Human-aware robot navigation using reinforcement learning

Lisa Spahn Lundgren (lissp373) TDDE05 Project Report October 23, 2021

1 Introduction

Autonomous mobile robots have been used for many years and in the past have often been utilized in human-free environments that are hazardous or hard to reach, such as space [2] or mining robots [1]. For robots to navigate these environments autonomously, mobile robot navigation has been researched extensively and is still continuously developing.

In recent years, deploying autonomous robots in human environments has gained in popularity as well. The applications range from vacuum cleaning robots that clean areas while humans or other objects are present, all the way to service robots, that directly interact with humans. Hence being able to avoid humans and obstacles is essential in order to guarantee the safe coexistence of humans and robots in these environments. [18]

A commonly used framework to control mobile robots is the Robot Operating System ROS [16]. It is a set of open-source software that comes with numerous libraries that help develop robot applications. Furthermore, it provides the communication interface to run and execute the developed applications either on the robot directly or in simulation. In this paper, ROS was used together with the open-source 3D robotics simulator Gazebo [9], to run the developed code on a simulated robot in a virtual environment.

1.1 Motion planning with obstacle avoidance

A commonly-used strategy for robot navigation, especially when controlling the agent using ROS, is to first use a path planner that calculates the complete path from agent to goal. This path planner usually runs at a lower frequency, as the path can be very long and computationally expensive. This plan will guide the agent around static obstacles such as walls and other mapped obstacles. Temporary obstacles such as humans can but do not have to be respected. In addition, a motion planner with a shorter horizon is used which runs at a higher frequency. It is the motion planner's task to follow the given plan and react to obstacles or changes that have occurred in between the re-planning times. Examples for obstacle avoiding motion planners are the dynamic window approach (DWA) [6], the elastic band approach [17] or model predictive controllers (MPC) [11], where the latter has the highest potential at avoiding dynamic obstacles due to its predictive nature.

1.2 Planning using learning

While the above-mentioned solutions are well researched and fairly reliable for static obstacles, it becomes more challenging when the obstacles are moving, such as humans in an indoor environment or cars in an outdoor environment. In a 2021 survey on social robot navigation, Mavrogiannis et. al [13] state that since the planning is not coupled to any future motion prediction of the obstacles, there is a great risk of the agent freezing, turning or oscillating on the spot, also known as the reciprocal dance problem [8]. Reasons for that can be that the global plan and the motion plan are conflicting, for instance if the global plan passes the obstacle on the right whereas the motion planner wants to send the agent past the left side of the obstacle. This problem also applies to MPC based solutions, as the global planner is still separated from the local MPC motion planner that does the predictions.

Hence in recent years, navigation strategies where the planning and obstacle motion prediction are coupled have been developed. These methods are generally learning-based, ranging from deep learning, imitation learning or generative adversarial networks to reinforcement learning (RL) [13]. These coupled approaches are especially powerful for dynamic obstacles, as the obstacles' predicted future trajectory is included in the planning process, meaning that everything happens in one layer and thereby preventing undesired behaviours due to for instance conflicting plans.

In this paper, a learning-based method using reinforcement learning was developed to achieve autonomous navigation while also avoiding dynamic obstacles. RL is an intuitive trial-and-error way of learning, that is similar to how humans would acquire a new skill. As shown by Everett et al. in 2021 [5] and Liu et al. in their 2020 paper [12], the agent can be very successful at navigating

pedestrian-rich environments using deep RL. For the scope of this project however, I decided to use the rather primitive Q-learning algorithm [20] instead of more advanced learning techniques such as deep Q-learning. The Q-learning environment is implemented using the ROS package `openai_ros`[1], which combines the open-source RL toolkit OpenAI Gym [3] with ROS and the Gazebo simulator, to train and later navigate an agent that is controlled using existing ROS libraries and simulated in Gazebo. More specifically, the ROS standard platform robot TurtleBot3[2] is used.

I expect the Q-learning based navigation solution to outperform other obstacle-avoiding motion planning methods such as DWA or the elastic band approach that were mentioned in section 1.1, as these do not use any information about the future trajectory of the humans and hence the resulting plan might not be optimal. To achieve good obstacle avoidance, it is important to identify patterns in the movement of the dynamic objects and use that knowledge to avoid them. In the case of humans, this could be that humans are most likely to walk in the direction in which they are facing. The probability of walking backwards or sideways is much lower. Therefore I expect my solution to learn those patterns and plan the agent's actions accordingly.

2 Methods

Given an environment with one agent and n independent humans, the learning space can be modelled as a two-dimensional grid map, see Quan et al. [15]. The agent and humans are then mapped to the respective X-Y-cell on that grid. The assumption is made that humans do not avoid or react to the agent, to make the humans' behaviour more predictable.

2.1 Q-learning

The chosen Q-learning method is a model-free reinforcement learning technique, meaning that the environment can not be modelled by a function. Instead, the agent will try various actions a, that are defined by action space A, in different states $\boldsymbol{s}$ and evaluate the profit of performing this action in that particular state, which is calculated by a reward function $R(\boldsymbol{s}, a)$. Based on that, a Q-table is built that lists the expected reward, called Q-value $Q(\boldsymbol{s}, a)$, for each action in every discovered state [20]. This Q-table can then be used as a look-up table to find the optimal action a^* in a given state $\boldsymbol{s}$ by finding the action that maximises the Q-value for that state, also known as the policy π. This can be described as follows

$$\pi(\boldsymbol{s}) = \underset{a \in A}{arg\,max}\ Q(\boldsymbol{s}, a). \tag{1}$$

At a time step t, the Q-value of state $\boldsymbol{s}_t$ when choosing action a_t is updated to the value $Q'(\boldsymbol{s}_t, a_t)$ using the following formula

$$Q'(\boldsymbol{s}_t, a_t) \leftarrow Q(\boldsymbol{s}_t, a_t) + \alpha \cdot \big(r_t + \gamma \cdot Q^*(\boldsymbol{s}_{t+1}, a) - Q(\boldsymbol{s}_t, a_t)\big), \tag{2}$$

with

$$r_t = R(\boldsymbol{s}_t, a_t) = R(\boldsymbol{s}_{t+1}) \quad \text{and} \quad Q^*(\boldsymbol{s}_{t+1}, a) = \underset{a \in A}{max}\ Q(\boldsymbol{s}_{t+1}, a).$$

$Q(\boldsymbol{s}_t, a_t)$ is the old Q-value for the state-action pair at time t before the Q-value update. $Q^*(\boldsymbol{s}_{t+1}, a)$ is the estimated maximal future Q-value when choosing the optimal action in the future state $\boldsymbol{s}_{t+1}$, that was reached after performing action a_t in state $\boldsymbol{s}_t$. This can also be interpreted as the estimated future reward. r_t denotes the reward that was obtained from executing action a_t in state $\boldsymbol{s}_t$. The parameter $\alpha \in [0, 1]$ is called learning rate, as it determines how much old information is replaced by the new information. A value of zero does not update the Q-value at all. A value of one will completely overwrite the old value with the obtained reward and the estimated future reward. Lastly, the parameter $\gamma \in [0, 1]$ is known as the discount factor and controls the weighting of future rewards. Lower values promote a greedy behaviour, where the agent tries to get the maximal reward mainly in the current state, whereas larger values result in a policy that maximizes the long-term rewards.

[1] `http://wiki.ros.org/openai_ros` [2] `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`

2.1.1 Q-learning algorithm

The training is performed in episodes. The number of required episodes for the Q-table to converge to an optimal solution is different for every environment, depending on the size of the state-action space and the hyperparameters. Each episode starts with the agent being in its initial position and ends when the agent has reached the goal, when the agent has crashed into an obstacle or when a maximum number of steps has been reached. If any of those termination conditions are true, the episode is considered done, the environment is reset and a new episode starts. The last check, if a maximum number of steps has been reached, helps to speed up the training. Moreover, since one of the objectives is to reach the goal as fast as possible, it makes sense to terminate the episode after too many steps, even if the agent did not crash or reach the goal, as the agent will learn that this long sequence of actions will not yield a high reward.

Using the concept and equations that were defined in section 2.1, the Q-learning algorithm for one episode can be described by the following loop:

1. Get the current state $\boldsymbol{s}_t$.
2. Choose an action a_t as follows:
   - with the probability of $\epsilon \in [0, 1]$, choose a random action.
   - with the probability of $1 - \epsilon$, choose the optimal action in state $\boldsymbol{s}_t$.
3. Execute a_t, observe the new state $\boldsymbol{s}_{t+1}$ and calculate the reward r_t of the state-action pair $\boldsymbol{s}_t, a_t$ as the reward in state $\boldsymbol{s}_{t+1}$.
4. Update $Q(\boldsymbol{s}_t a_t)$ according to equation 2.
5. Check if termination condition is satisfied.
   - If termination condition is **true**: Episode is finished, exit loop.
   - If termination condition is **false**: Update $t \leftarrow t + 1$, update decaying parameters α and ϵ, go to step 1 and repeat the loop.

The parameter ϵ is called exploration rate and determines the probability of choosing a random action over an optimal action based on the Q-table. A linear decaying technique defined by equation 3 was chosen for the exploration rate ϵ as well as the learning rate α to improve convergence. By decaying ϵ over time, the agent focuses on exploring the state-action space in the beginning, but will most likely receive many negative rewards due to collisions. After having explored a major part of the learning space, the lower exploration rate promotes choosing the optimal action more often, thereby earning more positive rewards and learning the optimal behaviour [19]. For α, a larger learning rate will update the Q-table in large steps. In the beginning, when the agent does not know much about the state-action space yet, keeping the majority of new information is useful to advance quickly in the learning process. At later stages however, when the agent has already explored and learnt from the environment, decreasing α will help converge to the optimal Q-table and the optimal policy [7]. The parameters are decayed using the following formula

$$p(\textit{current episode}) = p_{max} - (p_{max} - p_{min}) * \frac{\textit{current episode}}{\textit{total nr. of episodes}}, \tag{3}$$

where p denotes an arbitrary parameter and p_{min} and p_{max} its minimum and maximum value.

2.2 Problem formulation

This section describes the mathematical formulation of the environment, showing how the state space, the action space and the reward function are defined.

2.2.1 State space

The state space is based on the problem formulation described by Li et al. [10], however slightly simplified with fewer variables, to make the state space as small as possible. The observed state $\boldsymbol{s}_t$ at time t is defined as follows

$$\boldsymbol{s}_t = [\boldsymbol{s}_{agent,t} \ , \ \boldsymbol{s}_{obst,t}], \tag{4}$$

where $\boldsymbol{s}_{agent,t}$ denotes the agent's state and $\boldsymbol{s}_{obst,t}$ denotes the state of the human that is closest to the agent at time t. These are defined as follows

$$\boldsymbol{s}_{agent,t} = [p_x, p_y, \theta_a] \quad \text{and} \tag{5}$$

$$\boldsymbol{s}_{obst,t} \quad = [r_x, r_y, \theta_o, v_o], \tag{6}$$

with p_x, p_y and θ_a denoting the x-position, y-position and orientation of the agent, r_x and r_y are the x- and y-coordinates of the human relative to the agent, meaning that the human is described in an agent-centric frame to make the formulation more general, θ_o is the orientation of the human and v_o the human's forward velocity. The last two variables, θ_o, v_o, will give the agent information about in which direction and how fast the human is moving and ideally the agent can then learn when to wait for a human to pass or on which side to navigate past the human.

As mentioned in section 2, the environment is mapped onto a 2D grid.To speed up the training, the width and height of the grid cells are set to half a metre. This means that the agent's and human's coordinates are rounded to the closest .5 metre value. Similarly, the orientation of both agent and human is discretized into a set of four values, which represent the directions north, east, south and west. Lastly, the human's velocity is given in metres per second and is rounded to the nearest whole number.

2.2.2 Action space

As described by Zamore et al. [21], a small discrete number of actions are defined, to minimize the state-action space. Other papers [5, 10] chose the action space to be a combination of discrete linear and angular velocities. However, even when the set of linear and angular velocities contains only e.g. four values each, this still results in an action space of size $4 \cdot 4 = 16$. Hence I defined only four possible actions, with which the agent should be able to avoid obstacles and reach the goal. The action set A is defined as follows

$$A = [0, 1, 2, 3] = [\texttt{FORWARD, LEFT, RIGHT, STOP}], \tag{7}$$

where the action `FORWARD` moves the agent 0.5 metres in positive x-direction, the action `LEFT` moves the agent 0.5 metres in positive y-direction, the action `RIGHT` moves the agent 0.5 in negative y-direction and lastly the `STOP` action lets the agent stay in its current position for one second. The coordinate system is shown in figure 2, where the red line represents the positive x-direction and the green line the positive y-direction.

2.2.3 Reward function

Similar to the state space, the reward function is derived from Li et al. [10], with some modifications to fit this specific problem better. The reward that is received after transitioning from state $\boldsymbol{s}_t$ to state $\boldsymbol{s}_{t+1}$ using action a_t is calculated as follows

$$R(\boldsymbol{s}_t, a_t) = \begin{cases} 1, & \text{if } \textit{goal is reached,} \\ -0.5, & \text{if } \textit{crashed into obstacle,} \\ 0.1 \cdot (d_{min} - d_c), & \text{if } d_{min} > d_c, \\ -0.01, & \text{otherwise.} \end{cases} \tag{8}$$

The variable $d_{min} = \sqrt{r_x^2 + r_y^2}$ is the euclidean distance to the closest human and d_c is a parameter that defines a comfort zone around each human. The more the agent violates a human's comfort zone, the more that state-action will be penalized with a negative reward. The default case is a negative reward of -0.01. This encourages the agent to reach the goal in the fewest possible steps, as every extra step yields a small negative reward [4].

2.2.4 Alternative problem formulation

In section 2.2 it was mentioned that the problem definition was based on Li et al. [10]. However, changes were made to significantly shrink the state-action space. Initially, the agent and human states were discretized into much narrower 0.1 metre steps, and the action space was larger as well,

as described in section 2.2.2. A training was done with that formulation and after 1000 episodes, which took approximately 20 hours to run, there were no signs of learning progress. This can be seen in figure 1, as the cumulated rewards per episode do not increase over time and hence it seems as if the agent is not learning successfully and therefore does not improve. This was probably due to the large state-action space. When there are too many state-action pairs to explore, the learning requires a lot more episodes than if the agent is able to explore the entire learning space faster. Hence the adaptions mentioned in sections 2.2.1 and 2.2.2 were made to minimize the state-action space.

Figure 1: Cumulated rewards over 1000 training episodes before state-action space was minimized.

3 Results

The simulation environment is shown in figure 2 and resembles a corridor. The agent is located in the green square at the bottom of the corridor and has to reach the green square at the top of the corridor, which is 5 metres in front of the agent. The dynamic obstacle, shown here as a robot in the pink square, represents a human that is crossing the path of the agent. It will traverse back and forth through the openings on the left- and right-hand sides of the corridor.

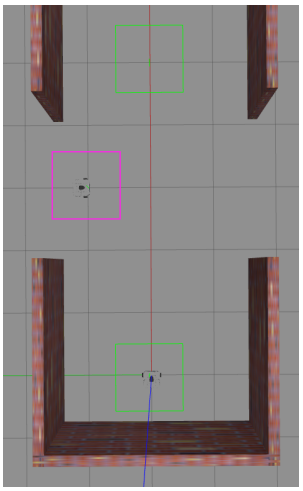

Figure 2: Environment setup in Gazebo.

The values for the hyperparameters are given in table 1.

Parameter	Value
Nr. of episodes	2100
Max. steps per episode	50
Grid / step width	$0.5m$
Comfortable distance d_c	$1.25m$
Discount factor γ	0.95
Learning rate α (decaying)	$0.25 \rightarrow 0.1$
Exploration rate ϵ (decaying)	$1.0 \rightarrow 0.1$

Table 1: Training parametrization.

3.1 Q-learning results

The training was done on a laptop with the operating system Ubuntu 20.04 and ROS version *noetic*. Running the 2100 training episodes took approximately 32 hours. The learning progress can be seen in figure 3, the cumulated rewards increase consistently over time and get close to the maximum reward value of 1.0. However, one can see that the rewards are still increasing at the end, meaning that the learning probably has not completely converged to an optimal policy yet, but due to time constraints it was not possible to train any longer. For further plots showing the training progress, the reader is kindly referred to figures 10 and 11 in the appendix, where the average expected reward is plotted for each cell in the grid world.

To evaluate the performance of this method, the simulation was run 75 times using the final Q-table from the training. The exploration rate was set to zero, meaning that the agent always chooses the

Figure 3: Cumulated training rewards.

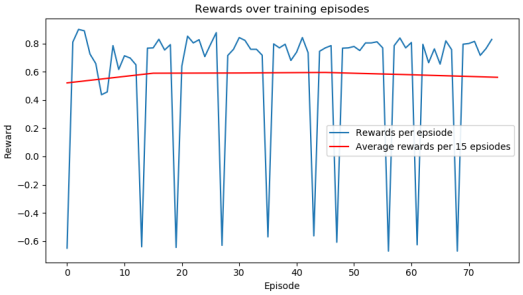

Figure 4: Cumulated execution rewards.

optimal action. The cumulated rewards for each run are shown in figure 4. One can see that the agent crashes into an obstacle every 5 to 10 runs, but the average cumulated reward stays consistently above 0.5. Out of the 75 runs, 10 runs resulted in a crash, yielding a success rate of approximately 87%. The average cumulated reward per episode was 0.575 and on average each run took 39.04 seconds. The average running time for only the successful runs was slightly longer with 40.42 seconds. The runs with the three highest rewards are listed in table 2. One can see that the best run was when the agent's path went straight to the goal without any turns and the agent only stopped once, probably waiting for the human to pass.

Reward	Duration [sec]	Actions taken
0.9	18.39	[0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0]
0.89	19.31	[0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0]
0.877	23.12	[0, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0]

Table 2: Statistics of the three best runs.

Two successful example runs, where the agent utilizes the human's motion pattern to its advantage, are shown in figures 5 and 6. Figure 5 (a) shows the agent in a position where continuing forwards would most likely result in a collision. Turning left does not make sense either, as that would be parallel to the human. Hence in (b), the agent decides to turn right and then move towards the goal, as seen in (c) and (d). This shows how the agent has learnt the movement pattern of the human and uses that knowledge to avoid the dynamic obstacle.

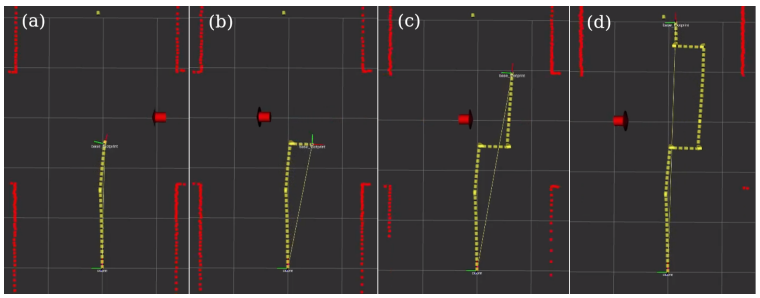

Figure 5: Agent successfully avoids human.

In figure 6, the agent realizes in (a) that the human ahead will move east, meaning that the cells that are currently occupied will be free by the time the agent arrives there. Hence in (b), the agent can simply continue forwards without stopping or turning and go to the goal as seen in (c), without getting close to the human.

A weakness of this solution is shown in figure 7, where the agent does not seem to reach the goal as fast as possible. In (a), one can see that the agent goes from start to P, then to Q, back to P, back to Q and only then only continues towards the goal. This behaviour does not seem optimal, as the agent is not close to the human at points P or Q and these unnecessary steps are penalized with negative

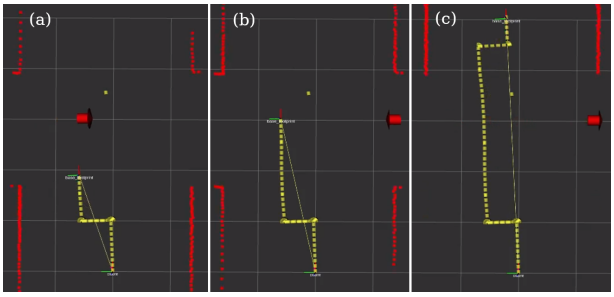

Figure 6: Agent sees that the path straight ahead is free.

rewards. Hence these must be states that the agent has not yet seen or not explored enough. This also shows the major disadvantage of this method: in unexplored states, the agent will most likely behave weirdly as the optimal policy is not known. For more examples of the agent's suboptimal behaviour, the reader is kindly referred to figures 12 and 13 in the appendix.

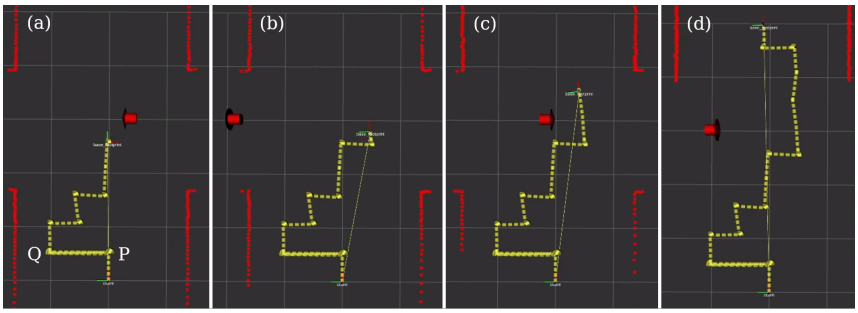

Figure 7: Execution with unnecessary actions in the beginning.

3.2 Comparison to DWA based motion planner

In order to further evaluate the RL solution, a different navigation solution was used on the same problem, utilizing the ROS `move_base`[3] package with a motion planner that uses the DWA[4] scheme for path following and obstacle avoidance as mentioned in section 1.1. The default parametrization that comes with the `turtlebot3_navigation` package[5] was used, only the obstacle avoidance was increased and the maximum linear velocity was set to the same speed as the forward velocity in the RL implementation, to make the two solutions as comparable as possible.

In total, 25 individual runs were executed. The duration was measured for each run, as well as if the agent crashed or if the goal was reached successfully. 7 out of the 25 runs were crashes, resulting in a success rate of approximately 72%, which is lower than for the RL solution. The average duration per run was 21.03 seconds and for only the successful runs the average running time was 24.25 seconds, so around 16 seconds faster than the RL solution. The three fastest non-crashed runs took 21.78, 22.52 and 22.53 seconds, which is comparable to the best RL runs. The three slowest runs however took 26.51, 27.35 and 31.11 seconds, which is significantly faster than the slowest RL runs, which took 63.30, 64.86 and 83.79 seconds. It can be concluded that the DWA solution is faster than the RL solution, but at the cost of a higher crash rate.

The reason for more frequent crashing is that the DWA planner considers a "snapshot" of the current environment and hence does not account for possible future trajectories of obstacles when planning. This can be observed in figures 8 and 9. Figure 8 shows two separate examples (a) and (b) where the DWA planner replans around the human, but in the same direction as the human is heading. This results in the agent taking a larger detour as necessary or even crashing into the human, as the DWA planner guides the agent parallel to and even slightly towards the human. Figure 9 shows a run that

[3] `http://wiki.ros.org/move_base`

[4] `http://wiki.ros.org/dwa_local_planner`

[5] `http://wiki.ros.org/turtlebot3_navigation`

resulted in a crash. As mentioned above, the DWA planner plans based on a recent snapshot of the environment, and in this example in (a), the human obstacle seems far away enough for the agent to pass by without having to detour. However, the planner does not know that the human is moving east and hence in (b) it comes to a collision shortly thereafter.

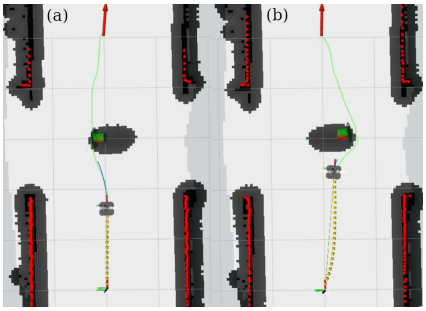

Figure 8: Suboptimal replanning around human.

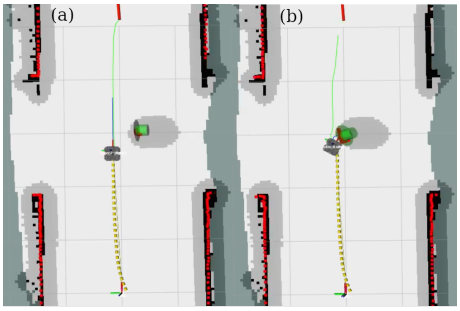

Figure 9: Agent crashes into human.

4 Discussion

Overall, the reinforcement learning navigation solution yielded the expected results. The agent was able to reach the goal and used the human's motion pattern to avoid the human smartly. Hence the collision rate of the RL solution is lower compared to using the DWA planner, as the DWA planner does not predict any future motion of obstacles. A downside to the RL method is that it is rather conservative when getting close to the human, taking large detours and hence on average being significantly slower than the DWA controller. Moreover, its generalization properties are quite poor. In a different environment, the RL method will most likely not perform well and would have to be re-trained for that specific case, whereas the DWA planner would work similarly well.

Regarding the conservativeness of the RL controller, this could be observed in the rewards of the test runs as well. Some runs had very low rewards compared to the best executions in table 2, even though the robot got to the goal quickly. For instance, one run with actions [0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0] that took 18.73 seconds got a reward of only 0.649, probably because the robot was in the pedestrian's comfort zone which was penalized. A similar case was a run with actions [0, 3, 3, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0], that took 22.14 seconds but the reward was only 0.437. To solve that, the reward shaping could be tweaked a bit more by decreasing the comfortable distance d_c or removing that penalty from the reward function completely, but re-training to get significant and comparable results takes extremely long, making reward shaping a very tedious process.

Hence another thing that could be improved in the future is the simulation environment. Since Gazebo is a real-time simulation environment, training is very slow and therefore exploring the entire state-action space and optimizing the parameters is an extremely time-consuming process. A faster and less complex two-dimensional simulator or a simulator that can be sped up would be a better choice, such as CoppeliaSim that can be accelerated up to 8 times [14].

To improve the generalization, it could be interesting to try different problem formulations. My implementation uses x-y-coordinates to describe the agent's position, meaning that the agent learns which grid cells are walls and which is the goal and hence will not perform well in other environments. Instead, it would be interesting to use for instance discrete laser scan readings in combination with the relative distances in x- and y-direction to the goal. This alternative formulation seems more general and I would expect it to perform better in new environments than my solution.

Lastly, using a more advanced deep reinforcement learning technique can be a future improvement. Deep RL is particularly useful for large state-action spaces which would take too long to explore with traditional Q-learning, as deep RL can learn continuous state spaces and possibly also continuous action spaces. In combination with a more general problem formulation, this might improve the method dramatically, especially in terms of generalization.

References

[1] Y. Baudoin and M.K. Habib. *Using Robots in Hazardous Environments: Landmine Detection, De-Mining and Other Applications*. Woodhead Publishing in mechanical engineering. Elsevier Science, 2010.

[2] R. Bogue. Robots for space exploration. *Industrial Robot*, 39(4):323–328, 2012. https://doi.org/10.1108/01439911211227872.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv eprint arXiv:1606.01540*, 2016. https://arxiv.org/abs/1606.01540.

[4] Özgür Şimşek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 95, New York, NY, USA, 2004. Association for Computing Machinery. https://doi.org/10.1145/1015330.1015353.

[5] M. Everett, Y. F. Chen, and J. P. How. Collision avoidance in pedestrian-rich environments with deep reinforcement learning. *IEEE Access*, 9:10357–10377, 2021. https://doi.org/10.1109/ACCESS.2021.3050338.

[6] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997. https://doi.org/10.1109/100.580977.

[7] Harsh Gupta, R. Srikant, and Lei Ying. Finite-time performance bounds and adaptive learning rate selection for two time-scale reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[8] Jeffrey Kane Johnson. The colliding reciprocal dance problem: A mitigation strategy with application to automotive active safety systems. In *2020 American Control Conference (ACC)*, pages 1417–1422, 2020. https://doi.org/10.23919/ACC45564.2020.9147351.

[9] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154, 2004. https://doi.org/10.1109/IROS.2004.1389727.

[10] Keyu Li, Yangxin Xu, Jiankun Wang, and Max Q.-H. Meng. SARL∗: Deep Reinforcement Learning based Human-Aware Navigation for Mobile Robot in Indoor Environments. In *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 688–694, 2019. https://doi.org/10.1109/ROBIO49542.2019.8961764.

[11] Heonyoung Lim, Yeonsik Kang, Changwhan Kim, Jongwon Kim, and Bum-Jae You. Nonlinear model predictive controller design with obstacle avoidance for a mobile robot. In *2008 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, pages 494–499, 2008. https://doi.org/10.1109/MESA.2008.4735699.

[12] Lucia Liu, Daniel Dugas, Gianluca Cesari, Roland Siegwart, and Renaud Dubé. Robot navigation in crowded environments using deep reinforcement learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5671–5677, 2020. https://doi.org/10.1109/IROS45743.2020.9341540.

[13] Christoforos Mavrogiannis, Francesca Baldini, Allan Wang, Dapeng Zhao, Pete Trautman, Aaron Steinfeld, and Jean Oh. Core challenges of social robot navigation: A survey, 2021. https://arxiv.org/abs/2103.05668.

[14] Savaş Öztürk and Ahmet Emin Kuzucuoğlu. Building a Generic Simulation Model for Analyzing the Feasibility of Multi-Robot Task Allocation (MRTA) Problems. In *Modelling and Simulation for Autonomous Systems*, pages 71–87. Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-43890-6_6.

[15] Hao Quan, Yansheng Li, and Yi Zhang. A novel mobile robot navigation method based on deep reinforcement learning. *International Journal of Advanced Robotic Systems*, 17(3), 2020. https://doi.org/10.1177/1729881420921672.

[16] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[17] S. Quinlan and O. Khatib. Elastic bands: connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, volume 2, pages 802–807, 1993. https://doi.org/10.1109/ROBOT.1993.291936.

[18] Antonio Sgorbissa and Renato Zaccaria. Planning and obstacle avoidance in mobile robotics. *Robotics Auton. Syst.*, 60:628–638, 2012. http://doi.org/10.1016/j.robot.2011.12.009.

[19] Sebastian B. Thrun. Efficient exploration in reinforcement learning. Technical report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, 1992. Technical Report CMU-CS-92-102.

[20] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, volume 8, pages 279–292, 1992. https://doi.org/10.1007/BF00992698.

[21] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. 2017. https://arxiv.org/abs/1608.05742.

Appendix

Q-value plots

Figures 10 and 11 show the expected reward for each x-y-cell in the grid, averaged over the agent's orientation, the actions and the dynamic obstacle. Halfway through the training at 1000 episodes, the agent has identified the goal and most of the walls, but is still colliding with the human sometimes. This can be seen from the slightly darker, horizontal line 3 metres ahead of the starting position, which is exactly where the human is crossing the corridor.

In figure 11, the agent has finished the training and has learnt how to avoid the human, as the horizontal line has evened out. The agent has also identified all of the walls and learnt how to get to the goal faster, as the average reward at the goal position has increased from around 0.4 to 0.8.

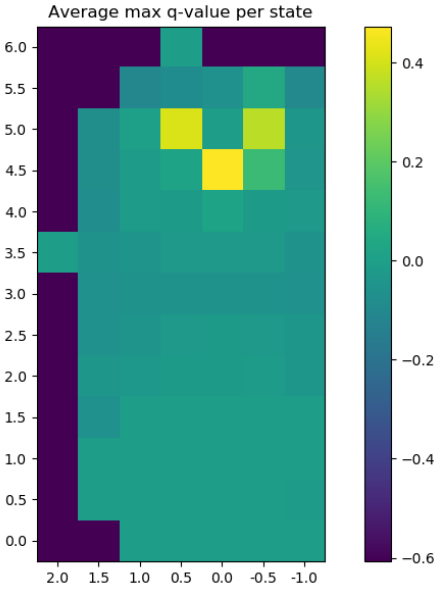

Figure 10: Q-value plot after 1000 episodes.

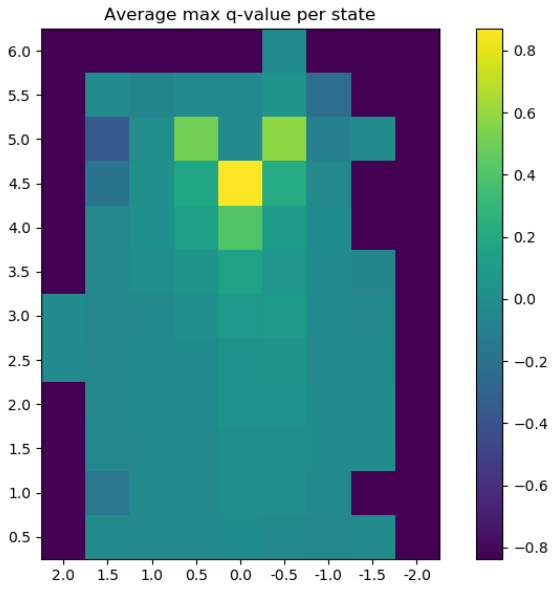

Figure 11: Q-value plot after 2100 episodes.

Suboptimal behaviour

Figures 12 and 13 show two independent runs where the agent reaches the goal, though not in the fewest possible steps. In both cases, the agent took a seemingly unnecessary step to the side and back, where maybe waiting for one time step would have had the same effect. This could be due to a lack of exploration or not enough training episodes, as these runs indicate that the agent has not completely converged to the optimal policy yet.

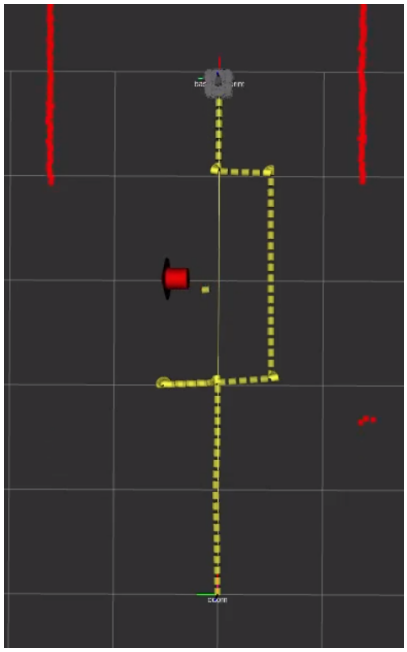

Figure 12: Unnecessary side step to the left.

Figure 13: Unnecessary side step to the right.