed by Microsoft, provides highly realistic environments for both drones and cars, leveraging the Unreal Engine for superior visual fidelity and physics accuracy. It supports hardware-in-the-loop simulations and offers APIs for integrating with various AI and robotics frameworks[25]. Despite its strengths, the complexity of setup and resource requirements can be a drawback for some users.

CARLA is specifically designed for autonomous driving research and offers a wide range of features for simulating urban driving scenarios. It provides realistic traffic scenarios and supports various sensors, making it a strong choice for traditional vehicle simulations[2]. However, it is less tailored for RC cars, which might limit its applicability in this context.

ISAAC Gym, developed by NVIDIA, focuses on high-fidelity physics simulations, and is optimized for GPU acceleration, making it ideal for robotics simulations. It offers extensive support for reinforcement learning algorithms, though its primary focus on robotics may not align perfectly with the goals of this project[28].

OpenAI Gym's simplicity and reinforcement learning focus make it the ideal fit for this application. Additionally, OpenAI Gym's wide acceptance in the academic community and extensive documentation provide a robust foundation for developing custom environments tailored to specific research needs[1].

6.2 2. Which Reinforcement Learning Techniques Can I Best Use in This Application?

For the autonomous navigation of a virtual RC car in a maze, various reinforcement learning (RL) techniques were considered, including Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Q-Learning, Proximal Policy Optimization (PPO), and Actor-Critic (AC). After careful consideration and testing, DDQN was selected as the most suitable technique for this project.

Deep Q-Network (DQN) was initially considered due to its significant breakthrough in RL, effectively handling high-dimensional sensory inputs and achieving impressive performance in many tasks. However, DQN tends to overestimate Q-values, leading to instability and slower learning. Due to these overestimation issues, DQN was not as stable or reliable as DDQN for this project[20].

Q-Learning, known for its simplicity and effectiveness in discrete and small state spaces, was also evaluated. While it is straightforward to implement and model-free, Q-Learning struggles with large or continuous state spaces, requiring a Q-table that grows exponentially, making it impractical for complex tasks. Given the complexity of maze navigation and high-dimensional sensory inputs, Q-Learning was not feasible for this application[21].

Proximal Policy Optimization (PPO) offers robustness, efficiency, and the ability to handle both continuous and discrete action spaces, maintaining stable updates with its clipped objective function[3]. However, PPO's policy optimization approach sometimes leads to less precise value estimations compared to DDQN, which focuses on accurate Q-value approximations. Although PPO is a powerful technique, the need for precise Q-value approximations in maze navigation made DDQN a better fit.

Actor-Critic (AC) methods combine the strengths of policy-based and value-based methods, reducing variance in updates and generally converging faster. Despite these advantages, AC methods can be complex to implement and may not achieve the same level of stability and performance as DDQN in tasks requiring precise action evaluation. The complexity and less consistent performance of AC methods compared to DDQN led to the decision to not use AC for this project[23].

Double Deep Q-Network (DDQN) addresses the overestimation bias in DQN by decoupling action selection from value estimation, resulting in more accurate Q-value approximations and improved learning stability. It handles high-dimensional sensory inputs effectively and balances exploration and exploitation well. After testing, DDQN proved to outperform other methods in maze-like virtual RC car scenarios, making it the optimal choice for this application[16].

By selecting DDQN, the project leverages its strengths in stability, accuracy, and performance, ensuring effective navigation and learning in complex, sensor-driven environments.

6.3 3. Can the Simulation be Transferred to the Real World? Explore the Difference Between How the Car Moves in the Simulation and in the Real World.

Transferring simulation-trained models to real-world applications involves addressing discrepancies in sensor data interpretation, action synchronization, and physical dynamics [29][4] Real-world sensors may introduce noise and inaccuracies not present in the simulation, and the car's physical dynamics, like friction and wheel slippage, can differ significantly from the idealized simulation [29][4]

To mitigate these issues, implementing sensor data normalization and action synchronization mechanisms is essential to align simulation outcomes with real-world performance [29][4] Introducing failsafe mechanisms and adjusting motor control timings are crucial in reducing collision risks and movement inaccuracies [29][4] Iterative testing and adaptation play a significant role in this process [29][4]

6.4 4. Does the Simulation Have Any Useful Contributions? In Terms of Training Time or Performance.

Efficiency: Simulations enable continuous and automated training sessions without interruptions, which greatly accelerates the development process. This efficiency allows developers to iterate and improve models faster than they could with real-world testing alone.

Safety: By eliminating real-world risks, simulations provide a safe environment for testing autonomous driving models. This safety aspect is crucial, as it allows for extensive testing without the potential for accidents or damage to property.

Computational Advantages: Using powerful computing resources, simulations can create high-fidelity environments that closely mimic real-world conditions. This capability accelerates training by allowing for more rapid and thorough testing of models under various scenarios.

Overall, simulations offer a practical and effective approach to Reinforcement Learning applications. They can significantly reduce training times and enhance performance, making them important tools in Sim2Real[31]

6.5 5. How Can the Trained Model be Transferred to the Real RC Car? How Do We Need to Adjust the Agent and the Environment for It to Translate to the Real World?

Applying a trained model to a physical RC car requires several adjustments. Effective Sim2Real adaptation involves fine-tuning sensor interpretations, implementing action synchronization measures, and adjusting physical dynamics to mirror the simulation. These steps include:

- **Sensor Calibration**: Ensuring the sensors used in the real RC car provide data in a format compatible with the trained model.
- **Motor Control Adjustments**: Adjusting motor control timings to match the physical dynamics of the real car.
- **Failsafe Mechanisms**: Introducing mechanisms to handle unexpected scenarios and reduce collision risks.
- **Incremental Testing**: Conducting iterative tests in real environments to validate and refine the agent's performance.

These adjustments are essential to ensure the successful application of the model in real-world scenarios, facilitating robust and reliable autonomous driving systems[8]. Additionally, implementing sensor fusion techniques can improve the robustness of the real-world model by combining data from multiple sensors to provide more accurate and reliable inputs[12]. This approach helps in mitigating the effects of sensor noise and inaccuracies, further aligning the simulation-trained model with real-world conditions.

7 Model Architecture and Training Insights

To understand how our Double DQN model learns and makes decisions, let's examine its architecture. The model has four dense layers that output three actions tailored to the RC car's movement.

Research has shown that, all things being equal, simpler models are often preferred in reinforcement learning. This is because they can lead to better performance, faster learning, and improved generalization[30] However, finding the right balance of model complexity is crucial. Simplicity is not just about the number of layers or parameters but also about capturing temporal regularities, such as repetitions, in sequential strategies[31]

With these insights in mind, I designed the Double DQN model to strike a balance between simplicity and effectiveness, ensuring optimal performance in maze navigation tasks. By leveraging the strengths of simpler models while addressing critical performance issues, the Double DQN maintains a robust and efficient architecture for reinforcement learning applications.

Model Architecture:

```
# Model: "sequential_52"

# Layer (type) Output Shape Param

=================================================================
dense_200 (Dense) (None, 32) 224
dense_201 (Dense) (None, 64) 2112
dense_202 (Dense) (None, 32) 2080
dense_203 (Dense) (None, 3) 99
=================================================================
Total params: 4515 (17.64 KB)
Trainable params: 4515 (17.64 KB)
Non-trainable params: 0 (0.00 Byte)

---
```

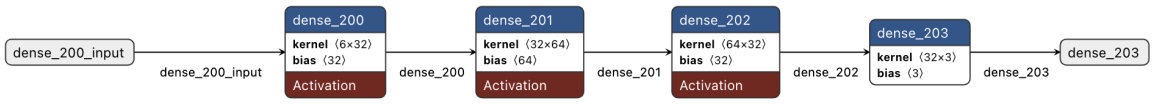


Figure 3: Model Architecture of the Double Deep Q-Network (DDQN) used in the study. (Image created by author)

7.1 Training Parameters

The training of the Double DQN agent was governed by the following parameters:

- **Discount Factor** (`DISCOUNT`): 0.90
  - The discount factor applied to the policy network's Q-values during training.
- **Batch Size**: 128
  - Number of steps (samples) used for training at a time.

- **Update Target Interval** (`UPDATE_TARGET_INTERVAL`): 2
  - Frequency of updating the target network.
- **Epsilon** (`EPSILON`): 0.99
  - Initial exploration rate.
- **Minimum Epsilon** (`MIN_EPSILON`): 0.01
  - Minimum value for exploration rate.
- **Epsilon Decay Rate** (`DECAY`): 0.99973
  - Rate at which exploration probability decreases.
- **Number of Episodes** (`EPISODE_AMOUNT`): 175
  - Total episodes for training the agent.
- **Replay Memory Capacity** (`REPLAY_MEMORY_CAPACITY`): 2,000,000
  - Maximum size of the replay buffer.
- **Learning Rate**: 0.001
  - The rate at which the model learns from new observations.

7.2 Training Procedure

1. **Initialization**: Start with a high exploration rate (`EPSILON`) allowing the agent to explore the environment extensively.
2. **Episodic Training**: For each episode, the agent interacts with the environment, collecting state, action, reward, and next state data.
3. **Replay Buffer**: Store these experiences in a replay memory, which helps in breaking the correlation between sequential experiences.
4. **Batch Learning**: Randomly sample a batch of experiences from the replay buffer to train the network.
5. **Target Network Update**: Every `UPDATE_TARGET_INTERVAL` episodes, update the weights of the target network with those of the policy network.
6. **Epsilon Decay**: Gradually decrease the exploration rate (`EPSILON`) following the decay rate (`DECAY`), shifting the strategy from exploration to exploitation.
7. **Performance Monitoring**: Continuously monitor the agent's performance in terms of rewards and success rate in navigating the maze.

7.3 Evaluation Metrics Overview

7.3.1 Simulation Metrics

Episodic Performance

- **Objective and Goal:** The aim of this metric is to monitor the agent's progress in mastering the maze. By evaluating the learning curve, I see how efficiently the agent can navigate to the end of the maze over successive trials. This gives us insights into its ability to optimize strategies and adapt over time.
- **How it's Assessed:** I measure the number of episodes the agent needs before it can consistently complete the maze. A reduction in episodes over time is a good indicator that the agent is learning and adapting well.
- **Analytical Techniques:** To examine episodic performance, I monitored the output of the script and after training plotted various performance metrics, such as rewards per episode, steps taken, visitation heatmaps, and loss values.

Step count

- **Objective and Goal:** This metric evaluates the agent's decision-making efficiency and ability to optimize its path through the maze. By measuring the steps the agent takes to solve the maze, fewer steps indicate a more efficient and smarter learning process.
- **How it's Assessed:** The step count is recorded at the end of each episode, providing a clear measure of the agent's progress in finding the optimal path. A decreasing step count over episodes indicates improved learning and strategy optimization.
- **Analytical Techniques:** I use quantitative analysis to examine trends in step count. Smoothing techniques may be applied to provide a clearer view of the overarching trends amidst episode-to-episode variability.

MSE Loss Measurement

$$MSE(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2$$

where:

y_i represents the actual value.
$\hat{y}_i$ represents the predicted value.
N is the total number of observations.

- **Objective and Goal:** This metric quantifies the accuracy of the agent's predictions by measuring the squared discrepancies between predicted values and actual outcomes, providing a clear gauge of learning precision.
- **How it's Assessed:** By calculating the mean squared error (MSE) loss during training, I can evaluate how well the agent's predictions align with the actual outcomes, indicating the model's learning progress.
- **Analytical Techniques:** Calculating MSE is straightforward, but understanding its trend requires examining how it correlates with different stages of the agent's learning, such as initial acquisition of knowledge versus later strategy refinement.

Reward Trend Analysis

- **Objective and Goal:** This analysis helps determine how effectively the agent's actions lead to positive outcomes, which are indicative of its learning and strategy development.
- **How it's Assessed:** By tracking and analyzing the rewards the agent accumulates over time, looking for trends that show an increase in reward collection.
- **Analytical Techniques:** Employing time series analysis or plotting cumulative rewards can very easily show us how good the agent is at finding the optimal path through the maze.

Epsilon Decay Tracking

- **Objective and Goal:** This metric monitors how well the agent balances exploration of new paths with exploitation of known successful strategies, key for adapting learning methods effectively.
- **How it's Assessed:** By observing the decline in the epsilon parameter over episodes, which indicates the agent's shift from exploring to exploiting.
- **Analytical Techniques:** The epsilon decay was printed out at the end of each episode, allowing me to quickly adjust hyperparameters if the decay rate was too slow or fast.

7.3.2 Real-World Metrics

Transitioning to real-world application involved assessing how well the strategies developed in simulation held up when the agent faced a physical maze with real obstacles and constraints.

- **Maze Navigation**: Observing the RC car as it maneuvered through a real-world maze served as direct proof of how effectively the training translated from simulation to reality.

- **Sensor Data Analysis**: By examining the real-time sensor data during testing, I could evaluate the agent's ability to interpret and respond to real-world inputs accurately. This analysis helped identify any discrepancies between simulation and reality, guiding necessary adjustments.

8 Experimental Outcomes and Comparative Analysis

8.1 Comparative Analysis of Reinforcement Learning Algorithms

In this analysis, I compare various reinforcement learning algorithms, namely Double Deep Q-Network (DDQN), Deep Q-Network (DQN), Q-agent, Actor-Critic (AC), and Proximal Policy Optimization (PPO). This comparison is based on their performance in navigating a complex maze, focusing on efficiency, learning rate, and adaptability.

8.1.1 1. Visit Heatmaps

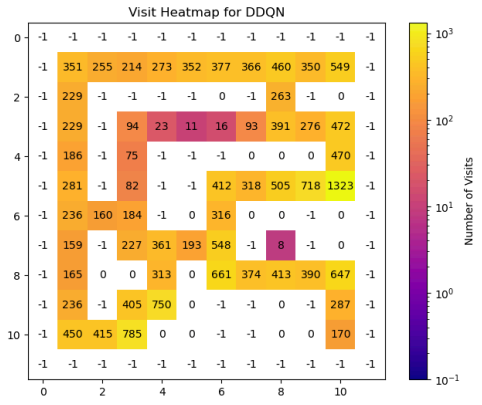


Figure 4: DDQN Heatmap (Image created by author)

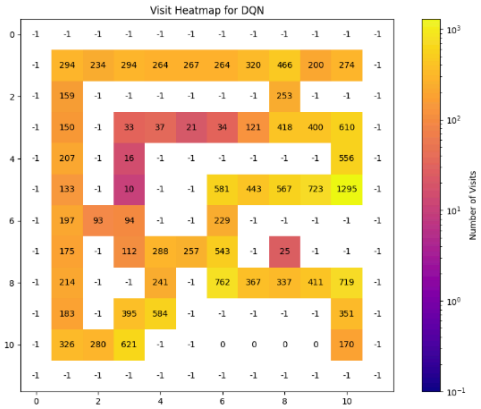


Figure 5: DQN Heatmap (Image created by author)

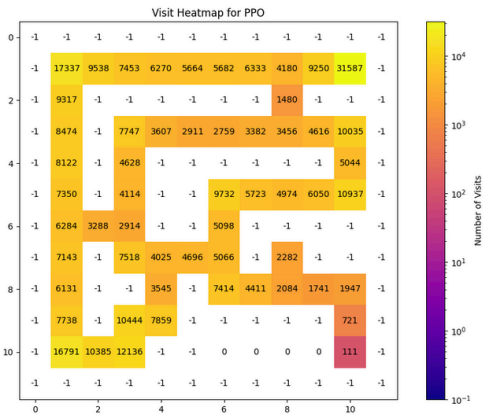


Figure 6: PPO Heatmap (Image created by author)

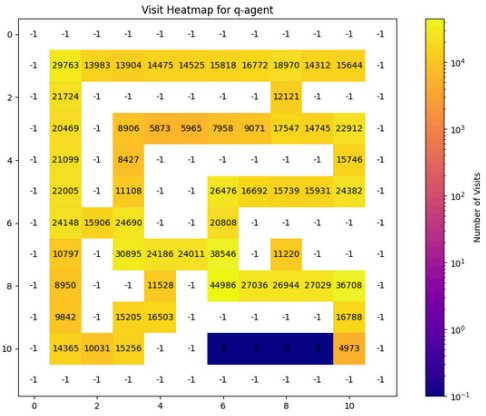


Figure 7: Q-agent Heatmap (Image created by author)

Commentary: The visit heatmaps provide a visual representation of the exploration patterns for different algorithms. The heatmap for Double Deep Q-Network (DDQN) shows a concentrated path, indicating the agent's ability to efficiently learn and focus on the optimal routes through the maze. Similarly, Deep Q-Network (DQN) exhibits focused exploration but with slightly more dispersion compared to DDQN, suggesting a robust learning process. On the other hand, Proximal Policy Optimization (PPO) and Q-agent demonstrate widespread exploration across the maze, indicating less efficient learning and decision-making. These dispersed patterns reflect the algorithms' struggle to consistently identify and follow optimal paths, resulting in suboptimal navigation strategies.

8.1.2 2. Maze Solution Efficiency

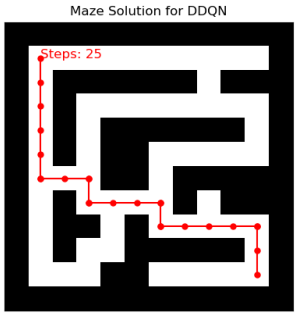


Figure 8: DDQN Maze Path (Image created by author)

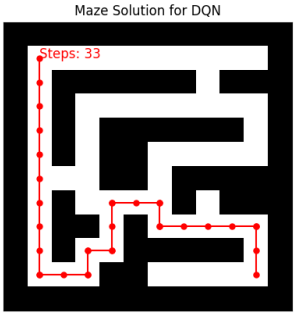


Figure 9: DQN Maze Path (Image created by author)

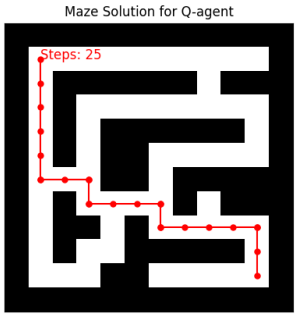


Figure 10: Q-agent Maze Path (Image created by author)

Commentary: The maze solution paths highlight the efficiency of each algorithm in navigating the maze. DDQN and Q-agent both manage to find the shortest path to the goal, though Q-learning takes around 1000 to 2000 steps to start actively using the shortest path, compared to DDQN which finds it under 100 steps. DQN was unable to find the absolute shortest path in my experiments but did manage to find a relatively efficient route. For the PPO and Actor-Critic (AC) algorithms, their paths were more complex and less direct, indicating a need for further optimization and learning to enhance their maze navigation efficiency.

8.1.3 3. Reward History and Distribution

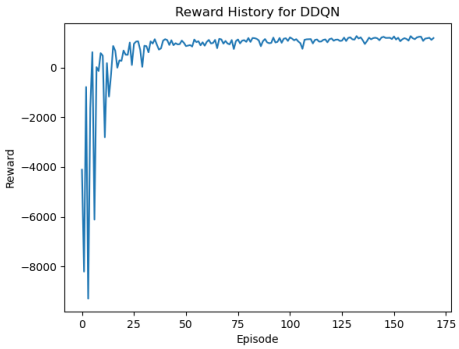


Figure 11: DDQN Reward History (Image created by author)

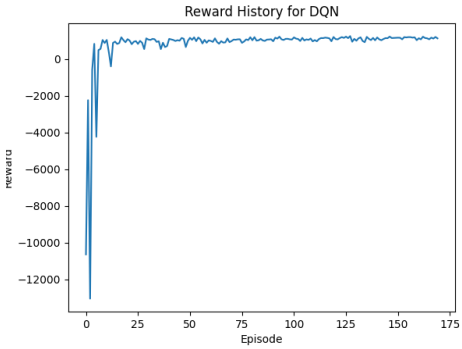


Figure 12: DQN Reward History (Image created by author)

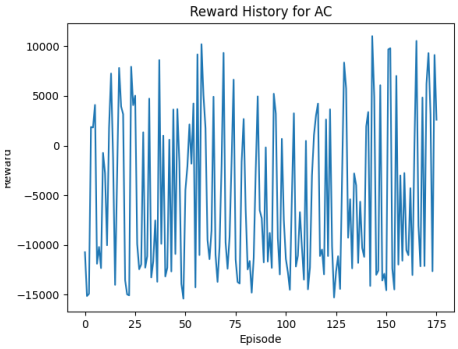


Figure 13: AC Reward History (Image created by author)

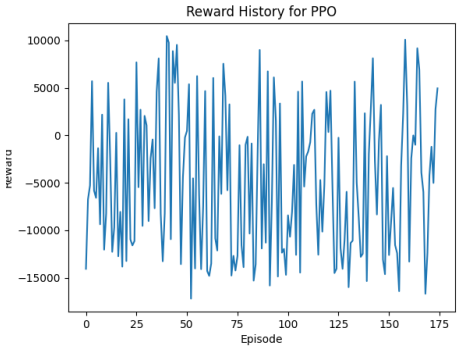


Figure 14: PPO Reward History (Image created by author)

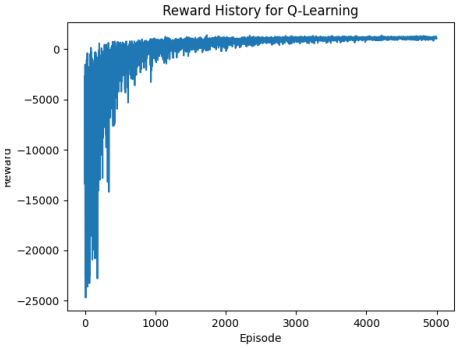


Figure 15: Q-agent Reward History (Image created by author)

Commentary: The MSE graphs track the learning accuracy and error management of DDQN and DQN over time. DDQN achieves the lowest and most stable MSE values, signifying its effective learning and strong capability in minimizing prediction errors. DQN also performs well with relatively stable MSE, though slightly higher than DDQN, indicating competent but less optimal error reduction. In contrast, AC shows higher and more variable MSE values, pointing to less effective learning and greater difficulties in managing prediction errors. These fluctuations suggest AC struggles to consistently reduce errors, impacting its overall performance. The comparison highlights DDQN's superior accuracy and error management, with DQN as a strong contender, while AC requires further optimization to enhance its learning stability and efficiency.

Unlike DDQN and DQN, Proximal Policy Optimization (PPO) and Q-learning do not utilize Mean Squared Error (MSE) for evaluating performance. Instead, PPO focuses on a loss function derived from the clipped surrogate objective, ensuring stable updates by constraining policy changes within a specified threshold. Q-learning relies on the difference between expected and received rewards to update its Q-values. Consequentially, the performance and learning stability of PPO and Q-learning are gauged through their respective loss functions rather than MSE.

8.1.4 4. Mean Squared Error (MSE) Over Time

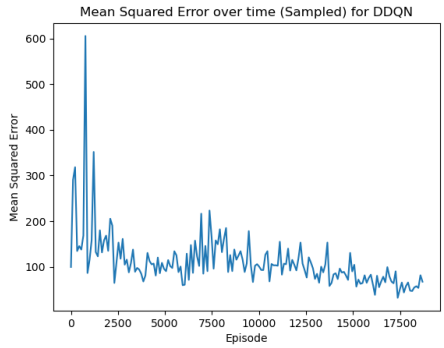


Figure 16: DDQN MSE (Image created by author)

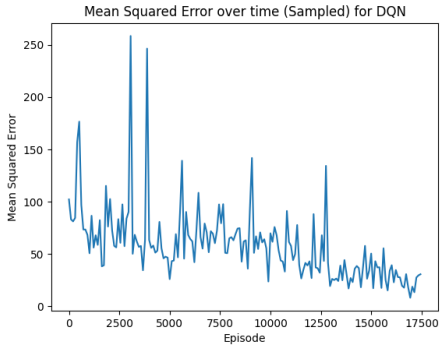


Figure 17: DQN MSE (Image created by author)

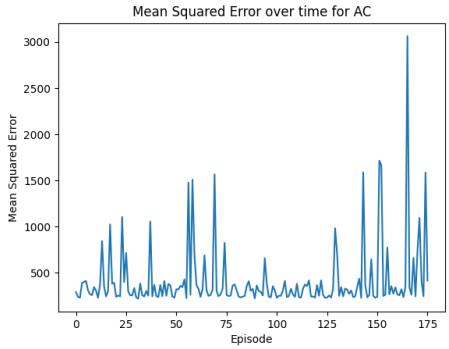


Figure 18: AC MSE (Image created by author)

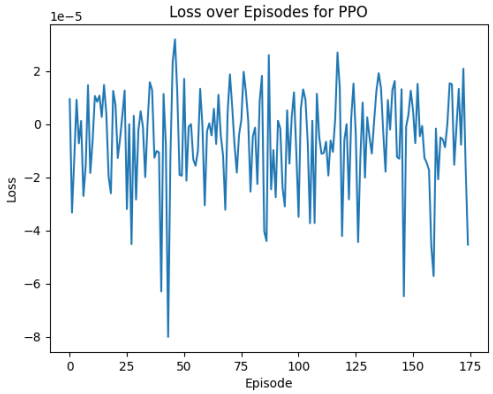


Figure 19: PPO Loss (Image created by author)

Commentary: The MSE graphs track the learning accuracy and error management of each algorithm over time. DDQN achieves the lowest and most stable MSE values, signifying its effective learning and strong capability in minimizing prediction errors. DQN also performs well with relatively stable MSE, though slightly higher than DDQN, indicating competent but less optimal error reduction. In contrast, AC and PPO show higher and more variable loss values, pointing to less effective learning and greater difficulties in managing prediction errors. These fluctuations suggest these algorithms struggle to consistently reduce errors, impacting their overall performance. The comparison highlights DDQN's superior accuracy and error management, with DQN as a strong contender, while AC requires further optimization to enhance its learning stability and efficiency.

8.2 Conclusion and Insights

This comprehensive analysis reveals distinct performance characteristics and efficiencies of various reinforcement learning algorithms in maze navigation. DDQN stands out for its balanced approach, efficiently solving the maze with the fewest steps while demonstrating superior stability and effective error management. DQN, though slightly less efficient in navigation, showcases robust learning stability, making it a reliable choice. Q-agent, despite its simpler approach, competes closely with DDQN in terms of the steps required to solve the maze but struggles during the initial learning phases.

AC and PPO exhibit higher fluctuations in their performance metrics, indicating the need for further optimization to achieve better consistency and efficiency. These algorithms show potential but require more refinement to handle the complexities of maze navigation effectively.

9 Implementation of Real-World Control Algorithms

9.1 Introduction to Real-World Implementation

In this section, I will explain the practical application of control algorithms that were developed through simulations, and are now being adapted to control a physical robot. This transition is pivotal for evaluating how simulated behaviors translate into real-world scenarios, thereby assessing the effectiveness and limitations of Sim2Real transfer.

9.2 System Overview

At the core of my RC car is the ESP32-WROOM-32 module, a small and powerful microcontroller with integrated Wi-Fi and Bluetooth capabilities, powered by a 18650 type battery.

The structure of the car is built on a 2WD miniQ robot chassis with a custom 3D printed top plate, which provides a sturdy base for mounting all components. To control the motors, an L298N dual H-Bridge motor controller was employed, allowing me to drive the motors in both directions and adjust their speed using pulse-width modulation (PWM) signals from the ESP32 microcontroller.

For sensor integration, the system utilizes HC-SR04 ultrasonic sensors for distance measurement and an MPU6050 gyroscope for orientation and stabilization. These sensors are mounted using custom 3D printed parts, specifically designed to hold the HC-SR04 sensors securely in place. Additionally, a OLED screen is incorporated into the setup to provide real-time feedback on the robot's status, such as its IP address and operational states, enhancing user interaction and debugging capabilities.

9.3 Code Architecture and Integration

System Initialization

Understanding the system's initial setup is required for ensuring robust and reliable operation. This involves preparing the robot by configuring its hardware interfaces, establishing network connectivity, and setting up sensors and actuators.

- **WiFi and OTA Configuration**: This sets up a network connection and facilitates Over-The-Air (OTA) updates, crucial for remote debugging and iterative improvements.
- **Sensor and Display Setup**: Activates ultrasonic sensors for distance monitoring and initializes a display to provide real-time feedback on the robot's status and IP address, enhancing user interaction and debugging capabilities.
- **MPU6050 Setup and Calibration**: Calibrates the gyroscopic sensor for accurate angle measurements, essential for precise navigation.
- **Motor Setup**: Configures motor drivers and establishes initial motor states, prepping the robot for subsequent movement commands.

Motor Control Mechanism

Movement functions are implemented to help translate simulated navigation algorithms into the real-world robotic system.

- **Variables for Motor Control**

  ```
  int initialSpeed = 125; // Higher initial speed for robust movement
  int minSpeed = 40;      // Minimum speed to maintain control
  int speed = initialSpeed;
  constexpr int TURNDURATION = 245;
  ```

  These variables dictate the motors' initial and minimum speeds, and the duration for turning, facilitating precise and controlled movements by adjusting the speed dynamically based on the robot's turning angle.

- **Forward Movement**

  The `moveForward` function initiates rapid forward motion, with real-time checks for obstacles to ensure safe stops—mimicking the real-world need for dynamic responsiveness.

- **Left Turn**

  The `moveLeft` function adjusts motor speeds dynamically, a strategy refined in simulations to accommodate physical and inertia effects during turns, ensuring smooth and controlled navigation.

- **Right Turn**

  The `moveRight` function applies similar adjustments and sensor feedback to execute precise right turns. Incorporating `calibrateSensors()` before each movement guarantees accurate gyroscopic data, vital for the precise execution of turns.

- **Stopping Movement**

  The `stopMoving` function is designed to immediately halt all motions, crucial for accident prevention and adaptation to sudden changes in dynamic environments.

Calibration and Sensor Data Interpretation

Calibration is mandatory for ensuring sensor accuracy and reliability and maintaining the accuracy over time. The `calibrateSensors` function periodically recalibrates the gyroscopic sensors to correct any data drift or inaccuracies.

```
void calibrateSensors()
{
    long gyroZAccum = 0;
    Serial.println("Calibrating...");
    for (int i = 0; i < 100; i++)
    {
        int16_t ax, ay, az, gx, gy, gz;
        # Read sensor data
        mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
        # Accumulate gyro Z-axis readings
        gyroZAccum += gz;
        # Delay for sensor stability
        delay(20);
    }
    // Calibration based on ±100 readings
    mpu.setZGyroOffset(-gyroZAccum / 13100);
    Serial.println("Calibration Complete");
}
```

10 Challenges and Limitations

Transitioning from simulated environments to real-world applications introduces unique challenges, particularly in interpreting sensor data and replicating vehicle movements. This section addresses the challenges I faced during the real-world implementation of the RC car and the strategies employed to overcome them.

10.1 Challenges and Solutions in Real-World Implementation

10.1.1 Challenge 1: Addressing Movement Discrepancies in Sim2Real Transfer

Description: Bridging the gap between how the RC car moves in simulations and in the real world.

Solution Attempt: I fine-tuned the action command frequency using an asynchronous method, waited for the motor to finish moving, and considered a queued action system. Precise movement in the real world turned out to be more critical than in simulations.

10.1.2 Challenge 2: Alignment Issues and Motor Encoder Implementation

Description: Ensuring the RC car moves in a straight line, as there was a persistent ~3-degree offset.

Solution Attempts:

- **Attempt 1**: Used motor encoders to improve accuracy but faced precision limits.
- **Attempt 2**: Switched to a more powerful motor, but the added weight brought back the alignment issue.
- **Attempt 3**: Added an MPU6050 gyroscope to measure and adjust orientation, which initially helped with 90-degree turns but didn't fix the offset.
- **Attempt 4**: Removed the Raspberry Pi and used only the ESP32 for all controls, resulting in a lighter, more precise robot, though it still struggled with consistent 90-degree turns.

10.1.3 Challenge 3: Ensuring Consistent and Effective Training

Description: Maximizing training efficiency and performance while keeping things consistent between simulations and real-world scenarios.

Solution: Training in a simulation was much more efficient due to the difficulties of resetting the RC car, dealing with manual interferences, and limited battery life.

10.1.4 Challenge 4: Accurate Sensor Data Normalization for Sim2Real Transfer

Description: Aligning sensor data between simulated and real-world environments for accurate model performance.

Solution: Implemented functions to ensure real-world sensor data matched the training data.

- **Real-World Sensor Data Normalization**:

$$\text{map\_distance}(d) = \begin{cases} d & \text{if } d < 25 \\ 25 + (d - 25) \times 0.5 & \text{otherwise} \end{cases}$$

- **Simulation Sensor Data Normalization**:

$$\text{normalize\_distance}(d) = \max\left(0, \min\left(\frac{d}{\text{sensor\_max\_range}}, 1\right)\right) \times 1000$$

10.1.5 Challenge 5: Integration of Failsafe Mechanisms

Description: Preventing collisions and ensuring safe navigation in the real world.

Solution: Developed a failsafe system to prevent unwanted forward movement and retrained the model with this feature, which solved the issue of the robot driving into walls and getting stuck.

10.1.6 Challenge 6: Training Environment and Technique Efficacy

Description: Finding the most effective environment and RL technique for training.

Solution: The DDQN technique was more efficient than DQN, Q-agent, PPO, and ActorCritic approaches, highlighting the importance of selecting the right technique.

10.2 Limitations

Insufficient Data for Exact Positioning: One significant limitation was that the RC car only had an estimated position in the maze. This estimation was based solely on the car's previous position and the direction it was moving in. There was no absolute positioning system, such as GPS or visual markers, to verify the car's exact location within the maze. As a result, if the car encountered an obstacle, slipped, or deviated slightly from its intended path, it would not be able to correct its position accurately. The agent controlling the car operated under the assumption that the car was still in the correct position, leading to compound errors. These positional inaccuracies could cause the car to make incorrect decisions, such as turning prematurely or missing turns, thereby failing to navigate the maze successfully. This limitation shows the need for more sophisticated localization techniques, such as integrating additional sensors or using advanced algorithms to better estimate the car's position and mitigate errors.

Hardware Limitations: The hardware used in the RC car, such as the ultrasonic sensors and motor encoders, degraded over time. The wheels would start to slip off their axis, causing unreliable movement. When this occurred, the encoder readings became inaccurate, leading to numerous issues.

Environmental Variability: Real-world environments are imperfect. Different floor textures caused the car to move inconsistently, and the walls needed to be perfectly straight for the ultrasonic sensors to work accurately.

Sensor Calibration and Data Normalization: Calibrating sensors like the MPU6050 gyroscope and HC-SR04 ultrasonic sensors required constant adjustments to maintain accuracy. Variability in sensor data between different environments added complexity, making the real-world performance inconsistent.

Mechanical Wear and Tear: Over time, mechanical components, such as motor encoders and wheels, showed signs of wear. The wheels slipping off their axis affected movement precision, and encoder malfunctions led to inaccurate readings. This degradation impacted the overall reliability and performance of the RC car.

Sim2Real Transfer Challenges: The discrepancy between simulated and real-world environments, known as the 'reality gap,' posed significant challenges. The behavior of the RC car in simulations did not always accurately translate to real-world scenarios, requiring extensive adjustments and recalibrations to achieve acceptable performance.

Synchronization of Motor Movements and Sensor Data: Achieving precise synchronization between the agent's inputs and the RC car's motor movements was challenging. The delay between sending a command and the car's response, as well as the lack of immediate feedback from the motors, made it difficult to know if the car had executed the intended action. The agent could and would send commands to the car before it had finished its previous movement, leading to the agent having outdated sensor data.

10.3 Conclusion for Challenges and Limitations

This section outlines the practical challenges encountered while applying reinforcement learning (RL) techniques to autonomous RC cars. My journey began with selecting OpenAI Gym as the virtual environment due to its simplicity and relevance to RL. The Double Deep Q-Network (DDQN) emerged as the most effective RL technique for navigating complex environments.

However, transitioning from simulations to real-world applications revealed significant discrepancies, particularly in movement control and sensor data alignment. I explored various solutions like motor encoders, power adjustments, and gyroscope integration, which partially addressed these issues. Efforts to normalize sensor data and implement failsafe mechanisms also contributed to better alignment with real-world conditions.

A significant advancement was achieved by simplifying the robot's design to use only the ESP32 module instead of a Raspberry PI and the ESP32, making it lighter and more precise. This change marked a considerable step in overcoming the previous challenges.

Although I made substantial progress in addressing these challenges, there is still room for improvement in achieving seamless Sim2Real transfer and ensuring consistent performance across different environments.

11 Discussion and Reflection

Looking back on my research journey, I've learned a ton and grown a lot as a person. Here's a rundown of the key insights and lessons from working on my RC car project, especially around Sim2Real transfer in reinforcement learning.

11.1 Embracing Innovation and Adaptability

One of the biggest takeaways from this project and my internship was the importance of staying open to new ideas and being flexible. This was a big challenge for me as I tend to get a bit stuck in my ways. The challenges I encountered forced me to think outside the box and come up with quick solutions to keep the project moving forward, given the time and resource constraints.

11.2 Bridging Theory and Practice

The transition from theoretical knowledge to practical application proved to be more complex than anticipated. While virtual environments were manageable and controlled, real-world conditions introduced unforeseen variables. This experience underscored the gap between simulation and reality, highlighting the necessity for continuous adjustment and iterative testing. Practical engagements, such as calibrating sensors and adjusting control algorithms, reinforced my ability to balance theoretical insights with practical needs. Feedback from interviews, including the possibility of using a more complex environment than just a 2D array as the representation of the maze, made me realize that I was too quick to believe that the environment I had was good enough.

A Reddit user suggested expanding the action space in the virtual environment to include direct motor control. This approach could have improved the agent's ability to adapt to the environment better and eliminated the need for perfect movement replication. However, this would have required a longer training period and increased the risk of encountering different types of problems, like the HC-SR04's being practically useless in such an environment as it would need to have a more reliable way of determining the car's position to be able to adjust the movements accordingly.

- **Highlighting Specific Problems**
  - **Sensor Calibration and Data Normalization:** Early on, I figured out that sensor readings between the real world and my simulation were bigger than expected. Getting them to match up meant a lot of trial and error with calibration steps and tweaking data normalization to get consistent readings. This could also have been avoided if my simulation had been more advanced in terms of the distance between the RC car and walls being more than just a 2D array, nevertheless, this was a pretty easy problem to solve.
  - **Motor Encoder Issues:** When I realized I couldn't move my RC car as precisely as I wanted, I decided to get motor encoders to try and fix this issue. Note: Yes, I should have bought these from the beginning; I made a huge mistake when figuring out which hardware to use. Although the car did drive better than without encoders, one of my encoders sadly broke just as things started working, and new ones would arrive too late. That's when I decided to use an MPU6050 to try and fix the motor control issues, which also sadly didn't work as intended.

Wouter Gevaert pointed out that while a 4WD setup could potentially provide better traction, it generally would not be better due to over time slippage and increased difficulty in controlling the movement. Additionally, switching from ultrasonic sensors to a camera setup, where the camera would act as the eyes of the agent, would not solve the problem of the agent not knowing the exact position of the car in the maze.

11.3 Anticipatory Thinking and Proactive Problem-Solving

Throughout the project, I constantly had to think ahead and anticipate potential issues. This is one field where I lacked a lot of experience; this is also something I learned during my internship. I tend to get an idea and just go with it, which is not the best approach. I'm lucky that most of the things I got stuck on were solvable, but I see how planning and discussing my plans with others before starting could have saved me a lot of time and effort.

11.4 Feedback and Continuous Improvement

During the evaluation of the practical part of this research, the jury provided positive feedback, recognizing the successful application of reinforcement learning techniques.

- **Practical Implementation:** The jury appreciated the practical implementation of the RC car and the efforts to address real-world challenges. They acknowledged the complexity of transitioning from simulations to real-world scenarios and commended the adaptability and problem-solving skills demonstrated throughout the project, even though the car didn't completely work as intended.

- **Suggestions for Improvement:** Hans Ameel suggested increasing the distance between the walls to reduce the impact of the car deviating from the path. This was based on the observation that the car tended to slightly drift off course due to the narrow maze walls.

11.5 Methodological Rigor and Insights

Building the custom maze environment was by far one of the most fun parts of this project for me. It gave me a more solid understanding of the possibilities and limitations of reinforcement learning. In our course `Advanced AI`, we did get to play with OpenAI Gym quite a bit, but it was always a pre-built environment. Building my own environment made me think about the actions I needed, the observations I wanted to make, the rewards I wanted to give, and how the sensors would work in the environment.

One thing that surprised me was how my virtual twin setup did not really add much value. I thought it would be something nice and useful, but it turned out that a 'simple' top-down view camera would be way more effective for real-time feedback, since the virtual twin didn't show when or where the real car was stuck. Additionally, the top-down camera could have been used as an additional input for the agent to help it navigate the maze along with the ultrasonic sensors.

11.6 Educational Value

The educational value of this project is huge. By documenting the whole process and the challenges faced, this project becomes a fantastic learning tool for anyone interested in AI and robotics. It shows how to apply reinforcement learning in the real world, effectively bridging the gap between theory and practice.

This project is all about hands-on learning. Students and researchers can set up their own experiments to see how reinforcement learning, sensor calibration, and robotic control work in real life. This kind of hands-on experience is invaluable for understanding these complex concepts.

For teachers, this project can serve as a useful resource. It can enhance AI and robotics courses by demonstrating how theoretical concepts apply in real-world scenarios. Additionally, the documented challenges can be used as case studies to help students with critical thinking.

The project's mix of successes and setbacks can also inspire others to dive into their own AI and robotics projects. It shows that hitting roadblocks is just part of the journey and can lead to major breakthroughs. This can motivate students to keep pushing forward, even when things get tough.

By making the project's code and documentation open source, it becomes an even more powerful educational tool. Other students and researchers can build on this work, make improvements, and adapt the methods for their own projects. This creates a collaborative learning environment where knowledge and resources are shared, promoting continuous learning and innovation in AI and robotics.

11.7 Personal Growth and Aspirations

This project made me realize how much I love research. Exploring new ideas, overcoming obstacles, and starting from scratch to build something tailored to my needs was incredibly rewarding. This project allowed me to see the potential of combining AI and robotics in practical applications.

Moving forward, I'm committed to lifelong learning, something that our school really values and continuously brings up. I'm excited to see where this journey will take me and I'm looking forward to the even bigger and more complex challenges that I will have the opportunity to solve.

11.8 A Final Reflection on This Journey

Even though this was an incredible project to work on and I loved every second of it, I can't help but feel like I limited myself by not actively researching before starting this project. I went into this (and many other projects before this) headfirst, starting to make a maze in Pygame even before the three-week period began. However, by doing this, I limited myself by not exploring other methods of completing this maze, like limiting the action space and forcing me to try and replicate the exact movement. I feel that I should have done more research on the hardware before starting this project, especially on how the encoders for the motors work and how to use them.

I doubt I would've been able to get as far as fast as I did if I had taken an alternative route like giving the agent full control over the motors like a Reddit user suggested, but I do think it would have been a lot more interesting and maybe even more rewarding. This also might be a good thing, as it shows that there is always room for improvement and that there is always something to learn from every project you do.

Feedback from Reddit came from several people who filled in my Google form. I posted the link on the subreddit `r/reinforcementlearning`, a platform that has been helpful in the past. Despite not receiving a lot of responses, the feedback did confirm some of the things I had been thinking about.

12 Advice for Students and Researchers

12.1 Practical Utilization of Simulations

Simulations are super handy in research because they let you develop and test algorithms in a safe, controlled environment without needing physical prototypes right away. They save a lot of money by cutting down on the need for physical models and endless real-world trials during the early stages.

Simulations also let you quickly try out new ideas and see how they work without having to build and rebuild physical models. This speed and flexibility are invaluable when you're developing complex systems like autonomous RC cars.

12.2 Strategies for Effective Transition from Simulation to Reality

Moving from simulations to real-world applications isn't always easy, but with the right plan, it's totally doable. Start by refining your algorithms in simulations and then gradually introduce real-world testing. Make sure to have continuous feedback loops to tweak your simulation models based on what you learn from the real world, making them more accurate and useful.

Incremental testing is key. Start small with controlled, simple scenarios that closely match your simulations. As your models prove themselves, gradually introduce more complexity and variability. This helps you catch issues early and adapt your models step by step.

Another great strategy is to use hybrid testing environments where simulations and real-world tests are combined. This approach lets you validate your models in a controlled setting before fully transitioning to the real world.

12.3 Overcoming Common Challenges in Simulation-to-Reality Transitions

Making sure your simulations match real-world conditions can be tricky, especially with sensor data and mechanical operations. Regularly calibrating your sensors and ensuring that the physical movements match what the simulations expect is key to smooth transitions.

For example, sensor noise can be a big issue. In simulations, you can control for noise, but real-world sensors will always have some level of unpredictability. Implementing noise models in your simulations can help your algorithms learn to handle real-world data more effectively.

Mechanical discrepancies are another challenge. Simulated environments often assume perfect conditions, but real-world robots deal with friction, slippage, and wear and tear. Continuously comparing your simulation data with real-world results and adjusting accordingly can help mitigate these issues.

12.4 Insights from My Research

During my research, I found that picking the right simulation platform is super important. Tools like OpenAI Gym are great, but for more complex scenarios, you might need additional tools. I also discovered that Double Deep Q-Network (DDQN) outperforms other models like DQN and PPO by reducing overestimations and making learning more stable.

A big takeaway is that you shouldn't be afraid to change the simulation environment after you start testing in the real world. I Did not think about this at all, in my mind the environment was perfect but it turned out that it was not. By adjusting the simulation to better account for real-world conditions, you can save time and resources by not having to try to perfectly recreate the simulation in the real world.

12.5 Methodological Advice

Use both qualitative and quantitative methods to thoroughly evaluate how well your simulations and real-world applications are working. Stay flexible and open to feedback to address any unexpected challenges effectively.

Documentation is your friend. Keep detailed logs of what works and what doesn't, including all the tweaks and adjustments you make along the way. This not only helps you track progress but also provides a valuable resource for troubleshooting and future projects.

Engage with the community. Platforms like Reddit or GitHub projects of other people can provide valuable feedback and suggestions from other researchers and enthusiasts who have faced similar challenges. Their insights can be incredibly helpful in refining your approach.

12.6 Practical Experiment Integration

Iterative design and prototyping are the way to go. This approach helps you progressively refine your systems, linking theoretical research with practical implementation. Regularly seek and incorporate feedback from stakeholders and peers to improve both your simulation models and real-world applications.

Prototyping doesn't just apply to your final product. Prototype your methods, too. Try different simulation setups, test various algorithms, and iterate on your approach. This experimentation phase is crucial for finding the most effective methods and tools.

12.7 Importance of Fail-Safe Mechanisms and Duplicate Components

Having fail-safe mechanisms and duplicates of all components is crucial. In my project, I learned the hard way that mounting components can sometimes lead to accidental damage. I ended up destroying several parts, which meant I had to wait for replacements and lost a lot of time. To avoid this, always have spares on hand. This ensures that if something breaks, you can quickly replace it and keep your project moving forward.

Fail-safe mechanisms are also vital. These systems can help prevent damage to your components by automatically shutting down the system if something goes wrong. This not only protects your hardware but also saves time and money in the long run.

12.8 Guidelines for Future Research

12.8.1 Introduction for Future Research

This chapter outlines a comprehensive methodology for researchers involved in simulation-based studies, focusing on smoothly transitioning from theoretical models to practical applications.

12.8.2 Step-by-Step Plan

Step 1: Selection of Simulation Environments

Research and evaluate different simulation tools like OpenAI Gym, Unity 3D, AirSim, CARLA, and ISAAC Gym. Set up criteria focusing on fidelity, ease of use, integration capabilities, and specific needs like sensor simulation. Do some initial tests to see how well each platform replicates real-world conditions using simple test cases.

Survey the community and read reviews. Other researchers' experiences can provide insights into the strengths and weaknesses of each platform. Consider joining forums or groups focused on the tools you're interested in to gather more detailed user feedback.

Step 2: Designing the Simulation Environment

Create a custom maze environment like RCMazeEnv in OpenAI Gym. Make sure it includes realistic physical properties like friction and wheel slippage. Integrate virtual sensors that match your real-world sensors and add elements like sensor noise, dynamic obstacles, and varied lighting conditions to mimic real-world challenges.

Don't forget to iterate on your environment design. Start with a basic setup and gradually add complexity. This step-by-step increase in difficulty helps ensure that your algorithms can handle real-world unpredictability.

Step 3: Incremental Testing and Feedback Integration

Start with initial simulation testing to see how well the agent learns and performs. Gradually introduce real-world testing with controlled, simple scenarios that match your initial simulation setup. Use continuous feedback from these tests to refine the simulation environment, adjusting parameters based on what you observe.

Document each test and its results meticulously. This data will be invaluable for identifying patterns and making informed adjustments. Also, consider using automated tools to gather and analyze test data to streamline the process.

Step 4: Addressing Sensor Discrepancies and Movement Alignment

Regularly calibrate your sensors to ensure accurate data collection in both simulations and real-world tests. Make sure the movement mechanics of the simulated and real RC cars are consistent, including motor speeds, wheel slippage, and turning radii.

Develop a routine for sensor calibration and stick to it. Consistency is key to ensuring that your data remains reliable over time. Also, consider using calibration tools or software that can automate parts of this process. This counts for both the real-world and simulated sensors. Sometimes the code for the sensors in the simulation can be off, and after adjusting things in the real world, this is something that can be easily overlooked.

12.8.3 How I would start over if I Could

If I could start over, I would begin by researching the hardware more thoroughly. I spend a lot more time deciding on the hardware I need and how to use it.

RC Car Selection

I would still use the same DFRobot 2WD MiniQ robot kit, but I would order the motor encoders at the same time as the car. I would also take some stronger motors than the default ones in the kit, and the same thing for the wheels, I would get some stronger wheels, since the ones in the kit are quite flimsy.

Motor Encoders: https://www.dfrobot.com/product-823.html

Wheel Kit: https://www.dfrobot.com/product-1558.html

Motor Kit: https://www.dfrobot.com/product-1487.html

- This alternative motor acts as a 360-degree servo motor, which would be perfect for getting those precise movements. And it works with the Arduino servo library, so it would be easy to implement. This would definitely be a better choice than the motors I used in my project.

Sensor Selection

As for the sensors, stay with the HC-SR04 ultrasonic sensors, but I consider adding a top down camera to the whole setup. This might be difficult to implement in the simulation, but it would be such a great addition to the real-world setup.

As seen in this image: Reference:45 in the appendix Or for a more detailed view, this image: Reference:20 below.

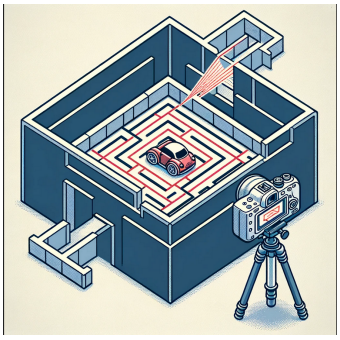


Figure 20: Top Down Camera (Image created by DALL-E)

Alternative environment design

Note: This, in my opinion is not entirely necessary, but it would be an alternative to existing environment design. I believe that with the previous suggestions, the current environment would be sufficient.

As I mentioned before, A more complex environment could help. Now this does not mean using a different library or software, but rather just a more complex action space or just a bigger maze. This would give the agent the opportunity to have more flexibility in its movements.

You would potentially need to adjust the agent to allow for diagonal movements, and adjust the training parameters to allow for longer training times and more exploration. Adjusting the termination conditions should also allow for more steps to be taken.

Alternative maze layout:

Figure 21: A 3-block wide path maze layout

12.9 Conclusion for my advice for students and researchers

By following these steps, researchers can systematically improve their simulation-to-reality projects, ensuring more accurate and reliable outcomes. This methodical approach leverages continuous feedback, interdisciplinary collaboration, and iterative testing to effectively bridge the gap between simulations and real-world applications effectively.

I hope highlighting these challenges throughout my thesis and solutions will help future students and researchers navigate similar projects more effectively or at the very least give them some ideas on how to avoid some of the mistakes I made.

Like I mentioned in my reflection I learned that the virtual twin setup I initially installed didn't add much value, since it can't show us where or why the car was stuck. This showed me the importance of choosing the right tools and being open to simpler and alternative solutions, I ignored this suggestion from Wouter at first, since in my mind the camera would be `too much` and a waste of time. So let this be a lesson to you, always be open to suggestions and feedback. Even if you think you have the `best` solution, at the very least try it out before dismissing it.

13 Sources of Inspiration and Conceptual Framework

The inspiration for this research comes from a mix of technical documentation, digital platforms, and academic literature. Key influences include the challenges of micro mouse competitions and the potential of reinforcement learning (RL) to navigate complex mazes. My interest was further sparked by dynamic RL applications in autonomous vehicle control showcased on YouTube and GitHub, alongside influential academic research.

13.1 Micro Mouse Competitions and Reinforcement Learning

Micro mouse competitions, where small robotic mice navigate mazes, served as a major inspiration. The use of RL in these competitions showed the potential for solving real-world problems and controlling autonomous systems. Insights from a Medium article by M. A. Dharmasiri on maze traversal algorithms and shortest path strategies provided practical algorithmic approaches relevant to this study[12].

13.2 Influential YouTube Demonstrations and GitHub Insights

YouTube videos like "Self Driving and Drifting RC Car using Reinforcement Learning"[9] and "Reinforcement Learning with Multi-Fidelity Simulators – RC Car"[13] vividly demonstrated RL's real-world applicability and the feasibility of Sim2Real transfer. GitHub repositories, such as the "Sim2Real_autonomous_vehicle" project[11], detailed the practical steps and challenges of implementing RL in physical systems.

13.3 Technical Exploration and Academic Foundation

Academic articles also significantly shaped this research. Notable works include Q. Song et al.'s article on autonomous driving decision control[10] and a survey by W. Zhao, J. P. Queralta, and T. Westerlund on Sim2Real transfer in deep RL for robotics[14]. These articles provided in-depth methodologies and highlighted the challenges of applying RL to autonomous systems.

13.4 Conclusion for sources of inspiration and conceptual framework

These were some of the key sources that led me to explore the potential of RL in navigating mazes using an RC car. By combining insights from micro mouse competitions, YouTube demonstrations, GitHub projects, and academic literature, I was able to develop a comprehensive research framework that made all this possible. Even if all of these sources didn't directly contribute to the final product, they all played a role in shaping the project and my understanding of the field.

14 General Conclusion

This thesis journey has been about exploring if a reinforcement learning (RL) agent, trained in a simulation, can effectively navigate a real-world maze with a remote-controlled (RC) car. It turns out the answer is yes, but with a lot of effort and overcoming significant challenges.

The biggest hurdle was making sure the data from sensors and the control algorithms worked as well in real life as they did in the simulation. It took a lot of tweaking to get things aligned properly. Picking the right virtual environments and RL techniques was crucial. The Double Deep Q-Network (DDQN) was the best fit for this project, offering the stability and performance needed for both the simulated and real-world maze navigation.

When I moved from the simulation to the real world, I ran into issues like movement discrepancies and sensor alignment problems. I tackled these with motor encoders, power adjustments, and adding a gyroscope, which helped but didn't solve everything. The research showed that transferring from simulation to reality is possible but requires careful planning and continuous tweaking. This study dives deep into the nuts and bolts of making this transition work.

This project highlights the need to blend theory with hands-on practice to make simulations work in the real world. Moving from a controlled environment to the unpredictable real world is challenging and needs constant adjustments. The reflections on the methodology and the importance of feedback and continual improvement were central to this project's success.

Working on this thesis has been a significant learning experience. Early on, a deeper understanding of hardware would have saved a lot of headaches. The challenges faced and the solutions found have taught me a lot about the iterative nature of research and the importance of staying open to new ideas.

For those venturing into this field, start with simpler simulations and gradually add complexity. Regular sensor calibration and consistency in movement mechanics between simulation and reality are key. Don't hesitate to seek feedback from the research community; it can provide valuable insights and improve your work. I wish I had done this more before starting my project.

In the end, making a trained RL agent work in the real world is doable, but it takes meticulous planning, flexibility, and ongoing refinement. This project underscores the need for continuous efforts to make Sim2Real applications robust and reliable. It contributes to the field by offering a detailed roadmap for future research, emphasizing the need for a systematic approach, continuous feedback, and practical experimentation to bring RL from simulation into the real world successfully. Thank you for joining me on this journey, and I hope this research inspires others to explore the exciting intersection of reinforcement learning and real-world robotics.

15 References

[1] G. Brockman et al., "OpenAI Gym," arXiv preprint arXiv:1606.01540, 2016. [Online]. Available: https://arxiv.org/abs/1606.01540. [Accessed: Jan. 29, 2024].

[2] A. Dosovitskiy et al., "CARLA: An Open Urban Driving Simulator," in Proc. 1st Annual Conf. Robot Learning, 2017.

[3] J. Schulman et al., "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347, 2017. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed: Jan. 29, 2024].

[4] J. Tobin et al., "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," in 2017 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS), 2017.

[5] K. Bousmalis et al., "Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping," in IEEE Int. Conf. Robotics and Automation (ICRA), 2018.

[6] A. A. Rusu et al., "Sim-to-Real Robot Learning from Pixels with Progressive Nets," in Proc. Conf. Robot Learning, 2016.

[7] S. James et al., "Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks," in Proc. 2019 Int. Conf. Robotics and Automation (ICRA), 2019.

[8] F. Sadeghi and S. Levine, "(CAD)(^2)RL: Real Single-Image Flight without a Single Real Image," in Proc. Robotics: Science and Systems, 2016.

[9] "Self Driving and Drifting RC Car using Reinforcement Learning," YouTube, Aug. 19, 2019. [Online Video]. Available: https://www.youtube.com/watch?v=U0-Jswwf0hw. [Accessed: Jan. 29, 2024].

[10] Q. Song et al., "Autonomous Driving Decision Control Based on Improved Proximal Policy Optimization Algorithm," Applied Sciences, vol. 13, no. 11, Art. no. 11, Jan. 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/11/6400. [Accessed: Jan. 29, 2024].

[11] DailyL, "Sim2Real_autonomous_vehicle," GitHub repository, Nov. 14, 2023. [Online]. Available: https://github.com/DailyL/Sim2Real_autonomous_vehicle. [Accessed: Jan. 29, 2024].

[12] M. A. Dharmasiri, "Micromouse from scratch | Algorithm- Maze traversal | Shortest path | Floodfill," Medium, [Online]. Available: https://medium.com/@minikiraniamayadharmasiri/micromouse-from-scratch-algorithm-maze-traversal-shortest-path-floodfill-741242e8510. [Accessed: Jan. 29,

2024].

[13] "Reinforcement Learning with Multi-Fidelity Simulators – RC Car," YouTube, Dec. 30, 2014. [Online Video]. Available: https://www.youtube.com/watch?v=c_d0Is3bxXA. [Accessed: Jan. 29, 2024].

[14] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: A Survey," in 2020 IEEE Symp. Series Computational Intelligence (SSCI), Dec. 2020, pp. 737–744. [Online]. Available: https://arxiv.org/pdf/2009.13303.pdf. [Accessed: Jan. 29, 2024].

[15] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA: The MIT Press, 2018.

[16] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," arXiv preprint arXiv:1509.06461, 2015. [Online]. Available: https://arxiv.org/abs/1509.06461. [Accessed: Jan. 29, 2024].

[17] "Double DQN Explained," Papers With Code, [Online]. Available: https://paperswithcode.com/method/double-dqn. [Accessed: Jan. 29, 2024].

[18] D. Jayakody, "Double Deep Q-Networks (DDQN) - A Quick Intro (with Code)," 2020. [Online]. Available: https://dilithjay.com/blog/2020/04/18/double-deep-q-networks-ddqn-a-quick-intro-with-code/. [Accessed: Jan. 29, 2024].

[19] D. Silver et al., "Deterministic Policy Gradient Algorithms," in Proc. 31st Int. Conf. Machine Learning, 2014.

[20] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529-533, 2015.

[21] C. J. C. H. Watkins and P. Dayan, "Q-learning," Machine Learning, vol. 8, no. 3-4, pp. 279-292, 1992.

[23] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in Proc. 13th Int. Conf. Neural Information Processing Systems, 2000, pp. 1008-1014.

[24] T. Saanum, "Reinforcement Learning with Simple Sequence Priors," arXiv preprint arXiv:2305.17109, 2024. [Online]. Available: https://arxiv.org/abs/2305.17109. [Accessed: Jan. 29, 2024].

[25] "AirSim on Unity: Experiment with autonomous vehicle simulation," Unity Blog, 2018. [Online]. Available: https://blog.unity.com/engineering/airsim-on-unity-experiment-with-autonomous-vehicle-simulation. [Accessed: Jan. 29, 2024].

[26] "Introducing Unity Machine Learning Agents Toolkit," Unity Blog, 2018. [Online]. Available: https://blog.unity.com/machine-learning/introducing-unity-machine-learning-agents-toolkit. [Accessed: Jan. 29, 2024].

[27] A. Puigdomènech Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the Atari Human Benchmark," arXiv preprint arXiv:2003.13350, 2020. [Online]. Available: https://arxiv.org/pdf/2003.13350.

[28] V. Makoviychuk et al., "Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning," arXiv:2108.10470 [cs.RO], 2021. [Online]. Available: https://arxiv.org/abs/2108.10470. [Accessed: Jan. 29, 2024].

[29] "Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model," arXiv.org, [Online]. Available: https://ar5iv.labs.arxiv.org/html/1610.03518.

[30] Z. Wang, "Dueling Network Architectures for Deep Reinforcement Learning," in *Proc. 31st Int. Conf. Mach. Learn.*, vol. 48, pp. 1–9, 2016. [Online]. Available: http://proceedings.mlr.press/v48/wangf16.pdf.

[31] C. Rizzardo, S. Katyara, M. Fernandes, and F. Chen, "The Importance and the Limitations of Sim2Real for Robotic Manipulation in Precision Agriculture," arXiv preprint arXiv:2008.03983, 2020. [Online]. Available: https://arxiv.org/abs/2008.03983.

16 Appendices

16.1 Guest Speakers

16.1.1 Innovations and Best Practices in AI Projects by Jeroen Boeye at Faktion

Jeroen Boeye, representing Faktion, shared valuable insights into the synergy between software engineering and artificial intelligence in developing AI solutions. He emphasized the importance of not only focusing on AI technology but also integrating solid software engineering principles to create robust, scalable, and maintainable AI systems. This holistic approach ensures that AI solutions are both technically sound and viable for long-term application.

During his lecture, Jeroen highlighted various aspects of AI application, particularly Chatlayer's contributions to conversational AI. He explained how Chatlayer enhances chatbot interactions through sophisticated conversational flows, improving the accuracy and relevance of exchanges with users. Another key point was Metamaze, which he praised for its innovative methods in automating document processing, creating concise summaries from extensive documents and emails and demonstrating the capabilities of supervised machine learning in administrative tasks.

Jeroen outlined a clear roadmap for successful AI project implementation, stressing the need to validate business cases and adopt a problem-first strategy. He discussed the crucial role of high-quality data as the foundation for any AI endeavor and offered strategies for creatively overcoming data limitations. The talk also covered the importance of viewing failures as opportunities for innovation and maintaining open communication with stakeholders about challenges and setbacks.

The lecture further presented various practical AI applications across different industries, such as solar panel detection, unauthorized pool identification, air freight container inspection, and early warning systems for wind turbine gearboxes. Jeroen demonstrated how AI could tackle complex challenges through innovative data sourcing, synthetic data generation, and anomaly detection techniques. He also explored case studies on energy analysis in brick ovens and egg incubation processes, emphasizing the importance of data preprocessing and machine learning models in improving efficiency and outcomes.

Key points from Jeroen's talk included mastering data preprocessing and treating data as a dynamic asset to better tailor AI models to specific needs. He shared practical tips on enhancing operational efficiency, such as using host mounts for code integration and Streamlit for dashboard creation, to streamline development processes.

In summary, Jeroen Boeye's lecture offered a comprehensive perspective on integrating AI technologies in real-world settings. His insights into the vital role of software engineering principles, alongside a deep understanding of AI capabilities and constraints, provided valuable guidance for developing effective and sustainable AI solutions. The lecture not only underscored current AI trends and future directions but also shared practical knowledge on navigating the complexities of AI project execution.

16.1.2 Pioneering AI Solutions at Noest by Toon Vanhoutte

Toon Vanhoutte, speaking on behalf of Noest from the Cronos Group, delivered an engaging lecture on the effective integration of artificial intelligence and software engineering in developing cutting-edge business solutions. With a dedicated team of 56 local experts, Noest has built a reputation for its pragmatic approach to projects, targeting global impact while valuing craftsmanship, partnership, and enjoyment as core principles. This philosophy extends to their diverse services, which include application development, cloud computing, data analytics, AI innovations, low-code platforms, ERP solutions, and comprehensive system integrations, all supported by a strong partnership with Microsoft.

Toon presented a case study on a packaging company that aimed to revolutionize image search capabilities based on product labels. The project faced various challenges, such as inconsistent PDF formats and large file sizes, which were adeptly managed using Azure Blob Storage for data handling and event-driven processing strategies for efficient, cost-effective solutions, showcasing Noest's skill in utilizing cloud technologies to address complex issues.

Another significant challenge was enhancing image searchability, which involved recognizing text and objects within images. This was tackled using Azure AI Search, supplemented by Large Language Models (LLMs) and vector search techniques. This approach allowed for nuanced search functionalities beyond simple text queries, demonstrating the advanced capabilities of AI in interpreting complex data.

Toon also explored advancements in semantic search, discussing how different search methods—keyword, vector, and hybrid—along with semantic ranking, could significantly improve the accuracy and contextuality of search results. Practical demonstrations, including comparisons between OCR and GPT-4 vision, illustrated the potential of AI to offer deeper insights based on semantic understanding.

A key takeaway from Toon's lecture was the importance of setting realistic client expectations regarding AI's capabilities and potential inaccuracies, highlighting the experimental nature of these technologies. The discussion on AI's evolving landscape emphasized the need for prompt engineering, the challenges of navigating a developing field, and the importance of client education in managing expectations about AI technologies like GPT.

In conclusion, Toon Vanhoutte's presentation not only highlighted Noest's innovative

work in AI and software engineering but also imparted crucial lessons on innovation, adaptable problem-solving, and the necessity for ongoing learning in the dynamic field of AI. This presentation showcased Noest's commitment to pushing technological boundaries to create impactful, pragmatic solutions that fully utilize AI's potential.

16.2 Installation Guide

This guide provides a comprehensive step-by-step approach to installing and setting up the project environment, ensuring a successful deployment of the autonomous navigation system.

16.2.1 Prerequisites

Before starting the setup, make sure you have the following tools and dependencies installed:

- **Git:** Necessary for cloning the project repository.
- **Docker:** Essential for containerizing the web application to maintain a consistent runtime environment.
- **Python 3.11 and pip:** Required if you prefer to run the project without Docker. Install Python and the necessary dependencies listed in `/web_app/web/requirements.txt`.

16.2.2 Repository Setup

First, you need to clone the project repository and navigate to the project directory. Open your terminal and run the following commands:

```
git clone https://github.com/driessenslucas/researchproject.git
cd researchproject
```

16.2.3 Hardware Setup and Assembly

Introduction to Hardware Components

The hardware setup involves several components, including an RC car, sensors, and microcontrollers. Proper integration of these components is critical for the autonomous navigation system to function correctly.

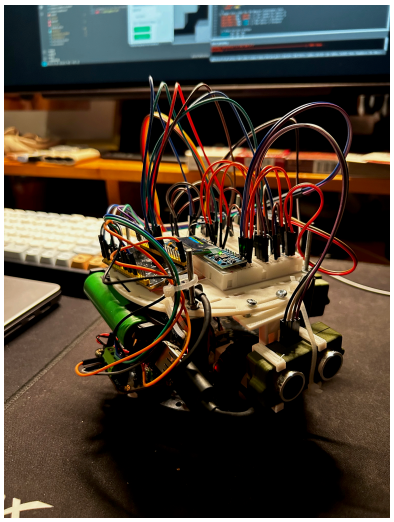


Figure 22: Final RC Car (Image created by author)

16.2.4 Components List

Core Components:

- **ESP32-WROOM-32 module**: Available at Amazon.com
- **3D printed parts**: Available at Thingiverse.com
  - HC-SR04 holders: `https://www.thingiverse.com/thing:3436448/files`
  - Top plate + alternative for the robot kit: `https://www.thingiverse.com/thing:2544002`
- **Motor Controller (L298N)**: Available at DFRobot.com
- **2WD miniQ Robot Chassis**: Available at DFRobot.com
- **Mini OLED screen**: Available at Amazon.com
- **Sensors (HC-SR04 and MPU6050)**: Available at Amazon.com
- **18650 Battery Shield for ESP32**: Available at Amazon.com

Supplementary Materials:

- **Screws, wires, and tools required for assembly**:
  - 4mm thick screws, 5mm long for securing the wood
  - M3 bolts & nuts
  - Wood for the maze (planks cut to 10cm width by 120cm length)
  - Available at most hardware stores

Tools Required:

- Screwdriver
- Wire cutter/stripper
- Drill (for mounting the top plate)

16.2.5 Assembly Instructions

Step 1: Base Assembly

Begin the assembly process by setting up the base of the robot chassis. The base provides the foundation upon which all other components will be mounted. Follow this YouTube video, created by the manufacturers, which provides a detailed visual guide on how to assemble the base correctly:

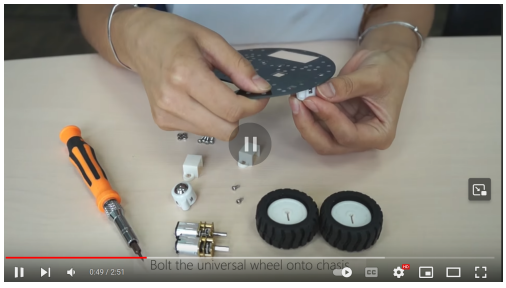


Figure 23: MiniQ 2WD Robot Chassis Quick Assembly Guide

Figure 24: QR code for MiniQ 2WD Robot Chassis Assembly Guide

Step 2: Attach Motor Driver

Next, secure the motor driver to the base using the two screws provided in the kit. The motor driver is a crucial component as it controls the motors that drive the wheels of the RC car. Make sure the motor driver is positioned correctly on the base so that it does not obstruct any other components. Proper placement will ensure that all connections and future steps can be carried out smoothly.

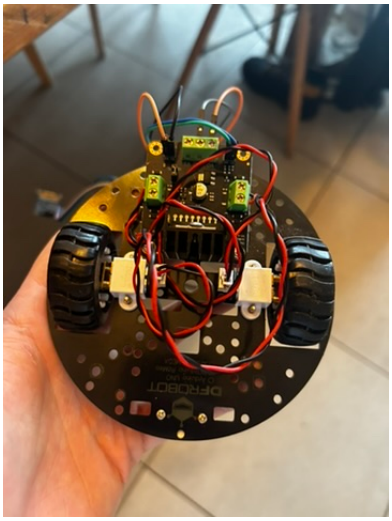


Figure 25: Motor Driver Attached to the Base (Image created by author)

Step 3: Connect ESP32-WROOM-32 Module to the Motor Driver

Now, connect the ESP32-WROOM-32 module to the motor driver. This involves wiring the motor driver to the appropriate pins on the ESP32 module. Follow the electrical schematic below carefully to ensure correct connections. The wires should be connected as follows:

```
int E1 = 2; //PWM motor 1
int M1 = 17; //GPIO motor 1
int E2 = 19; //PWM motor 2
int M2 = 4; //GPIO motor 2

int sensor0Trig = 27; //GPIO right sensor
int sensor0Echo = 26; //GPIO right sensor
int sensor1Trig = 33; //GPIO left sensor
int sensor1Echo = 32; //GPIO left sensor
int sensor2Trig = 25; //GPIO front sensor
int sensor2Echo = 35; //GPIO front sensor

// OLED display pins
```

```
#define SDA_PIN 21 // this is the default sda pin on the esp32
#define SCL_PIN 22 // this is the default scl pin on the esp32
```

These connections allow the ESP32 to control the motor driver and, subsequently, the motors.

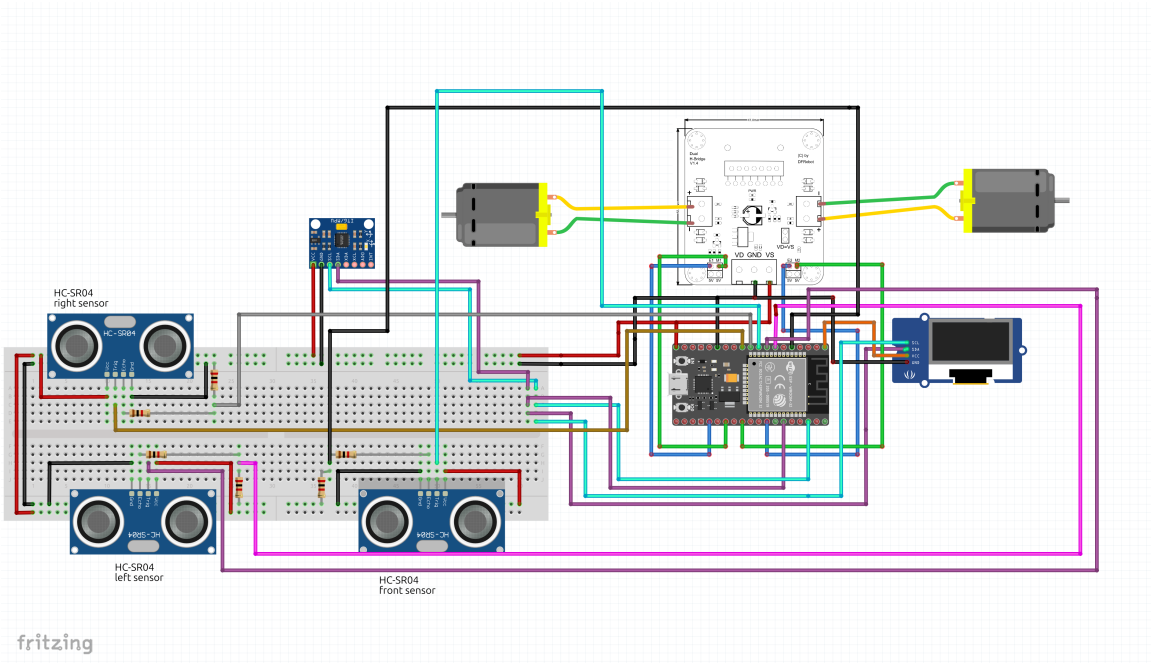


Figure 26: ESP32 Wiring Schematic (Image created by author)

Step 4: Cut the Support Beams

To provide structural integrity and ensure that the top plate can be securely attached, cut the support beams to approximately 7cm in length. These beams will act as spacers between the bottom plate and the top plate, providing enough room for the components mounted on the bottom plate.

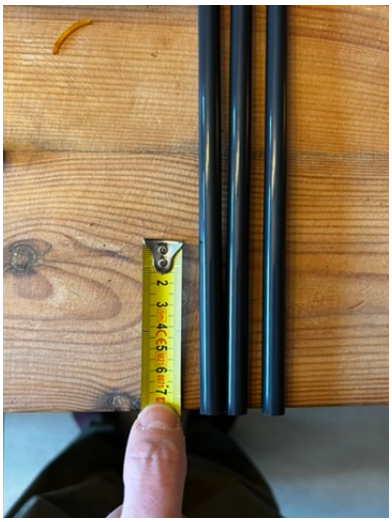


Figure 27: Cut Support Beams (Image created by author)

Step 5: Attach Supports to the Bottom Plate
Secure the support beams to the bottom plate using screws. This step is crucial as it lays the groundwork for attaching the top plate. Make sure the supports are firmly attached and evenly spaced to maintain stability.

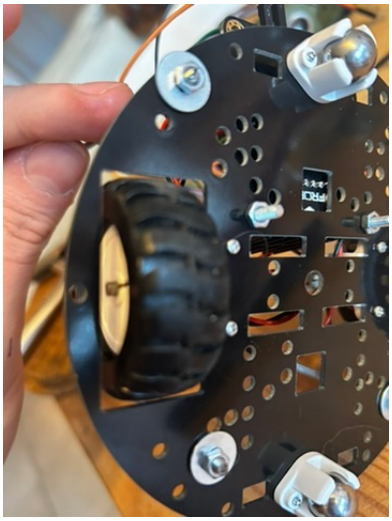


Figure 28: Supports Screwed on the Bottom Plate (Image created by author)

Step 6: Mount All Supports

Once the supports are securely attached to the bottom plate, proceed to mount all of them as shown in the figures. Ensure that all supports are firmly in place to provide a stable structure for the top plate.

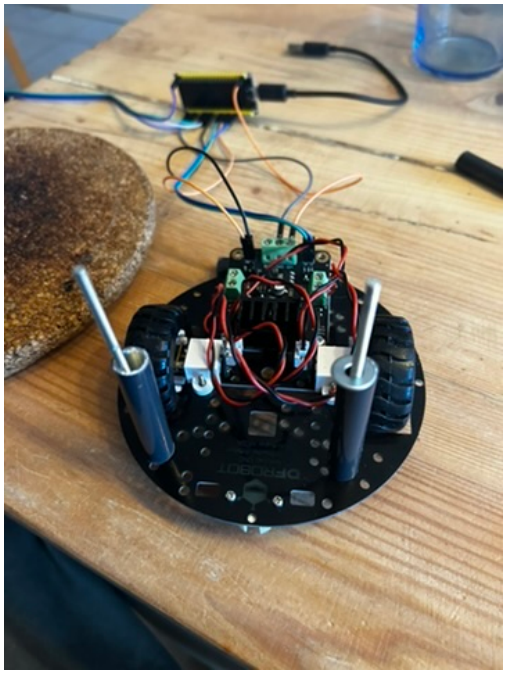


Figure 29: All Supports Mounted (Image created by author)

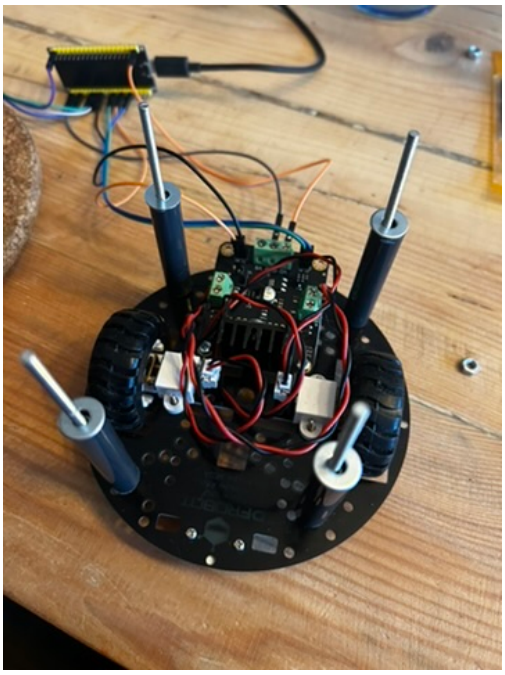


Figure 30: Complete View of Mounted Supports (Image created by author)

Step 7: Attach the Top Plate

Drill holes in the top plate to align with the support beams. Once the holes are drilled, attach the top plate securely to the support beams. This step finalizes the structural assembly and provides a stable platform for the remaining components.

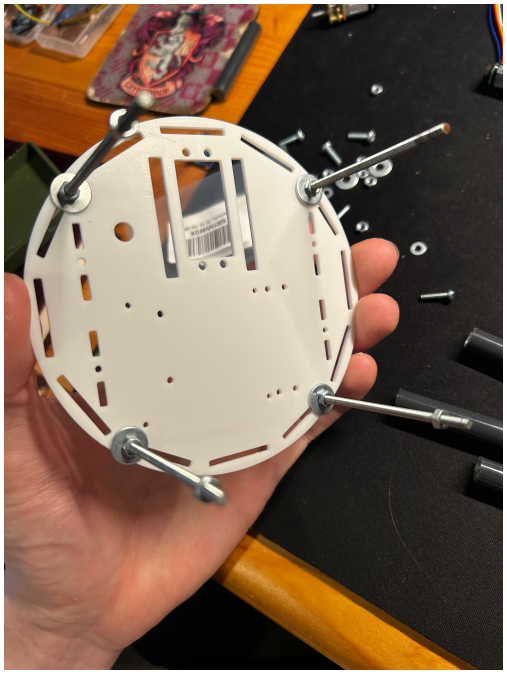


Figure 31: Top Plate Assembly (Image created by author)

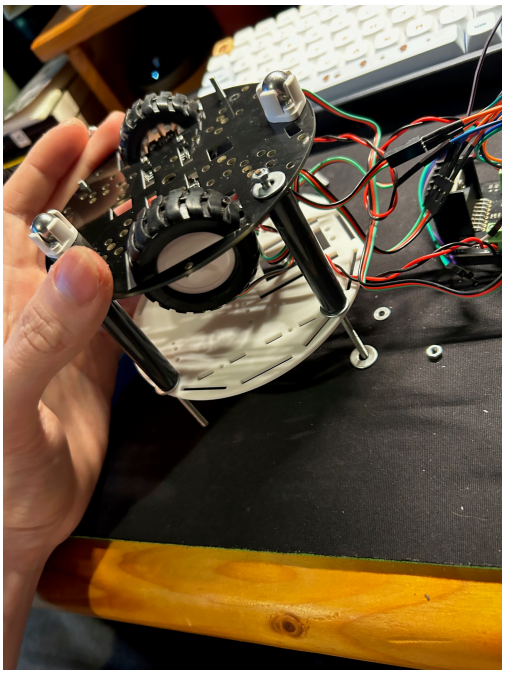


Figure 32: Bottom View with Supports and Top Plate (Image created by author)

Step 8: Attach the Ultrasonic Sensor

Mount the ultrasonic sensor to the top plate. This sensor is essential for the RC car's autonomous navigation as it provides distance measurements to detect obstacles. Secure it firmly to ensure it remains stable during operation.

Figure 33: Ultrasonic Sensor Attached to the Top Plate (Image created by author)

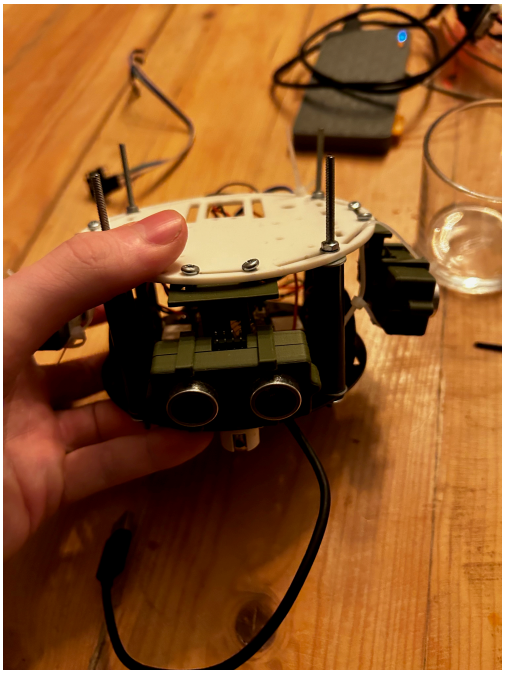


Figure 34: Ultrasonic Sensors Attached (Image created by author)

Step 9: Place the ESP32 on the Top Plate

Finally, place the ESP32 module on the top plate along with a mini breadboard for the sensor wires. Secure the ESP32 battery with zip ties to ensure it stays in place during operation. This completes the hardware assembly of the RC car.

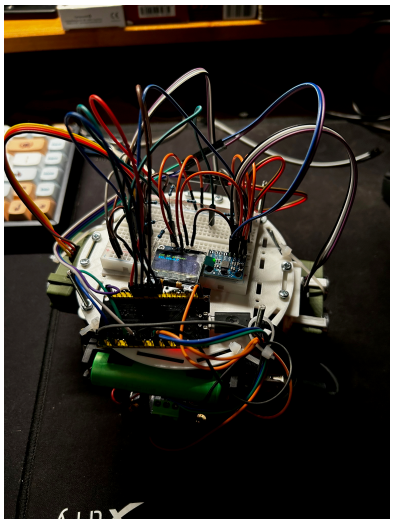


Figure 35: ESP32 Placement on Top Plate (Image created by author)

16.2.6 Wiring Guide

ESP32 Wiring

The ESP32 wiring connections are

shown in the diagram below. The pins connect to the motor driver, sensors, OLED display, and MPU6050 gyroscope. Follow the schematic carefully to ensure all connections are correct.

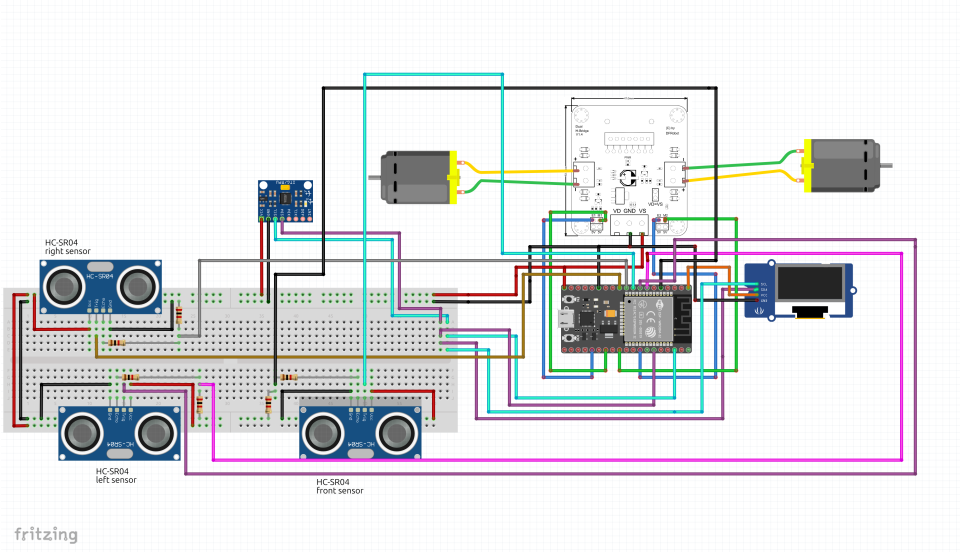


Figure 36: Wiring Diagram for ESP32 (Image created by author)

16.2.7 Software Configuration

1. **Arduino IDE Setup**: Install the Arduino IDE to program the ESP32 microcontroller. Follow the instructions from Espressif Systems to add the ESP32 board to the Arduino IDE. This will allow you to write and upload the necessary code to the ESP32.

2. **Library Installation**: Install the `ESP32_SSD1306` library for OLED display functionality. This library provides the functions needed to interface with the OLED screen and display information.

3. **Code Upload**: Transfer the scripts from the `esp32` folder to the ESP32 device. Modify the WiFi settings in the script to match your local network configuration. This will enable the ESP32 to connect to your network and communicate with other components.

16.2.8 Web Application Setup

Note To ensure a smooth setup of the virtual display, it is recommended to run `docker-compose down` after each session. This will stop and remove any existing containers, ensuring a clean start for the next session.

Steps

1. Navigate to the web application's source code directory:

   ```
   cd ./web_app/
   ```

   This command changes the directory to the location of the web application's source code.

2. Launch the Docker containers with:

   ```
   docker-compose up -d
   ```

   This command starts the Docker containers in detached mode, allowing the web application to run in the background. Docker ensures that all dependencies are correctly set up and that the application runs in a consistent environment.

16.2.9 Usage Instructions

1. Open your web browser and go to `http://localhost:8500` or `http://localhost:5000`. This URL opens the web application interface where you can control and monitor the RC car.
2. Enter the ESP32's IP address in the web app and select the desired model for deployment. This step allows the web app to communicate with the ESP32 and send commands.
3. You can also run a virtual demonstration without engaging the physical vehicle. This feature is useful for testing the software and ensuring that everything is working correctly before using the actual RC car.
4. Start the maze navigation by clicking the `Start Maze` button. This command initiates the autonomous navigation system, allowing the RC car to navigate the maze.

A demonstration of how to use the interface is available (see Web App Demo in the appendices under `Extra Content`)

16.2.10 Additional Information: Model Training

- You can use a pre-trained model or train a new model using the script in `train/train.py`.
- This training script is optimized for efficiency and can be run directly on the Raspberry Pi.
- After training, you will be prompted to save the new model. If saved, it will be stored in the `web_app/models` directory of the `web_app` folder.

By following these steps, you can successfully set up and deploy the autonomous navigation system, ensuring it runs smoothly both in simulations and real-world scenarios.

16.2.11 Building the Maze

Result

The following images show the final build of the maze used in the project.

Figure 37: Maze Build (Image created by author)

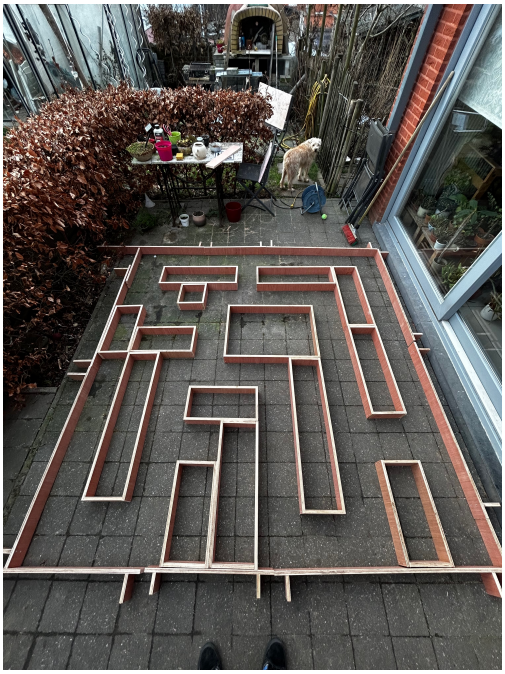


Figure 38: Final Maze Build (Image created by author)

Prerequisites

The materials and tools required for building the maze are listed below:

Figure 39: Screws (Image created by author)

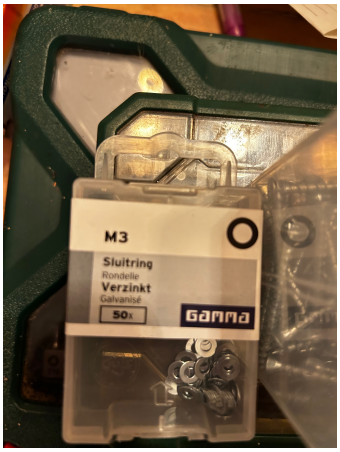


Figure 40: Nuts (Image created by author)

Figure 41: Supports (Image created by author)

- Wood used:
  - Planks cut to 10cm width by 120cm length

Figure 42: Wood Planks (Image created by author)

Step 1: Calculations

Each cell in the maze is 25cm x 25cm. Calculate the number of planks and their lengths needed to build the maze.

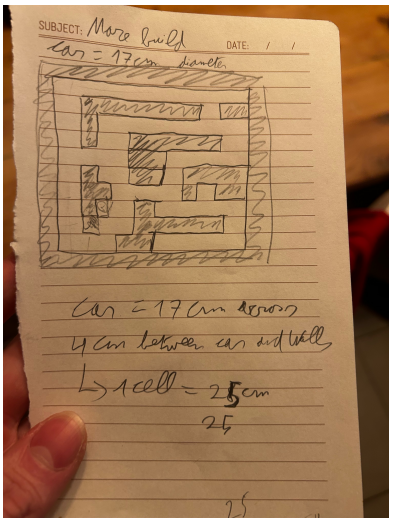


Figure 43: Size Calculations for Maze (Image created by author)

Step 2: Cut the Wood

Cut the wooden planks to the correct size (10cm x 120cm) to form the walls of the maze. You can have the store cut the wood for you or do it yourself using a saw.

Step 3: Screw the Wood Together

Drill holes and screw the wood pieces together to form the maze structure. Ensure the joints are secure to prevent any movement during the RC car's navigation.

Drilling Wood Frames for Maze (Image created by author)

It should turn out like this; repeat this for all the blocks in the maze:

Figure 44: Wooden Frames for Maze (Image created by author)

By following these steps, you can successfully set up and deploy the autonomous navigation system, ensuring it runs smoothly both in simulations and real-world scenarios.

16.3 Extra Content

16.3.1 Top down camera view of the maze (self drawn)

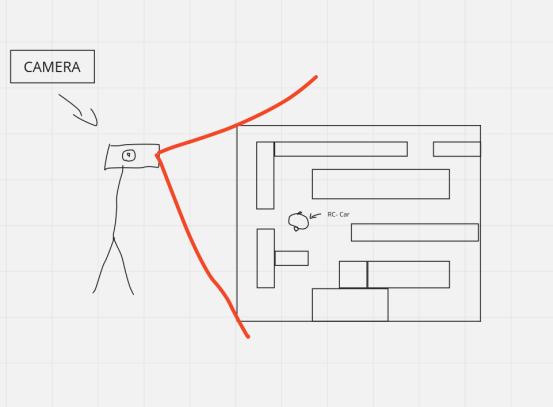


Figure 45: Top Down Camera (Image drawn by author)

16.3.2 Video References

Web App Demo: A demonstration of the web application's functionality, showcasing the user interface and the autonomous navigation system's control features.

- Click here to go to the video or scan the QR code below: Web App Demo

Figure 46: QR code for Web App Demo. (Video by author.)

DDQN Simulation test: A simulation test of the DDQN model navigating a maze environment, demonstrating the model's learning capabilities and decision-making processes.

- Click here to go to the video or scan the QR code below: DDQN Simulation

Figure 47: QR code for DDQN Simulation. (Video by author.)

All other videos: Videos of some of the tests I did are available in the GitHub repository. You can access following the link below:

`https://github.com/driessenslucas/researchproject/tree/main/videos/`

Or by scanning the QR code below:

Figure 48: QR code for some test videos. (Videos by author.)