SoSe 2020

Technische Universität Berlin

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Telekommunikationssysteme
Fachgebiet Industry Grade Networks & Clouds

Project VIS
summer semester 2020

02 Semantic Navigation for autonomous mobile robots

Bornemann, Marvyn, 374213
Kerz, Kyra, 329289
Klaas, Tim, 372113
Lorkowski, Hannes, 376218

supervisor
Prof. Dr.-Ing. Jens Lambrecht
M. Sc. Linh Kästner

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich wesentliche Teile zur vorliegenden Arbeit beigetragen habe. Weiterhin erklären wir als Gruppe, dass keine weiteren Personen an der Anfertigung der Arbeit beteiligt waren. Die verwendeten Quellen werden am Ende des Dokuments benannt.

Bornemann, Marvyn

Kerz, Kyra

Klaas, Tim

Lorkowski, Hannes

Contents

1 Introduction

Mobil robots have gained a lot of popularity due to their broad range of applications. Their key requirement is to navigate in their environment safely and robustly. As of today there already exist many path planning algorithms for navigating successfully from A to B through dynamic environments. Those however only take spatial information about the environment into account. We propose that including semantic information about objects into the path planning algorithm can further improve navigation behaviour. Therefore we implemented a proof of concept of including semantic reasoning into an existing reinforcement learning based planning algorithm.

Most navigation systems today first compute a higher representation of their environment which is then used as the input for calculating a plan of how to move through it. The most common algorithms for this purpose is simultaneous localization and mapping (SLAM). SLAM is a process where the agent keeps track of its location while creating and updating a map of its environment with the help of sensor data. SLAM is especially powerful in static environments, but falls short in dynamically changing environments. Furthermore it is computational costly, especially when applied on larger areas. To avoid these high computational costs we integrate semantic navigation into reinforcement learning based path planning, which does not make use of a higher representation, but uses the raw sensor data directly.
As alternative we propose a deep reinforcement learning (DRL) approach for highly dynamic environments and to reduce the computational costs. To this purpose we are modelling behavioural rules or rewards based on semantic input and implementing them onto an agent. This agent will be then trained in a simulated environment the - ARENA2D framework. Our focus lies not just on the robot to navigate safely through a static environment, but also on human-robot collaboration, making our environment highly dynamical. To this purpose human behaviour corresponding to motion patterns and human to human interaction is implemented in the simulation. The agent is based on a neural network structure in our case the deep Q-network (DQN), which maps states to Q values. For each robot action the network estimates a Q value and the action with the largest Q value gets executed by the robot. For this estimation the network needs input information given by a LiDAR sensor. Additionally distances and angles to the goal, as well as related to the humans is also to be considered for the reward shaping and therefore used for the calculation of the next actions of the agent. After the agent is trained, a second agent is used for the evaluation of the trained model. Therefore the main contributions of this work are the following:

- Implementation of humans in simulation as well as detection in real world.
- Reward shaping of an DQN agent taking additional input information into account.
- Evaluation of the performance in terms of an human robot collaborated environment.

This report is structured as follows. First the objective of the project is defined and the components on how to achieve those are described in 2. Secondly the state of the art research for path planning for dynamic environments is presented in section 3. After that fundamental concepts of the applied approaches in this project are explained in section 4 and the conceptual design choices are presented in 5. In section 6 the actual implementation of our project is described in detail, divided into simulation environment, agent, reward functions and deployment. Chapter 7 show the results and Chapter 8 and 9 end with a conclusion and a future outlook respectively.

2 Objective

Goal of this project is to improve human-robot interaction in path planning with reinforcement learning. This can be summarizes in two main points:

- Find a goal while maintaining a sufficient distance to humans
- Contact between robot and human must be avoided at all times

The goal is illustrated in fig. 1.It shows the path of the robot to the goal when a person blocks the shortest way. It can be seen that the robot recognizes the person and maintains a certain distance from him.

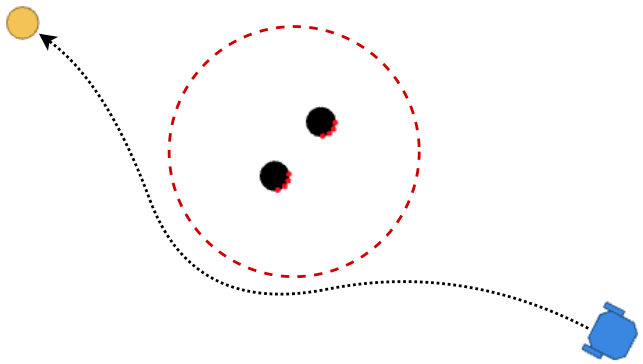


Figure 1: schematic of DQN-agent

The implementation of this aim is divided into five main components of which a detailed description can be found in Chapter 6:

1. **Integration of human behavior in simulation**
   First humans need to be integrated in the simulation. For the best possible result, it is important to adapt the behavior of the integrated objects to human behavior.

2. **Integration into the deep-reinforcement-algorithm**
   The next part is to extend the learning agent, so the neural network can learn to deal with humans. Therefor additional inputs and rewards need to be introduced.

3. **Evaluation of the new trained neural networks**
   In order to be able to compare changes in the training process, suitable evaluation parameters must be extracted from the simulation data. In addition, at the end of the project it should be clear whether the change has brought an improvement in human-robot interaction for path planning.

4. **Object detection and positioning**
   In order to provide the required semantic input to the neural network a semantic object detection system has to be integrated as well as an algorithm for calculating the corresponding positions relative to the robot.

5. **Deployment on real robot**
   In order to test the full system the trained agent as well as the object detection is implemented on a real robot(Turtlebot 3 Burger) using the Robot Operating System (ROS) infrastructure.

3 State of the Art

This chapter gives a brief overview of the state of the art in path planning algorithms.

Path planning in robotic applications already has a long history of which a good overview can be found in [8]. One of the most advanced algorithms today, showing real time capabilities and robustly producing short and smooth paths is called Timed-Elastic-Bands (TEB) [13]. The TEB algorithm formulates the path planning problem as a hypergraph structure, where nodes represent obstacles and trajectory positions, and edges between the nodes represent policies between the nodes such as keeping distance between obstacle and trajectory nodes. The hypergraph is then implemented as an optimization problem, where the policies are formulated by costfunctions. Minimizing these costfunction by adjusting the trajectory nodes leads to the desired trajectory. An approach presented in [6] further takes semantic objects into account while also using a hypergraph to formulate the path planning problem. They do this by further adding nodes representing semantic objects to the hypergraph as well as new polices in regards to those nodes.

In contrast to that the approach taken in this project makes use of an existing reinforcement based neural network, which directly computes velocity outputs based on laser scan inputs [4]. In order to extent this algorithm to also base its control output on semantic information about the environment, the input of the neural network is extended to also include semantic object positions. Semantic policies are implemented by adding further reward functions that are used to train the neural network. Similar work in this direction has been done in [2] where they also used deep reinforcement learning for their path planning approach in dynamic multi agent environments. What was special in this approach was that they do not use a specific model for other decision making agents such as pedestrians as is done in other work, but they let the neural network figure out its own model. Furthermore by making use of a long short-term memory (LSTM) cells for the network input, they managed to convert a varying amount of agents into a fixed size vector and therefore were not dependent on a fixed input size of agent inputs unlike other approaches.

In terms of object detection, the currently most promising approach is YOLO (You Only Look Once) described in [12], which uses an RGB image to compute labeled hitboxes of objects. The approach described in [9] uses YOLO to detect objects and further calculades their position using RGB-D images for a SLAM based planning approach. This paper uses YOLO as well with an own implementation of how to compute the position of objects.

4 Fundamentals

Reinforcement learning is a concept of machine learning in which an agent independently learns a strategy to maximize a reward of a reward function. Consequently the agent learned independently in which situation which action is the best. The reward function describes which reward value a specific action in a state has.
For the description of a reward learning problem are the following parameters necessary

- S set of all States
- A set of actions
- $T(s, a, s') \implies [0, 1]$ transition function which describes the probability that the agent change his state from s to s' with action a
- $R(s, a, s') \implies \mathbb{R}$ reward function which assigns a reward for the change from state s to s' with action a
- $\gamma \in [0, 1]$ Discount factor affects the length of a way to get a reward. For $\gamma = 1$ is the decision of the agent independend from the length of the way to get it. For $\gamma = 0$ the agent would always decide for the reward with the shortest way.

In a typical RL-problem the agent knows nothing about his environment. Thus, the learning algorithm can not assume knowledge about the transition function T or the reward function R. The agent must therefore move through the states independently via actions in order to experience the various transitions and their rewards and probabilities for themselves. After each transition, the agent receives an external notification of the reward.
There are a large number of different approaches to solving a reinforcement problem. In this context, the theory for the approach used later should now be explained. Many RL-algorithms are based on the direct learning of Q-Values. Thereby $|A|$ values are saved per state. As a result, the Q-Values provide two additional important informations. The highest possible reward $V(s)$ and the optimal next action $\pi(s)$.

$$V(s) = \max_{a \in \mathcal{A}} Q(s, a)[1] \tag{1}$$

$$\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s, a)[1] \tag{2}$$

Another challenge is the exploration of the environment. During the learning phase, the agent has to decide in every step whether he trusts in the learned reward and transition function or he wants to explore the space better using random decisions. There are many variations to find a compromise for this decision. The $\epsilon - greedy$ exploration is later used and will be explained now. Another hyper parameter is introduced for this: $\epsilon \in [0, 1]$. Epsilon is zero at the beginning of the training. During the training, the agent decides with a probability of $(1 - \epsilon)$ whether he will make an arbitrary decision or trust the information he has learned so far. Epsilon growth with the training time.
Another difficulty is finding a suitable function to calculate the Q-values. Particularly in complicated environments, normal approximated functions and their input parameters are not sufficient for the agent to make optimal decisions. A solution is the deep Q-learning, where neural networks calculate the Q-Values. The neural networks gets several informations about the environment

as an input. The output is $|A|$-dimensional and includes the Q-Values for the different actions. The Q-learning with neural networks can be improved with double Q-learning. The normal Q-learning (or deep Q-learning) tends to overrate the Q-values. This problem persists in many updates after it has occurred once. Double Q-learning offers a solution for this problem. Therefor two identical Q functions (or neural networks) are defined. The are called policy and target function/net. The policy net is used for the decisions of the next actions, but for updating the expected Q-Values are calculated with the target net. The target net is only synchronized to the policy net after a predefined number of steps.
The field of specific training approaches in reinforcement learning is large. Different approaches were used for training in this project. Since it was not part of the task to program the agents, the most successful approaches are only briefly explained below.

DQN

With DQN learning, a neural network is trained with previously described double q-learning. The Agent explores the environment with $\epsilon - greedy - exploration$ and saves all necessary values in a buffer. Then the policy-net is optimized with stochastic gradient decent. This shows the double-q learning, since the optimization of the policy net is carried out with the calculated q-values of a second neuronal network. This is called target network. The weights of the target net are synchronized to the weights of the policy network after a predefined number of steps. Figure 2 shows a schematic overview of the training process.

1. initialize policy net and target net
2. choose random/policy action
3. record in buffer
4. sample batch from buffer
5. optimize policy net
   - 5.1 calculate least square error target net - expected reward
   - 5.2 update policy net with SGD
6. every N steps synchronize target net
7. repeat from 2 until boundary is reached

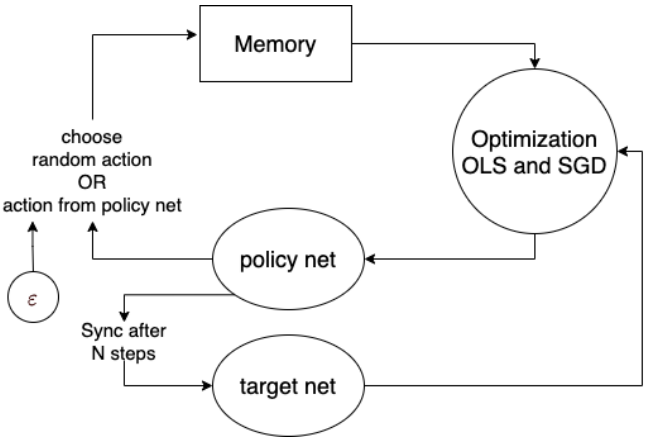


Figure 2: schematic of DQN-agent

5 Conceptual Design

This Chapter gives a brief overall overview of the project. A schematic of this is shown in Figure 3. The main component in this project is the local planner. It computes a velocity control command which will be executed by a mobile robot, in our case a TurtleBot 3. The local planner is realized by a neural network which takes in distances to obstacles, distances and angle to detected humans and the high level goal as inputs. The distances to obstacles are provided by a LIDAR sensor, which is mounted on the TurtleBot. For detecting humans a RGB-D camera is used, which is also mounted on the TurtleBot. Based on the RGB image persons are detected and their position is calculated based on the depth image. Lastly there is a top down camera mounted in the workspace which detects the robot position and the high level goal based on QR markers and from that computes the relative goal position in respect to the robot position, which is then fed to the local planner.

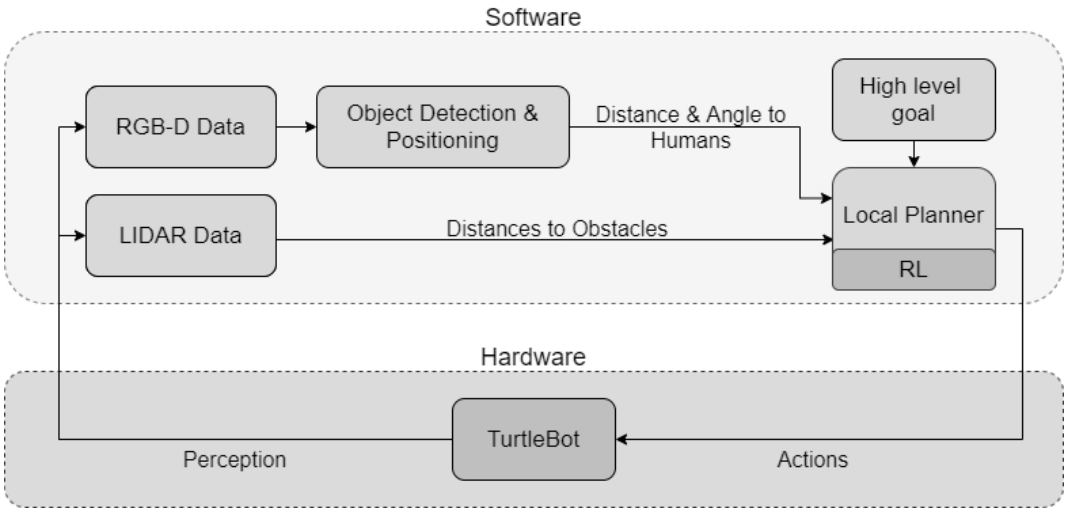


Figure 3: Overview of the overall project structure

6 Implementation

This part gives a detailed description of the implemented algorithms in the project. It is divided into the blocks of the reinforcement learning, evaluation of the trained neuronal networks and the application on a turtlebot. The reinforcement learning part includes the training process of the neural network in the simulation using newly created environments. The evaluation block describes how the training results are compared. The application part describes the installation and usage of the trained network on a turtle bot.

6.1 Reinforcement learning system

The aim of the project is to find out how a robot handles semantic information (in this case, people) if its decision is based on a neural network which is trained with reinforcement learning. The training of the agent and its integrated reinforcement algorithms is completely done in 2D simulation. For this purpose the ARENA2D framework is used, which is using the physics engine Box2D [11]. Its main purpose is to provide a highly dynamic environment in order to prevent overfitting and bridge the simulation-reality gap [4]. ARENA2D is fused by two main components - the agent and the environment of the agent.
The simulated environment is build in C++ and aims to map static and dynamic real world surroundings of industrial mobile robots into 2D level. Then the agent is supposed to find a way to its goal while using LiDAR data to avoid static and dynamic obstacles. The description of the level is divided into two parts. First the implementation and reasoning behind design choices of the level are presented in subsection 6.1.1. Secondly the dynamic obstacles or human wanderers and their influence on the environment of the agent are explained in more depth in subsection 6.1.1.
The agent for the training process is build in python3. For the extension of the agent to handle the new semantic information, additional inputs had to be added to the neural network. In addition, additional rewards have been introduced so that the robot learns a certain way of dealing with people. A detailed description can be found in section 6.1.2

6.1.1 Simulation Environment

This section describes the graphical environment for the simulation. This includes every surrounding and object the agent has to navigate in and with, which are also displayed for the user in the ARENA2D environment. For our training process the environment consists of an enclosed area within square arranged borders - the arena. Inside the arena the agent has to navigate around static and dynamic obstacles. The dynamic obstacles don't just include an implementation of placement and shape like the static obstacles, but also a movement and motion profile. For this purpose the environment creation is split in the enclosed arena and static obstacle as level classes, while the dynamic obstacles each have their own classes, that can be called inside the level. In our case we are using human wanderer as dynamic obstacles.

Level Design

The goal of the level design is to simulate the environment of the mobile robot as close as possible. However in order to determine the performance of the agent, different surroundings and therefore different levels are used for the training. Overall six different level configurations are used. An overview of the different level can be seen in figure 4. There are three different baseline setups for the static obstacles, that are explained further in the following. On top of that for each of those baseline configurations dynamic obstacles can be added as well.

- **Random**
  The first environment configuration is the level *random*. This is depicted in figure 4a. The agent tries to capture the goal while facing static objects, that are randomly placed each time the level resets. Not just the position of the obstacles is randomized, which means they can also spawn intersecting each other, but also the size which is constraint by setting a maximum and minimum value for it. The corresponding values can be set in the settings configuration of the arena, which are also displayed in table 1. This is the baseline configuration for the training of the agent, where it is supposed to learn how to avoid static obstacles with the help of LiDAR data. Figure 4b shows level *random –human*. This level configuration is the same as 4a, but dynamic obstacles are additionally placed via *–human* flag and depicted with two little circles in black. The purpose is to simulate human movement, with focus on leg motion while walking and human behaviour like stopping while passing another human for interaction. For our training purposes for human wanderers are spawned in the level. The details of the implementation are discussed in depth in subsection 6.1.1.

- **Custom**
  The next configuration is seen in figure 4c, which is the level *custom*. One of the major challenges for autonomous mobile robots is the navigation through narrow paths [5, 10]. Therefore this level introduces corridor shaped static objects on top of the random objects. Those are consisting of two single wall shaped objects. The first wall is randomly placed inside the arena, either horizontal or vertical, while the second wall is always placed in a random distance depending on the position of the first wall. Since the emphasis lies on changing environments during training in order to ensure a better generalization performance of the robot, the length and width of the corridors was randomized as well. Especially the width of the corridors is a critical factor as the agent might pick up a significant different behaviour depending on the space between the walls. The range of the width is dependent on the agent diameter, ensuring the agent is able to fit through. Therefore values between two to three time of the agents diameter are randomly chosen. The height is set randomly to a value between 0.5 and 2.5. The ratio of corridor elements to random object elements is randomized as well, with a probability of $\frac{1}{3}$ and $\frac{2}{3}$ respectively depending on the chosen value for the number of total obstacles inside the level, in this case it is set to 9. Furthermore it is ensured that the goal is not spawning in an unreachable part of the map, that is enclosed by corridors. To this purpose the bonding box of each corridor and random shaped static obstacle is calculated. Each time bonding boxes are intersecting on spawning a new placement is search for the item. In order to prevent long computation times on generating the level, a limit is set to 200 tries for positioning the obstacles without overlapping. If such a configuration isn't found in this time, the item without proper placement is dropped. This means there can be less than 9 static obstacles at the stage, which results in further randomization. Similar to this configuration is the level *custom –human* depicted in 4d. This is the same setup as in 4c, but now human wanderers are added on top of it.

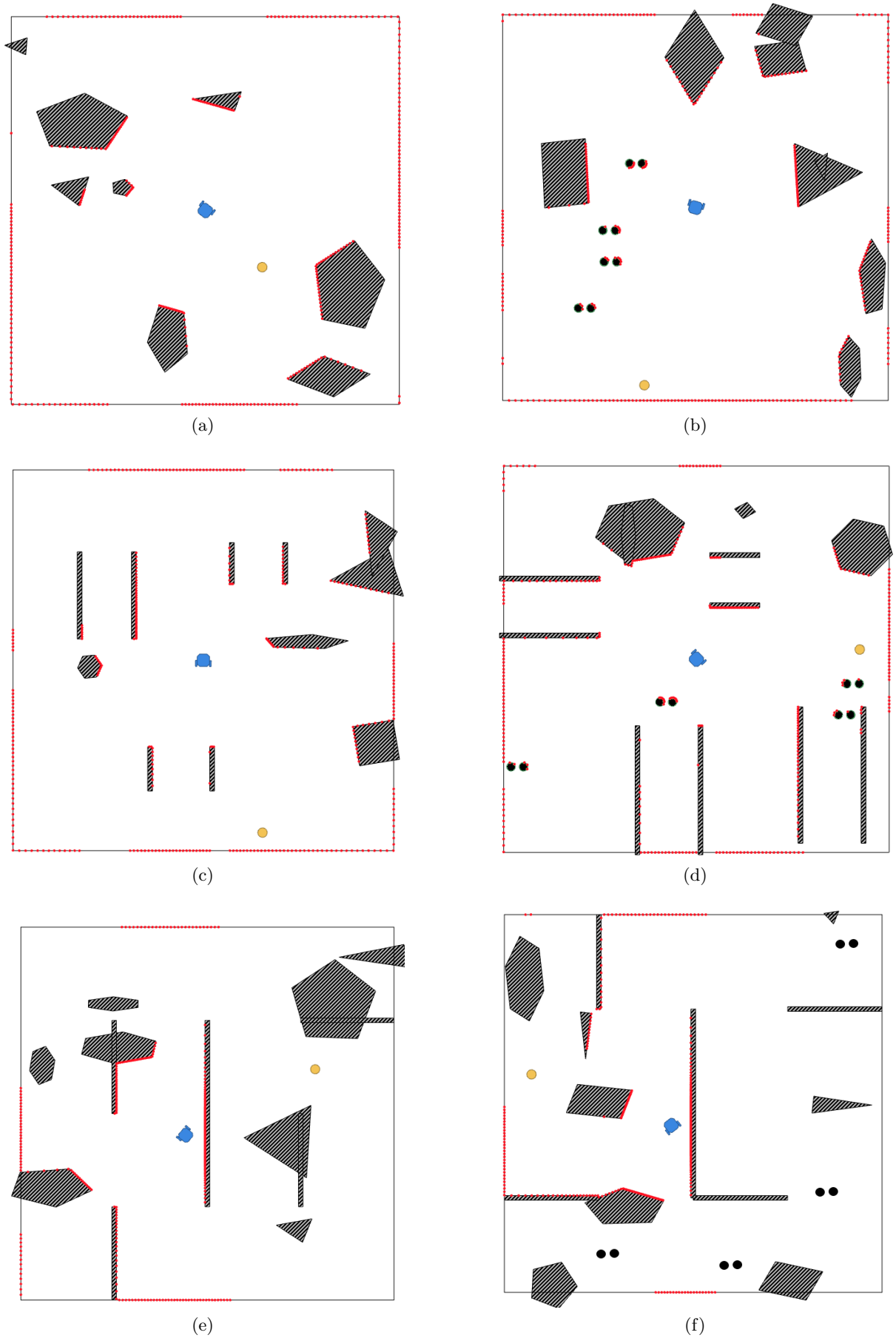
- **Maze**
  The last static configuration that is used in training is the level *maze*, which is presented in figure 4e. This setup is a bit more relaxed in terms of narrow paths, but it is still incorporating walls at the stage for mimicking e.g. office environments. There are always five wall shaped elements placed, which are using always the five same anchor points. On each anchor point a wall, with a fixed length is set, with the wall set at the center point of the arena being the biggest and the other four being shorter, but having all the same length. The four shorter wall are placed in each quadrant. In order to get the variation the anchor points are shifted by 0.5 or -0.5 and the orientation is either vertical or horizontal.

The long wall is placed at the center of the arena. In order to prevent the agent to spawn on top of the wall, its spawn point is moved to the third quadrant of the arena. Again the orientation is either horizontal or vertical, but without the shifting of the anchor point. The last configuration is the level *maze –dynamic* depicted in 4f. This is the same setup as in 4e, but now human wanderers are added on top of it. This is especially critical, if the agent is encountering a human wanderer in a corridor, because it is limiting the possible avoiding strategies of the agent and therefore making it more difficult for the agent to navigate.

Each level has parameters that can be modified in the settings before the training process. Table 1 is showing the parameters for the stage with the actual used values and an explanation of the parameters. The last four parameters are intertwined to the human wanderer and are therefore explained in the following subsection 6.1.1. The parameters max_obstacle_size, min_obstacle_size and num_obstacles are explained further above in context of the level description. Finally there is the level_size, which is set to 4 for the training, but can also be changed for exploring the behaviour of the agent in bigger or smaller operating spaces.

Parameter	Value	Explanation
level_size	4.0	width and height of default levels
max_obstacle_size	1.0	maximum diameter of static obstacles
min_obstacle_size	0.1	minimum diameter of static obstacles
num_obstacles	8	number of static obstacles
dynamic_obstacle_size	0.3	size of dynamic obstacle
num_dynamic_obstacles	4	number of dynamic obstacles in static_dynamic level
obstacle_speed	0.08	in m/s for dynamic obstacles
max_time_chatting	50.0	in ms for max human wanderer interacting time

Table 1: Parameter for level configuration



(a) (b) (c) (d) (e) (f)

Figure 4: The three base level configurations used in training, as well as with activated human wanderers

Human Wanderer

In order to make the environment more realistic to industrial workplaces, a modified version of the already existing wanderer class was created - the human wanderer. The core idea is already explained in [3], where a bipedal movement profile is implemented in order to simulate the motion of humans. The implementation of the conversion of the wanderer class to the human wanderer class consists of three core concepts - the object shape, the movement simulation and the interacting simulation, which are depicted in figure 5. First the dynamic obstacle has to be displayed by two round objects instead of one in order to simulate the legs. To this purpose Box2D accommodates several possible solutions like connecting two bodies with a joint configuration [11]. But in our case we are using only one body with two circular shapes, because this object configuration plays a role in how the movement of the human wanderer will be generated. As presented in figure 5a the diameter of the body didn't change, but instead dependent on the center point two shapes where placed on the body.

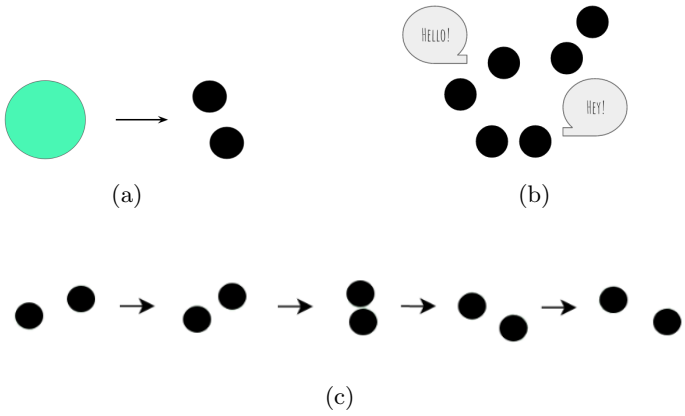


Figure 5: Core components of the human wanderer. **(a)** represents the change of shape of the wanderer to the human wanderer. **(b)** depicts three wanderers stopping near each other to simulate human interaction. **(c)** is presenting the movement sequence of the human wanderer

This choice of configuration comes into play for the component, the movement simulation. The goal is to model the movement of human legs and map it onto the object body. Ideally both circles or legs should be able to be able to move dependently of each other. Using two bodies did not work, as the bodies wouldn't move synchronous to each other and were independently influenced by Box2d collisions physics. On collision with a wall or obstacle the body is changing its direction. This results in drifting apart of the two bodies. Using a pre-implemented joint of Box2d would have prevent the drifting apart, but the movement would still be asynchronous after collision. Therefore the aforementioned body configuration was chosen. The motion was constructed by destroying and recreating the body in each update step of the simulation. The details for creating the motion are shown in algorithm 1. For each simulation step the update() function of the human wanderer is called. First the list of all shapes, that belong to the body are called and destroyed. The new fixtures are generated with the help of a sine function. Upon generation of a circle shaped fixture, a position with x and y coordinate need to be set. Because both fixtures are dependent on the center point of the body, the x value is set fixed to 0.05 for one leg and -0.05 for the other. This is supposed to illustrate two parallel standing legs. The y value on the other hand is dynamic. A counting variable is used to determine the y position. The float modulo operation is used on the counter with a modulo operator of 2π in order to match the domain of the sine function. Each time the update() function is called, the counter is increased by a changeable frequency, in our case it is set to 0.1. This is indicating the amount

of change of the y value making the movement more smooth. The y value is now plugged into a sine function. This means each shape or leg of the body is separately moved, with a value between -1 and 1. In order to generate an asynchronous diagonal motion, the the y value of second circle is also getting added π, making the output of the sine wave for each shape differ in their sign. But using the output of the sine as step size would be too large in terms of the arena environment and therefore has to be normalized to the arena size. In our case this is done by multiplying with a normalization factor of 0.1. Theoretically the counter could grow indefinitely as input for the modulo operation, but in terms of a computer system a buffer overflow needs to be prevented and therefore each 100π update() calls, the counter is reset to zero. After calculating the y position value, the new shapes of the body are generated at the corresponding positions. The new positions of the body is resulting into a sliding motion mimicking more leg movement instead of steps like in [3] presented. Figure 5c is showing 5 example steps of the motion swinging one leg from back to forth than stopping in the middle and than the other leg is used.
Another feature of the human wanderer is the possibility to interact with other human wanderers. As stated in the beginning, the goal is to mimic human behaviour like stopping and chatting upon meeting each other. This helps to train for uncertainty in the environment, as the movements and stopping pattern of humans in reality can be quite erratic and hard for a robot to predict. In algorithm 2 the pseudo code for implementing this feature can be observed. For each simulation step the function update() is called, in order to calculate the next movements of the human wanderer. A boolean named chat_flag is given as function parameter. If a human wanderer is in the range of another human wanderer their class attribute chat_flag will be set to true, indicating they are entering the interaction mode. During this time the linear and angular velocity are set to 0. The interaction time is set randomly between one and 10 seconds for each simulation step in as global variable in the human wanderer class, which is indicated by the threshold in algorithm 2. This means each human wanderer is stopping for a different amount of time, until they move on. In order to prevent, that they start interacting again the moment they are in range of each other and therefore preventing moving on, a reset counter is added. This reset counter prevents the human wanderers to enter a new interaction phase for a certain amount of time, after they quit the last one where they had participated in. This is also indicated by a threshold, which is randomly chosen between the values - the reset_threshold.
Finally the human wanderer don't just stop while interacting with each other. For further modelling of human movement behaviour, it also stops with a given randomly with a given stop_rate as threshold, in this case it is 0.2. After stopping, no matter if its through interaction, collision or the stop_rate, the orientation and velocity are changing randomly between values of 0 and 360 degree and the half of the maximal velocity and the maximal velocity plus and offset of its half respectively.

Algorithm 1 Human Wanderer Motion

```
for each update() do
    old_shapes = body->getShapeList()
    body->destroy(old_shapes)

    counter = 0
    step_frequency = 0.1
    x = 0.05
    y = fmod(counter, 2π)
    circle1_position = (sin(y)·0.1, x)
    circle2_position = (sin(y + π)·0.1, -x)

    if counter > 100π then
        counter = 0
    else
        counter = counter + step_frequency
    end if

    addToBody(circle1, circle2)
    updateVelocity()
end for
```

Algorithm 2 Human Wanderer Chatting

```
for each update(chat_flag) do
    if chat_flag == True then
        if chat_time < threshold then
            chat_time++
            Linear Velocity = 0
            Angular Velocity = 0
            Return
        end if
    else
        if chat_time ≥ threshold then
            reset++
            if reset > reset_threshold then
                chat_time = 0
            end if
        else
            reset, chat_time = 0
        end if
    end if
    updateVelocity()
end for
```

6.1.2 Learning environment

This section is a detailed description of the training process. It is divided in three parts: The fist part is the description of the training process itself. The next part describes the interface between the training agent and the simulation environment. The last part explains the calculation of the rewards for the agent.

Training

In reinforcement learning, a function is trained to estimate reward values for certain actions (in this case seven different movements of the robot) from various parameters of the environment. This learning process is carried out by a so-called agent. The agent does an action and calculates a reward with the inputs available to him. In the next step, he gets the actual reward value. With this information he can then optimize his function. The environment of an industrial hall with objects and people is too complicated to map the inputs to the rewards with a normal function. Therefore, the function is replaced by a deep neural network. The general structure of the neuronal network can be seen in fig.6. The input features are 360 samples from the lidar, the distance and the angle to the goal as well as the distance and the angle to all humans. The output is the Q-value which consist of the estimated reward value for every possible action as well as the estimated next best action (compare eq.1). For the integration of the human-robot interaction the original network is extended by the red marked inputs.

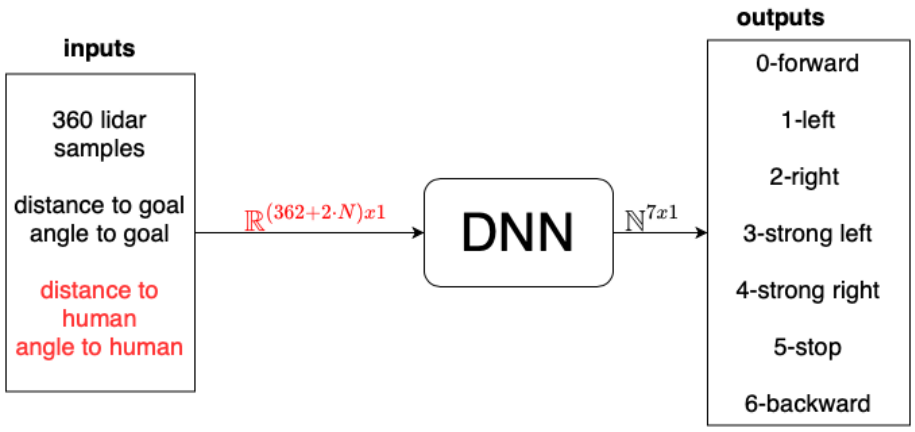


Figure 6: overview of the neuronal network

For training this network with reinforcement learning different agents are used. These are: DQN, lstm, gru, a3c.

Since it was not part of the project to implement the agents, the specific implementation will not be explained. As above explained, it was only necessary to adapt the net structure. The theory of the agents with the best training's results as well as a the reinforcement training theory are shortly explained in sec.4.

Additional Input

As described before the agent is provided with additional input data of the humans. This paragraph explains how these are calculated and hand over to the agent.

First of all the distances and angle to all humans relative to the robot are calculated as depicted in fig. 7. The distance between the robot and a human is the euclidean norm of the connecting vector $v_{robot2human}$ minus the radius of the robot and human. As formula:

$$v_{robot2human} = \begin{pmatrix} x_{robot} \\ y_{robot} \end{pmatrix} - \begin{pmatrix} x_{human} \\ y_{human} \end{pmatrix}$$

$$d = \|v_{robot2human}\|_2 - r_{robot} - r_{human}$$

The angle between the facing direction of the robot and the human should be negative up to $-180°$ if the human is right of the robot and should be positive up to $180°$ if the human is left. If the human is in front of the robot the angle should be $0°$. This is done by calculating first the angle using the scalar product and then allocating the sign using the cross product.

Calculation of the angle using the scalar product:

$$\alpha = \arccos\left(\langle \frac{v_{facing}}{\|v_{facing}\|_2}, \frac{v_{robot2human}}{\|v_{robot2human}\|_2} \rangle\right)$$

Important notice: Because of some rounding issues while calculating with the computer, it can happen that the scalar product exit the boundaries of -1 and 1 of the arccos()-function. Therefore a check to handle this issue is implemented.

Cross-product:

$$\begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix} = \begin{pmatrix} v_{facing} \\ 0 \end{pmatrix} \times \begin{pmatrix} v_{robot2human} \\ 0 \end{pmatrix}$$

Allocate the sign:

$$angle = \begin{cases} -\alpha, & \text{if } z < 0 \\ \alpha, & \text{otherwise} \end{cases}$$

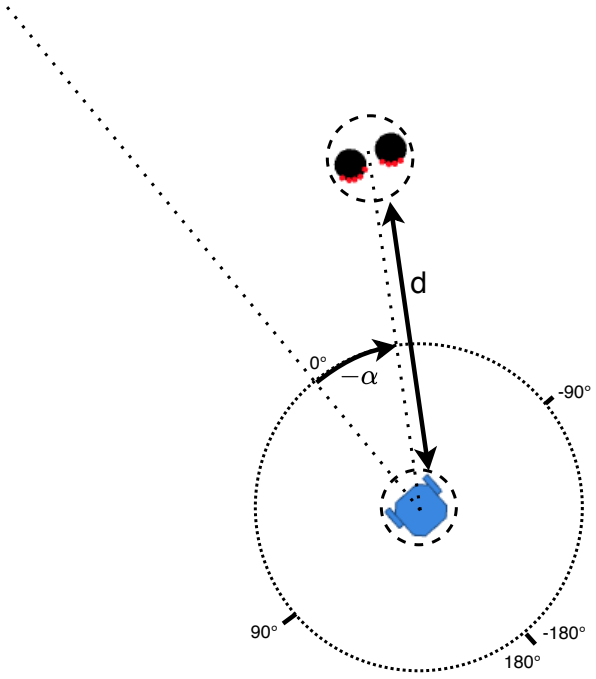


Figure 7: Calculation of the distance and angle to a human.

Before the distances and angles to humans can be passed as additional input to the agent the humans are sorted from the smallest distance to the largest. This allows to consistently forward the data of the human with the shortest distance at first and the following human data in ascending order. Furthermore a fixed number of humans which can be given to the agent needs to be defined. This is necessary, because the number of inputs for the neural network is fixed. Thus independently of the real number of observed humans in the room, a constant number of distances and angles of humans need to be provided to the agent. The maximum number of humans the agent should be possible to observe is defined by the variable `num_obs_humans` in the *training* section of the *settings.st* file.
A second aspect to consider before handing the inputs to the agent is the camera angle. The trained network is later used on a robot in a real environment. For the human detection the robot has a camera with an usual angle of 90 degrees available. Thus the space where the real robot can detect humans and provide the additional information about the distance and the angle is limited. This also needs to be implemented in the simulated human detection.
After the humans are sorted the ones, which are inside the camera view, are forwarded to the agent as additional input. With the variable `camera_angle` in the *robot* section of the *settings.st* file the angle of the camera view can be set. If the angle of the camera is set, one can simulate all the inputs the robot would get in a real environment.
This two aspects creates three possible situation to handle: This creates three different possible situations:

1. Number of observed humans in the camera angle is smaller then the number of humans to provide to the agent: In this case the angle of the missing humans is set to zero degree

and the distance is set to $2 \cdot level\_size$. The value of the distance is chosen such that it don't influence the decision of the neural network.

2. Number of observed humans in the camera angle is larger then the number of humans to provide to the agent: In this case the nearest humans to the robot are provided to the agent.

3. Number of observed humans in the camera angle is equal then the number of humans to provide to the agent: In this trivial case all observed humans are provided to the agent.

To clarify the dealing of the additional inputs fig.8 illustrates an example: In the room are three humans. The number of humans the agent should observe (`num_obs_humans`) is also three. The camera angle of the robot is set to 90 degree. Only two of the three humans are located inside the camera angle. Thus the robot can only observe this two humans. The number of humans that the robot can observe is smaller then the number of humans to provide to the agent, which is the above explained situation one. So the angle and the distance of the third missing human are set to zero degree and twice the level size.

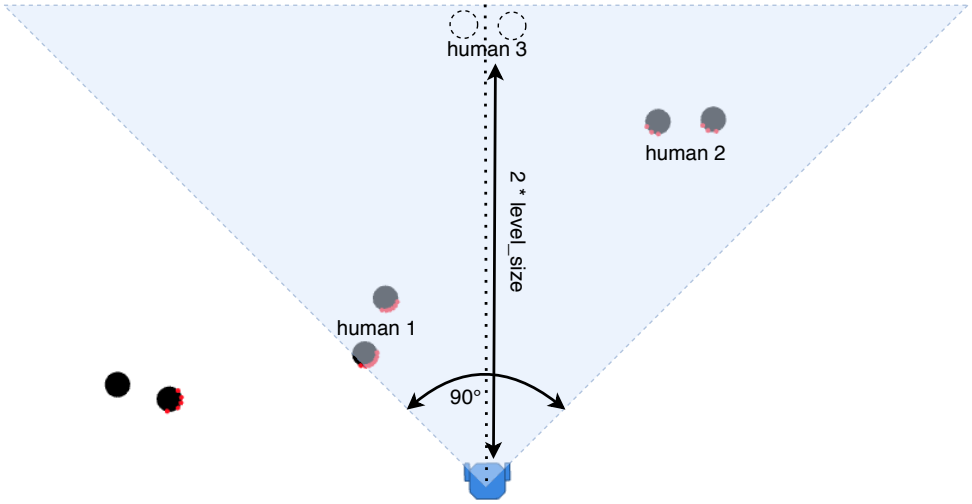


Figure 8: Observation of humans

Rewards

The agent also needs the true reward to optimize the calculation of the estimated reward of the DNN. The calculation of this true reward is really crucial. The way how this rewad is calculated influences the trained behavior of the robot most. This paragraph explains how the calculation is done.

The reward, except the *moving to human reward* is calculated in a discrete way. This means that there are different rewards for certain situations. These rewards are defined as a fixed number before the start of the training. Depending on the situation, the total reward is the sum of all incoming individual rewards. Tab. 2 shows all individual rewards with a short description. The red marked rewards are the new quantities which are introduced to improve the human-robot interaction in path planning. These are a reward if the robot hits a human as well as a distance changed reward for decreasing or increasing the distance between the robot and a human. The saftey distance human is not a reward. This variable will be explained later in this paragraph.

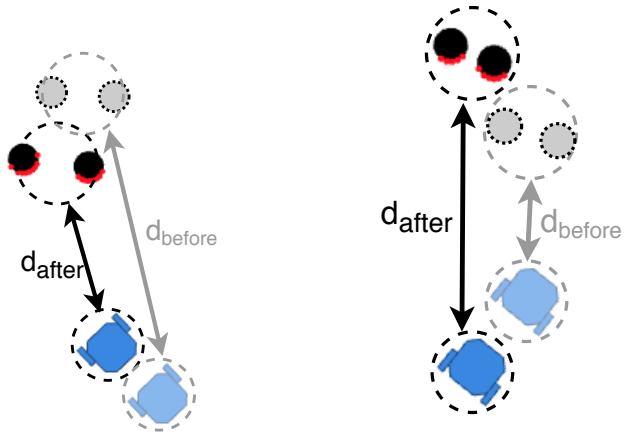
Parameter	Value	Explanation
reward_goal	100	reward if robot reaches the goal
reward_towards_goal	0,1	reward if the distances robot-goal decrease
reward_away_from_goal	-0.2	reward if the distances robot-goal increase
reward_hit	-100	reward if robot hit an obstacle
reward_time_out	0	reward if robot run in time out (not used)
reward_human	-100	reward if robot hit a human
safety_distance_human	0,8	all human rewards counts only if robot-human distance is within safety distance
reward_distance_to_human_decreased	-2,0	reward if the distances robot-human decrease
reward_distance_to_human_increased	2,0	reward if the distances robot-human increase

Table 2: Different rewards

Now the new introduced rewards will be explained in detail. The most important behavior of the robot, which should be trained, is that he never comes into contact with a person. Therefor the human hit reward is introduced. This is a very high value which will be subtracted from the total reward if the robot hits a human.
The second behavior to train is that the robot maintain a certain distance to all humans while driving. This behavior should be influenced by the distance changed reward. As already explained, the distance between the robot and every human is tracked. The distance changed reward is calculated for every distance between the robot and the human individually. If the distance decreased, a negative reward is added to the total reward. This can be formulated in the following way:

$$\text{Reward} \mathrel{+}= \text{reward\_distance\_to\_human\_decreased}$$

If it increased a positive reward is added to the total reward. The distance changed reward is illustrated by the fig.9 below.



(a) Reward if distance to humans decreased.

(b) Reward if distance to humans increased.

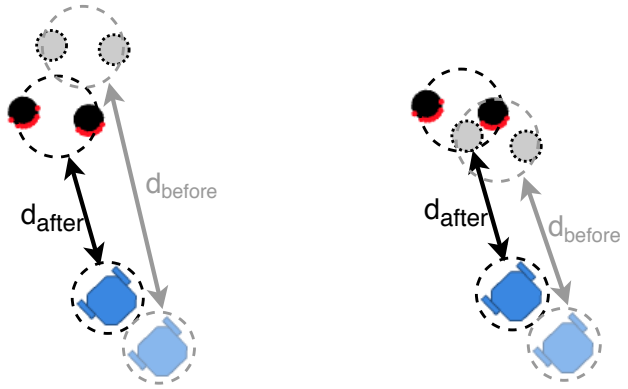
Figure 9: Rewards if distance to humans changed.

The problem with a discrete calculated moving reward is that it may be difficult for the robot to learn the moving behavior of humans in this way. Therefore, a second variant of the calculation of the distance changed reward is introduced. With this variant, the distance changed

reward value is calculated using a linear relationship with the change in distance. This can be mathematical formulated in the following formula:

$$\text{Reward+} = \text{reward\_distance\_to\_human\_decreased} \cdot (d_{before} - d_{after})$$

For example: If a human moves quick towards the robot and the distance between them decreases fast, the reward punishment is higher compared to a slower decreasing distances. This example is illustrated by the figure 10 below.



(a) Distance to human decreased a lot.

(b) Distance to human decreased slightly.

Figure 10: "Constant reward" vs "linearly depending on distance change". The distance change of $(d_{before} - d_{after})$ is in a) larger than in b).

The distance changed reward is only applied if the distance to the human is smaller than a predefined safety distance. This distance is introduced to avoid that the behavior of the robot is influenced by humans which are so far away that the robot can move without taking the risk of getting too close to a person. How to deal with this distance is illustrated in fig.11.

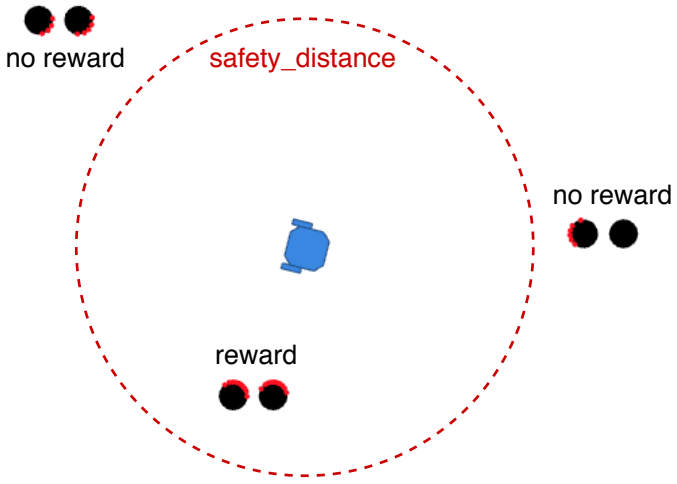


Figure 11: The distance changed reward is only applied if the distance to the human after a robot action is smaller than the safety distance.

The introduction of the safety distances creates to different cases for the calculation of the distance changed reward:

- case 1: no reward is added, because the distance after an action is larger than the safety distance even if the distance before was smaller.

- case 2: reward is added, because the distance after an action is smaller than the safety distance even if the distance before was larger.

The last paragraph discusses for which humans the reward should be calculated. As already explained in the later deployed application the robot has a limit area where he is able to observe humans. He only has concrete information about the humans which are in this area. Anyway it may improve the success of the robot if he gets rewards for all humans in the room, independently from the ability to observe them. This has two main reasons:

1. The distance changed reward is only used for humans, that were observed by the robot before he executed an action. This is explained by the fact that the distances of the same human before and after a robot action is necessary to calculate this reward. The agent chooses his action based on the observations before this action. So the agent should only get a reward for the humans he observed before an action. An illustration of an example can be seen in fig.12. It shows that the distance change reward is only calculated for the human which was in the camera angle before the robot executed his action.
   This creates two more cases for the distance change reward:
   - case 3: reward is added, because the human was observed by the robot before his action even if the human isn't observed any more after the action.
   - case 4: no reward is added, because the human wasn't observed before the robot action even if the human is than observed after this action.

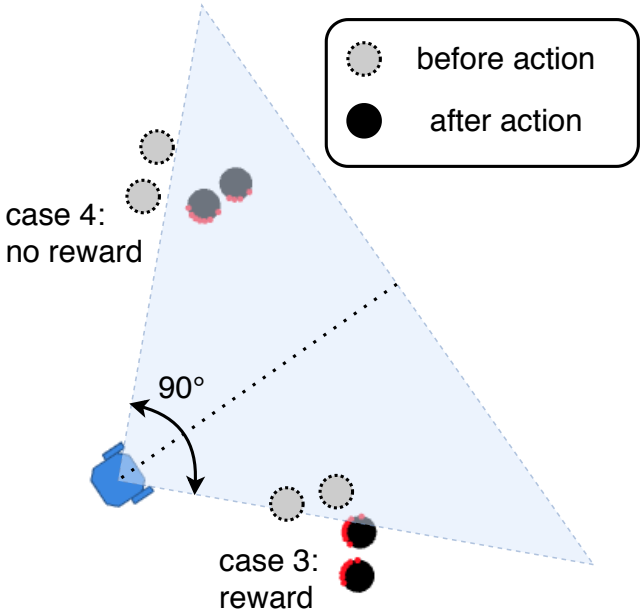


Figure 12: The distance changed reward is only applied for humans, that were observed by the robot before he executed an action.

2. If a reward is calculated for all humans independently from the ability of observation, the neural network may also learn semantic behavior based on the lidar data.

Thus, the reward calculation differs according to two basic settings: The first is for which humans the reward counts and the second is distinction between a constant calculated distances

change reward and a linear dependent distances change reward. This results in four different reward functions which can be chosen before starting a training session. An overview can be found in tab.3

RF	Human observation	Distance change reward
1	only observed humans in camera angle	constant
2	all humans in room	constant
3	all humans in room	linear dependent
4	only observed humans in camera angle	linear dependent

Table 3: Different reward-functions (RF)

6.2 Evaluation

The purpose of the evaluation is to make a statement about the performance of the trained neural network regarding the added rewards and inputs. The most important information about the network is how safely the robot keeps his distance from humans and how certain he still finds the target. There are furthermore much more interesting cite information about the performance. The evaluation process as well as the extracted features will be explained in the following.
To evaluate the trained neural network a second agent for testing was implemented. This agent passes through a predefined number of episodes. The default number is 10000. In comparison to the training agents, this agent only decides the next action based on the decision of the neural network. The hole exploration and optimisation is of course skipped. During the test simulation, various variables are saved in a csv-file for later evaluation purposes. The variables are the following:

- The episode number
- A string how the episode has ended (*human, wall, time, goal*)
- The distance and angle to the goal
- The (x,y) position and (x,y) direction of the robot
- The distance between the robot and every human

The csv-file is processed in a separate python file to extract features to make a statement about the performance of the network. The following paragraphs explains the different features.
The first features are four different values which make a statement about how the episode has ended. It differs in four sizes success rate, human hit rate, wall hit rate and timeout rate. The features includes the relative frequency of how often the episode had this specific ending. Also the variance for these four sizes is estimated. The features can be found in a text file. A more precise behavior of the four variables can be seen in a plot by showing the courses. Therefore the value of the sizes per 100 episodes is shown.
The next feature is a matrix of distances between the robot and all observed humans for every action over the hole test simulation. The distances are sorted in bins and shown in a histogram. The number of distances in a bin is normalized to the number of total recorded distances over the hole test. This plot can be used to estimate the effect of the human rewards on maintaining a desired safety distance between humans and the robot. Furthermore the relative number of distances smaller than the safety-distance (called safety distance rate) is printed in a text file.
The human hit rate, the success rate and the safety distance rate are the most important features for the evaluation of this project. They are good comparable and give a precise and easy statement about the achievement of the main goal (compare 2).
In order to judge the straightness of the robot's path, the next feature evaluated the chosen

actions. Therefor the percentage of occurrence of the various actions is determined over the test phase. The result can be seen in a pie chart. For straight path planning, the main action should be forward. Backward should show the slightest upside. The remaining actions should be roughly evenly distributed. If an action that drives to the left or right has a strong occurrence, this is an indication that the robot has driven in circles. It also can happen that the robot drives mainly backwards. Obviously this also can be seen if the the backward action has a strong occurrence.
The last feature is a subtraction between the distance that the robot needed to reach the goal and the direct distance from the starting point to the goal. This feature is calculated for each episode in which the robot has reached the goal. The resulting vector is shown in a box-plot. This feature makes also a statement about the straightness of the robot path. A low number means that the robot drives straighter. It can also be used to compare how much time the robot needs to reach the goal. This is especially interesting to see how great the loss of efficiency is in order to maintain a safe distance from humans. The distance vector for the real distances is calculated in two steps. First a two point moving window over the points in the episode calculate the distance the robot made in every action. In a second step, this vector is summed up.

$$distance_{real} = \sum_{i=0}^{N-1} \left|\left| \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} \right|\right|_2 ; \text{N=length(points of episode)}$$

6.3 Deployment

This section describes in detail the hardware that was used in this project as well as the necessary software implementations. The software was developed and tested succesfully, however due to time reasons could not be tested on the actual robot.

6.3.1 Hardware

The robot used in this project is the TurtleBot3 Burger by Robotis Inc.. It features a 360 LIDAR sensor, an onboard computer (Raspberry Pi 3) for computation, another circuit board for power management and motor control, a LiPo Battery and a differential drive consisting of two motorized wheels and a ball caster. Additionally an Intel Realsense D435i camera is mounted on the Turtlebot and connected to the onboard computer for image detection.

Apart for the robot there is an ordinary webcam mounted on the sealing of the workspace, which is used for detection of the robot position as well as the high level goal. And lastly there is a remote PC, which is used for resource intensive computations such as the local planner and the object detection. Alternatively a stronger on board computer such as the NVIDIA Jetson could be deployed on the TurtleBot in order to do the person detection on board and thefore avoid a lot of data transfer over the network.

6.3.2 ROS

The software framework used for this project is the robot operating system (ROS). In simple words ROS consists of different nodes that perform certain operations and communicate via topic message system. In Figure 13 an overview of the implemented ROS architecture is shown.

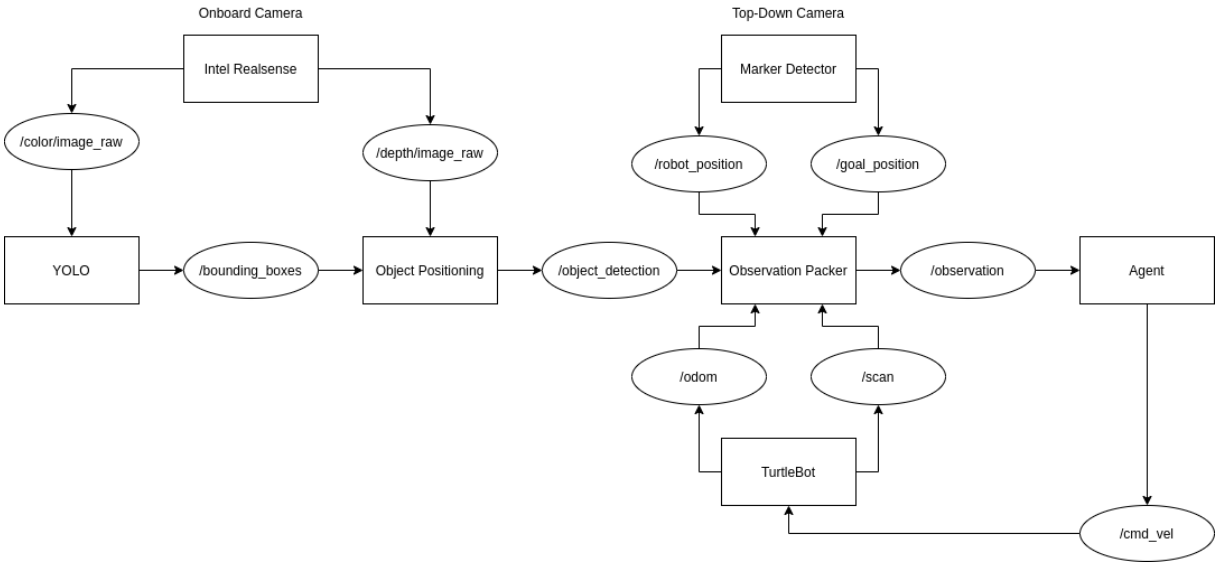


Figure 13: Overview of the ROS architecture. Squares represent nodes and ellipses represent topics.

Form a previous project at this chair there already existed the marker detector node, which detects the QR markers from the webcam image, the observation packer node, which republishes all required sensor information under one topic, and the agent node, which is the neural network based local planner. The TurtleBot nodes are provided by the manufacturer. In this project the person detection system was added as well as an newly trained agent with additional inputs and the observation topic was extended to also include the person detection data.

For the person detection and positioning 3 different nodes have been deployed. First there is the camera driver node provided by Intel, which reads out the camera and republishes the camera data as a ROS topic, one for the RGB data and one for the depth data. Next there is the object detection node, which is a ROS wrapper for the object detection framework called YOLO (You only look once)[12] which takes in the RGB image and outputs labeled bounding-boxes that corresponds to objects within the image. Lastly an object positioning node was implemented that takes in the bounding-boxes provided by YOLO as well as the depth image. From here two basic methods have been implemented for a proof of concept.

1. Distance to the object is computed as the average over the center points of the bounding-box.

2. Minimum distance within the bounding-box is taken (worst case approach).

Lastly in order to compute the angle based on the centerpixel of the bounding-box the following formula is used:

$$Angle = \left(Centerpixel - \frac{Imagewidth}{2} \right) \times \frac{FieldofView}{Imagewidth} \tag{3}$$

The calculated distance and angle to each person in the image is then send to the observation packer in order to include the information into the observation that is send to the local planner agent.

A ”How to install” guide can be found in the appendix Section 10.1.

7 Results

In this section the results of different training sessions and their tests are shown. Every training session is based on the standard settings as you can find them in appendix 10.2.1. Only the important settings which change often during different training sessions and tests are highlighted each time.

7.1 Test of DQN agent on level random

The main part of testing has the purpose to find the quality of the human rewards and the influence of the additional inputs. In all of these tests the DQN agent was trained on the *random* level. The parameters for the DQN agent were the same for each training session as they are shown in appendix 10.2.2.

7.1.1 Base case

At the base case the DQN agent was trained on the *random* level without humans as it was already done in the project of [4], what was the base of our project. Additionally this trained agent is tested with our new proposed evaluation methods (see sec. 6.2). In tab. 4 the results of the tests on the *random* level without and with humans are shown.

Test number	1	2
Test level	random	random –human
Success rate [%]	94,4	90,2
Human hit rate [%]	-	2,2
Wall hit rate [%]	1,9	5,3
Timeout rate [%]	3,7	2,3
Safety distance rate [%]	-	20,7

Table 4: Results of base case: The DQN agent was trained without humans on the random level, but tested with and without humans.

Test number 1 shows the results of the DQN agent trained and tested on the *random* level **without** humans. In contrast to the success rate calculated over 100 episodes during a training session is our success rate for testing calculated over 10000 episodes. The difference is highlighted by the following: As the success rate was 100% over 100 episodes a view times during the training session by chance and as the best agent gets picked based on the best success rate over 100 episodes, the result of the training session would be proposed with a 100% success rate of the agent. Obviously this would be misleading as the success rate over 10000 episodes is only $94,4\%$ and the 100% success rate over 100 episodes were just a lucky pick of 100 episodes out of these 10000.

Test number 2 on the other hand was trained **without** humans, but is tested **with** humans. It is important to notice that the agent, trained just to avoid crashing into walls, is able to avoid hitting humans up to $2,2\%$ on average. As the agent learned to avoid walls based on the laser-scan data, he treats the laser-scan data of the human legs as they were walls. Compared to the test number 1 the wall hit rate rose from $1,9\%$ to $5,3\%$ maybe caused by situations where the robot tried to avoid hitting a human and then crashed into a wall. The lower time out rate could be explained by situations where the robot is in a deadlock and only the change of the environment due to dynamic moving humans helped him to change his decision and finally find the goal.

Especially the result of test number 2 with humans can be used to compare with other training results and to find out, how well the additional rewards for humans are working. The main focus

should therefore lie on the human hit rate and the safety distance rate, because they are both a good indicator of the quality of human rewards.

7.1.2 Human hit reward

In the next parts the DQN agent was trained on the *random –human* level, i.e. the random level with humans. Notice that from now on each test number belongs to a test of one separate training session with a special settings configuration.

First of all two different values for the human hit reward, called `reward_human` in the settings file, are tested. The results are shown in the following tab. 5. Notice, that in the last column the base case (= test number 2) is shown for comparison.

Test number	3	4	2
reward_human	0	-100	-
Success rate [%]	**93,9**	89,9	90,2
Human hit rate [%]	**1,2**	**1,2**	2,2
Wall hit rate [%]	**2,5**	3,6	5,3
Timeout rate [%]	2,5	5,3	**2,3**
Safety distance rate [%]	21,6	**20,5**	20,7

Table 5: Results of human hit reward

At the first glance it seems, that the human hit reward 0 is better than the human hit reward -100, because it has a better success rate of 4 percent points. And this could really be the case, but it could also just happened by chance as later shown with different results of repeated training sessions in sec. 7.1.4. The question, if it happened by chance or not, couldn't be answered satisfactorily and may be worth to answer by making more tests in the future. But all the presented tests in the next parts were performed with `reward_human` $= -100$.

Additionally one can argue that the difference in the success rate is only caused by the wall hit rate and time out rate. That the human hit rate stays the same for both values of the human hit reward could emphasise the argument, that it makes no difference if the `reward_human` is 0 or -100. But for both reward values the human hit rate is smaller compared to the base case (test number 2). This can be explained by the fact, that the agent was this time trained **with** humans. Also notice, that the safety distance rate for all three tests is nearly equal at around 21%. This leads to the conclusion, that the human hit reward and the training with humans doesn't influence the safety distance rate, i.e. on average the humans are equally frequent inside the safety distance of the robot.

7.1.3 Human distance changed reward

Here the rewards for decreasing and increasing the distance to the humans are added in the training sessions. The purpose of this part is to find out, how large one should choose these rewards. This is done by comparing different reward values on the basis of 4 different configurations of additional inputs to the agent.

(i) only laser-scan data, but rewards for all humans inside the level.

First of all the different reward values for moving towards/away from humans are compared on the basis of no additional inputs (`num_obs_humans` $= 0$), i.e. the agent gets only the laser-scan data and the distance and angle to the goal as input. These rewards are calculated for all humans inside the level. As described in tab. 3 this applies only for reward function 2 and 3. In the following tab. 6 two values for reward function 2 (constant reward) and three values for reward function 3 (linear dependent reward) are compared with the base case (test number 2).

Test number	5	6	7	8	9	2
reward_function	2	2	3	3	3	-
reward_distance_to_human_decreased	-0,3	-2	-10	-50	-100	-
reward_distance_to_human_increased	0,3	2	10	50	100	-
Success rate [%]	87,1	80,5	**94,3**	92,5	84,0	90,2
Human hit rate [%]	3,1	2,4	**0,9**	1,0	1,0	2,2
Wall hit rate [%]	5,6	3,4	**2,1**	2,5	2,9	5,3
Timeout rate [%]	4,3	13,8	2,7	4,1	12,1	**2,3**
Safety distance rate [%]	19,4	20,6	21,1	19,1	**15,5**	20,7

Table 6: Results for (i) only laser-scan data, but rewards for all humans inside the level. Here two values for reward function 2 (constant reward) and three values for reward function 3 (linear dependent reward) are compared with the base case (test number 2). Settings: `num_obs_humans` = 0;

Comparing the results of test number 5 and 6 (the two reward values for the constant reward function 2) it seems that test number 5 is better since it has a larger success rate than test number 6. But compared to test number 2, the base case, have both test number 5 and 6 worse results, because their success rate is smaller and the human hit rate and time out rate is larger. Also the safety distance rate didn't improved. It could be concluded, that a constant moving reward function is hard to learn with only laser-scan data.

Comparing the results of test number 7,8 and 9 (the three reward values for the linear dependent reward function 3) it sticks out, that as the reward values are chosen larger the success rate is getting smaller, the timeout rate is getting larger and the safety distance rate is getting smaller. Both the success rate and the timeout rate are rates of episode endings. As well as the human and wall hit rate, which both are pretty equal for all three test numbers. When two of the episode ending rates are nearly constant, a rising timeout rate implies a declining success rate. The rising time out rate can be explained with the declining safety distance rate. A smaller safety distance rate means that on average over time the humans are less frequent inside the safety distance. As the agent has only the laser-scan data as input it can't really distinguish between walls and legs of humans. So it not only keeps a larger distance to humans, it also keeps a larger distance to some walls on average and this makes some configuration of walls harder to drive around and therefore leads to a higher timeout rate. The second influence of a smaller safety distance rate on the timeout rate is the fact, that keeping a larger distance to humans on average also means for the robot, that it has to do some evasive moves to avoid humans and therefore has to drive a longer route, what in the end also needs more time.

In comparison to the base case have the three test cases of reward function 3 a better human hit rate. This is mainly caused by the training with humans as the rate stays at 1% like in tab. 5 of the section before.

To conclude: It could be shown with test number 9, that a large value of the distance change reward can lead to a improvement of the safety distance rate. This improvement comes with the cost of a larger time out rate. While comparing the results of reward function 2 with those of reward function 3 it is obvious, that the linear dependent reward of reward function 3 can be learned better than the constant reward of reward function 2.

(ii) additional data of 3 observed humans with a 360° camera and only rewards for observed humans. As second step the different reward values for moving towards/away from humans are compared again, but this time with additional input data of 3 observed humans (`num_obs_humans` = 3) with a 360° camera. This means, the distance and angle of the nearest 3 humans are given

to the agent as additional input and the humans can be observed all around the robot. Also the rewards are calculated only for the 3 nearest humans. As described in tab. 3 this applies for reward function 1 and 4. In the following tab. 7 three values for reward function 1 (constant reward) and two values for reward function 4 (linear dependent reward) are compared with the base case (test number 2).

Test number	10	11	12	13	14	2
reward_function	1	1	1	4	4	-
reward_distance_to_human_decreased	-0,3	-2	-5	-50	-100	-
reward_distance_to_human_increased	0,3	2	5	50	100	-
Success rate [%]	**94,0**	89,8	11,4	87,6	87,1	90,2
Human hit rate [%]	0,5	0,4	1,0	0,6	**0,3**	2,2
Wall hit rate [%]	**2,5**	3,4	11,7	2,8	3,6	5,3
Timeout rate [%]	2,9	6,4	76,0	9,0	9,0	**2,3**
Safety distance rate [%]	20,7	16,2	22,5	14,5	**12,8**	20,7

Table 7: Results for (ii) additional data of 3 observed humans with a 360° camera and only rewards for observed humans. Here three values for reward function 1 (constant reward) and two values for reward function 4 (linear dependent reward) are compared with the base case (test number 2). Settings: `num_obs_humans` = 3; `camera_angle` = 360;

Comparing the results of test number 10, 11 and 12 (the three reward values for the constant reward function 1) is a good example of "to low", "good" and "to high" chosen reward values. The reward value of test number 10 was chosen to low, because the safety distance rate didn't improve compared to the base case (test number 2). In test number 11 on the other hand this rate could be improved to $16,2\%$. Also worth to mention is the low human hit rate of just $0,4\%$ and $0,5\%$ in these both tests. A to high chosen reward can be seen in test number 12. The success rate is really low at just $11,4\%$, because the robot got a to large reward by just following humans and driving back and forth over the boundary of the safety distance. The reward by doing so was larger than the reward for driving to the goal. Thus leading to a small success rate. This problem may be fixed by just selecting a smaller reward value for `reward_distance_to_human_increased` and leaving the other value for decreasing the distance as it is. A suggestion that remains to be tested in the future.

Comparing the results of test number 13 and 14 (the two reward values for the linear dependent reward function 4) with the base case it can be shown, that the safety distance rate can be really improved a lot. But as in case (i) a small safety distance rate comes with the cost of a large time out rate. In contrast to case (i) the costs of the time out rate per improved safety distance rate are smaller. This is probably mainly caused by the additional input information of case (ii), because the agent is now able to distinguish between humans and walls. Therefore only the second influence of the safety distance rate to the time out rate can be used for explanation: While the robot tries to keep a larger distance to humans on average, it has to drive longer routes, thus needing more time on average to drive to the goal.

In test number 14 the lowest safety distance rate and human hit rate of all tests could be achieved. In comparison to the tests of reward function 1 it could be argued, that the linearly dependent reward is better than the constant, because it achieved a lower safety distance rate. But keep in mind, that another constant reward value could lead to similar good results as test number 14, for example a constant reward value of -3/3 or -4/4.

To conclude: It is important how large the reward value is chosen, it can be to high or to low. Furthermore it could be shown, that the additional input data of 3 observed humans helped to

improve the safety distance rate and human hit rate. The improvement of the safety distance rate comes again with the cost of a larger time out rate, but slightly better than in case (i) with no additional input.

(iii) additional data of 3 observed humans with a 90° camera and only rewards for observed humans. In contrast to the tests in the paragraph before, where the camera angle was 360°, is the camera angle this time 90°. As the robot again can only observe 3 humans inside the 90° camera, the reward gets only calculated for these 3 observed humans. As described in tab. 3 this applies for reward function 1 and 4. In the following tab. 8 two values for reward function 1 (constant reward) and two values for reward function 4 (linear dependent reward) are compared with the base case (test number 2).

Test number	15	16	17	18	2
reward_function	1	1	4	4	-
reward_distance_to_human_decreased	-0,3	-2	-50	-100	-
reward_distance_to_human_increased	0,3	2	50	100	-
Success rate [%]	**90,9**	76,8	50,0	72,6	90,2
Human hit rate [%]	**1,0**	2,4	3,1	1,6	2,2
Wall hit rate [%]	**2,7**	5,1	7,4	6,0	5,3
Timeout rate [%]	5,4	15,7	39,6	19,8	**2,3**
Safety distance rate [%]	20,2	19,1	**17,6**	19,5	20,7

Table 8: Results for (iii) additional data of 3 observed humans with a 90° camera and only rewards for observed humans. In this table two values for reward function 1 (constant reward) and two values for reward function 4 (linear dependent reward) are compared with the base case (test number 2). Settings: `num_obs_humans` = 3; `camera_angle` = 90;

Comparing the results of test number 15 and 16 (the two reward values for the constant reward function 1) it can be noticed, that the lower reward value of test number 15 has a better overall test result than the larger value of test number 16. Only the safety distance rate could be improved slightly to $19,1\%$ in test number 16.

Comparing the results of test number 17 and 18 (the two reward values for the linear dependent reward function 4) the opposite picture is shown. Here it's the lower reward value of test number 17, which has the worse over all results than the larger value of test number 18. But the safety distance rate is better in test number 17. It is important to notice, that this is unusual. In the two paragraphs before always the larger reward value had the lower safety distance rate.

When all four tests are compared with the base case, only test number 15 could improve something while not dropping the success rate. It improved the human hit rate to $1,0\%$. The other 3 test could improve the safety distance rate a bit, but at the same time their success rate dropped a lot. Overall it can be said, that all four tests not really lead to a significant improvement.

To conclude: Calculating the rewards only for the observed humans inside a 90° camera doesn't lead to a significant improvement and therefore seems to be not the best choice.

(iv) additional data of 3 observed humans with a 90° camera, but rewards for all humans inside the level. To overcome the problems of the last paragraph the rewards are again calculated for all humans inside the room, while the additional input data still remains the same, 3 observed humans inside a 90° camera. As described in tab. 3 this applies only for reward

function 2 and 3. In the following tab. 9 two values for reward function 2 (constant reward) and two values for reward function 3 (linear dependent reward) are compared with the base case (test number 2).

Test number	19	20	21	22	2
reward_function	2	2	3	3	-
reward_distance_to_human_decreased	-0,3	-2	-50	-100	-
reward_distance_to_human_increased	0,3	2	50	100	-
Success rate [%]	-	86,1	**93,2**	79,0	90,2
Human hit rate [%]	-	0,9	**0,5**	1,6	2,2
Wall hit rate [%]	-	4,0	**2,4**	5,6	5,3
Timeout rate [%]	-	9,0	3,9	13,8	**2,3**
Safety distance rate [%]	-	17,0	17,2	**14,3**	20,7

Table 9: Results for (iv) additional data of 3 observed humans with a 90° camera, but rewards for all humans inside the level. Here two values for reward function 2 (constant reward) and two values for reward function 3 (linear dependent reward) are compared with the base case (test number 2). Settings: `num_obs_humans` = 3; `camera_angle` = 90;

Comparing the results of test number 19 and 20 (the two reward values for the constant reward function 2) ...

Comparing the results of test number 21 and 22 (the two reward values for the linear dependent reward function 3) it can be shown, that test number 21 has pretty good results whereas test number 22 could improve the safety distance rate to 14, 3%. In comparison to the base case has test number 21 improved four rates out of five, the success rate, human hit rate, wall hit rate and the safety distance rate. With this result it could be proven, that the additional semantic information can improve the behaviour of the robot to humans. Also test number 21 is applicable for deployment on a real robot, with just a 90° camera on top and 360° of laser-scan data.

To conclude: The results prove, that the additional inputs of just a 90° camera can improve the safety distance rate and human hit rate. Thus giving the best applicable results for the real world.

Comparison of the cases (i)-(iv) with one reward value of -50/50. In this paragraph one result of each case (i)-(iv) is compared directly to the others to show the influence of different additional inputs to the agent. Therefore the four results of the reward value of -50/50 are pictured in one diagram 14.

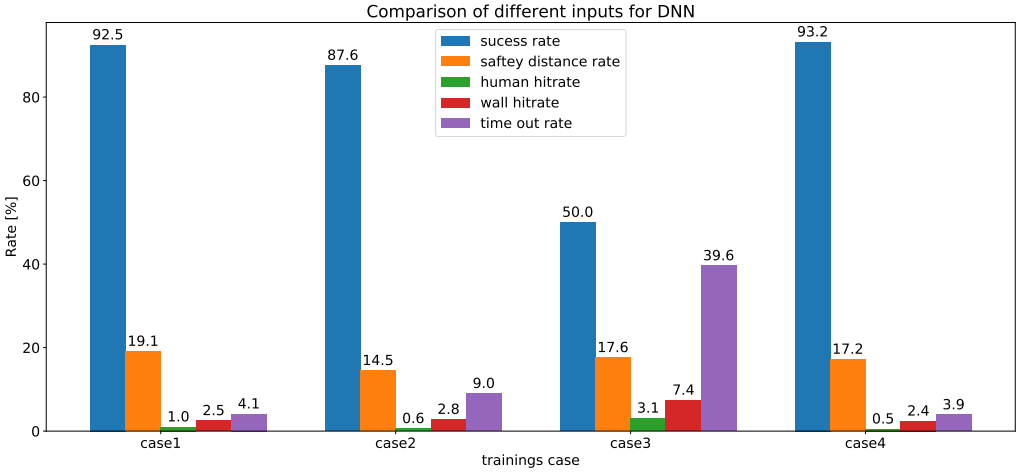


Figure 14: Comparison of the cases (i)-(iv) with one reward value of -50/50.

In the following the result of each case is shortly explained and compared to the others:

- **Case (i) - no additional inputs:** The agent has only the laser-scan data to distinguish between humans and walls. That's why the safety distance rate only improved from 21% to $19,1\%$. A larger reward value or additional input information are needed to further improve this rate.

- **Case (ii) - data of 3 humans in $360°$ camera as input:** The agent has additional information of 3 humans in all directions and can better distinguish between humans and walls. Hence it is easier to learn the reward function, what leads to an improvement of the safety distance rate to $14,5\%$. This improvement comes with the costs of a larger time out rate. Also the human hit rate could be improved with these additional inputs.

- **Case (iii) - data of 3 humans in $90°$ camera as input and the reward is calculated only for these 3 humans:** As the rewards are only calculated for the humans in front of the robot, the agent can avoid these rewards by just turning away from the humans and driving backwards. This is an unintended behavior of the robot. And leads to bad results with a success rate of 50% and a human hit rate of $3,1\%$. This emphasises the importance of the reward shaping.

- **Case (iv) - data of 3 humans in $90°$ camera as input, but the reward is calculated for all humans in the room:** The problems of case (iii) can be solved by calculating the rewards for all humans in the room. The safety distance rate could be improved to $17,2\%$ and the human hit rate to $0,5\%$, while maintaining a large success rate of $93,2\%$. A better safety distance rate could only be achieved in case (ii) due to the $360°$ camera giving perfect additional information of the humans in all directions. Notice, that case (iv) not always has a better success rate than case (ii) as it can be seen for example with reward value -100/100. But case (ii) has always the best safety distance rate compared to the other three cases. The second best safety distance rate and overall results could be achieved in case (iv). Since case (ii) needs a $360°$ camera, it is not really applicable for deployment on a real robot. Whereas case (iv) only needs a $90°$ camera and has therefore the best applicable results for the real world.

7.1.4 Repeated training sessions

In this part the influence of repeating a training session on the results is evaluated. Therefore two training sessions have been repeated with the exact same settings. The results of test number 3 and test number 13 with their repetitions are shown in tab. 10.

Test number	3	3v2	13	13v2
Success rate [%]	93,9	92,2	87,6	88,9
Human hit rate [%]	1,2	1,5	0,6	0,4
Wall hit rate [%]	2,5	3,5	2,8	4,6
Timeout rate [%]	2,5	2,8	9,0	6,1
Safety distance rate [%]	21,6	20,4	14,5	15,5

Table 10: Results of repeated training sessions. Here the results of test number 3 and test number 13 with their repetitions 3v2 and 13v2 are shown.

Comparing test number 3 with its repetition 3v2 and comparing test number 13 with its repetition 13v2 show both, that the success rate, wall hit rate, timeout rate and safety distance rate can all differ 1-3 percent points from one training session to its repetition. Only the human hit rate differs just 0,2-0,3 percent points. Notice, that for all rates except the human hit rate a 1-3 percent point difference between tests has no meaning, because this could have happened by chance by just repeating the training session. This also applies for the human hit rate with a 0,3 percent points difference between tests.

To conclude: The rates differ between training sessions to a certain extend. The larger the difference is the more meaning can be given to it.

7.2 Test of DQN agent on level custom and maze

In this part the influence of additional obstacles in the level is tested. In these tests the DQN agent was trained on the *custom* and *maze* level. For comparison the same agent as in the base case of sec. 7.1.1 (agent without additional input, without additional rewards and without humans) has been trained on the custom and maze level. Also different numbers of static obstacles were tested. The results of four tests are shown in tab. 11.

Test number	1	23	24	25
Train and Test level	random	custom	custom	maze
num_obstacles	8	8	6	0
Success rate [%]	**94,4**	50,4	87,4	81,6
Wall hit rate [%]	**1,9**	14,2	5,4	10,3
Timeout rate [%]	**3,7**	35,4	7,2	8,1

Table 11: Results of tests of the DQN agent trained on level random, custom and maze.

In test number 1 the DQN agent was trained and tested on the random level with 8 randomly placed and shaped static obstacles. The good result of this test can be used to compare with the results of the other two levels.

In test number 23 the agent was trained and tested on the custom level with also 8 static obstacles as in the random level, but this time some are replaced by randomly shaped and placed corridors. With a success rate of $50,4\%$ and a wall hit rate of $14,2\%$ the result of this test is bad compared to test number 1.

But if the number of obstacles is just 6 in the custom level, as in test number 24, the result can be improved towards the result of the random level. This aspect shows, that the number

of static obstacles has a influence on the training results. If the number is to large, some level tend to be to complex and some tend to have infeasible goals. Both are aspects, which lead to a lower success rate. One solution to improve the success rate even with eight obstacles in the custom level is to first train the network with a smaller number of obstacles. Then train this pre-trained network again with a larger number of obstacles. This leads to a better success rate. In a test-scenario where the agent was trained on the custom level with eight obstacles given a network, pre-trained before on six obstacles, a success rate of $77,5\%$ could be reached.

In test number 25 the agent was trained and tested on the maze level with no static obstacles besides the randomly placed walls of the maze (see sec. 6.1.1). With no additional static obstacles the agent could achieve a success rate of $81,6\%$, which is quite small compared to the $94,4\%$ of test number 1. And the wall hit rate is large with $10,3\%$ compared to $1,9\%$. With additional static obstacles these result would probably get worse.

To conclude: The more complex level custom and maze have worse results than the easier level random. It seems, that complex level still remain a challenge for the DQN agent.

7.3 Test of LSTM and GRU agent on level random

Here the influence of different agents is tested. In the following tab. 12 the LSTM and GRU agents are compared to the DQN agent, all three trained and tested on the random level without humans and without additional input data and additional calculated rewards.

Test number	1	26	27
Agent	DQN	LSTM	GRU
Success rate [%]	**94,4**	86,4	91,5
Wall hit rate [%]	**1,9**	8,2	4,4
Timeout rate [%]	**3,7**	5,4	4,2

Table 12: Results of tests of the DQN, LSTM and GRU agent trained on level random.

Both the LSTM and GRU agent have worse results than the DQN agent, because they both have a smaller success rate than $94,4\%$, a larger wall hit rate than $1,9\%$ and a larger timeout rate than $3,7\%$. But the GRU agent has a better result than the LSTM agent. The results of these three tests alone are not really meaningful. But also with a few more test results, which can be found in the table in the appendix 10.2.3, these results are merely a hint for the question, how well a certain agent is doing. This means, that other training sessions and tests with other settings in the future could maybe show, that the LSTM or GRU agent can be better than the DQN agent.

To conclude: In the here proposed training sessions and tests the LSTM and GRU agent had worse results than the DQN agent.

8 Conclusion

The goal of this project was to improve human-robot interaction in path planning with reinforcement learning. To achieve this goal four main components were implemented.

Firstly the simulation environment was modified to close the simulation-reality gap and prevent overfitting. For this purpose three different level generation methods were presented that used randomized static objects for a dynamic environment for the agent to navigate in. More importantly for further modelling after a real industrial environment, humans were incorporated to the simulation environment. To this end we modelled the motion profile after real life humans, as well as human behaviour like interacting with other humans or changing direction and speed while walking. The incorporating of those dynamic and randomized elements made it more challenging for the agent to plan its path and reach the goal.

Secondly new rewards as well as new inputs for the training of the neural networks were included. To give the neural network additional semantic information about humans, the angle and the distance to the humans in the room were transmitted to the neural network as an input. Therefore well suited solutions for two general problems had to be implemented:

1. The number of inputs in a DNN is fixed while the number of humans in the room is variable.
   Solution: Only transmit the N-nearest humans in the room.

2. The later used robot has only one camera mounted with a limited angle for human detection.
   Solution: Implementation of a variable camera angle in simulation for human detection.

The calculation of the rewards has also been adapted to deal with humans. Therefore three new rewards (hit human, moving towards/away from human) have been introduced. A detailed description of the customized training session can be found in sec.6.1.2.

Several training sessions were carried out for testing. The resulting networks were evaluated with a focus on the improvement on the main goal: Find a goal while maintaining a sufficient distance to humans. The following main aspects could be learned from the various training sessions:

- Just training the agent with humans leads to a improved human hit rate.
- If a good value for the rewards is chosen, the robot can keep a larger distance to humans on average (called *safety distance rate*).
- The additional semantic information further improves the safety distance rate and the human hit rate.
- Importance of finding a well suited reward function: Don't calculate the rewards only for humans inside a small camera angle in front of the robot.
- By repeating training sessions the uncertainties in the evaluation parameters are shown.
- More complex environments have worse results.
- The best RL agent for this specific approach is the DQN agent.

In the end, the settings of test number 21 (see tab. 9) turned out to be the most suitable for a later use in real world.

Lastly in order to be able to test the training results on a real robot, a object detection and positioning technique has been implemented using a RGB-D camera and an object detection framework, which was all embedded in ROS. unfortunately there was no time left to test the ROS implementation on the real robot.

9 Future Work

In this section future work and further improvements for our approach and implementation are presented. This focuses mainly on the areas in which we participate. These are the improvement of the simulation environment, adapting reward shaping and further evaluation of the performance of the trained model.

Improving the dynamic environment

As the focus lies on simulating dynamic environments, enhancing the behaviours of this dynamics inside the framework would be beneficial. One way would be of course implementing additional dynamic obstacles. Those could be other robots, that have their own behaviour for the agent to predict. This would also result in human robot collaboration tasks, where humans interacting with other robot are behaving differently from human to human interaction. This could lead e.g to changes in the reward shaping, as the safety-distance might get reduced, if the human to robot interaction in this scenario is slower than human to human interaction.
Another possibility in order to close the simulation reality gap, is the improvement of the human wanderer. To this purpose the behaviour and motion of the human could be modelled even more realistically. For this purpose it could be feasible to incorporate one of the already existing human simulation models e.g Pedestrian Crowd Simulation (PEDSIM) [7]. This simulator provides more detailed and sophisticated implementations of human motion modelling, individual trajectories of the agents and crowd behaviour.

Improving reinforcement learning

The results have shown how important it is to have a suitable reward function. Through the intelligent choice of rewards, the trained behavior of the robot can be greatly influenced. Thus more effort on testing different rewards and their influence on the behavior of the robot should be investigated. The following list are possible changes to the reward function:

- Include dynamic reward based on human stride length and speed. Thus the robot may be able to learn more about human running behavior.
- Calculate the hit reward as soon as the robot falls below a minimum distance to a person. This may better protect the safety distance to people.

Furthermore, in order to be able to realistically apply the trained network, further semantic information has to be included. Interaction with other robots would be one important aspect. With the introduction of new semantic information, setting up a suitable reward function becomes even more difficult, as further behavior must be trained.
Another useful extension would be a dynamic goal, which moves while the robot tries to find it.

Evaluation in real world environment

So far the agent was trained, tested and evaluated in simulation. Because of the simulation reality gap, the model needs to be tested and evaluated in a real scenario too. To this purpose the developed ROS architecture needs to be deployed on the TUrtleBot and evaluated.
Furthermore more testing and evaluating can be done while using different network structures for the agent, like long short-term memory (LSTM), gated recurrent unit (GRU) or asynchronous actor-critic agents (A3C). Therefore this could mitigate some of the weaknesses the DQN network brings to the table, like possessing a finite replay buffer, which may lead to loss of information that is gathered on the very beginning of an episode or being able to train on several environments at once.

Bibliography

[1] Patrick Dammann. "Einführung in das Reinforcement Learning". In: (), pp. 1–12.

[2] Michael Everett, Yu Fan Chen, and Jonathan P. How. "Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning". In: *CoRR* abs/1805.01956 (2018). arXiv: 1805.01956. URL: http://arxiv.org/abs/1805.01956.

[3] Ronja Gueldenring. *Applying Deep Reinforcement Learning in the Navigation of Mobile Robots in Static and Dynamic Environments*. Hamburg, 2019.

[4] Linh Kästner, Cornelius Marx, and Jens Lambrecht. "Deep-Reinforcement-Learning-Based Semantic Navigation of Mobile Robots in Dynamic Environments". In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2020, pp. 1110–1115.

[5] D. Kiss and D. Papp. "Effective navigation in narrow areas: A planning method for autonomous cars". In: *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2017, pp. 000423–000430.

[6] T. Klaas, J. Lambrecht, and E. Funk. "Semantic Local Planning for Mobile Robots through Path Optimization Services on the Edge: a Scenario-based Evaluation". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 711–718. DOI: 10.1109/ETFA46521.2020.9212092.

[7] A. Kormanová, M. Varga, and N. Adamko. "Hybrid model for pedestrian movement simulation". In: *The 10th International Conference on Digital Technologies 2014*. 2014, pp. 152–158. DOI: 10.1109/DT.2014.6868707.

[8] Steven M. LaValle. *Planning Algorithms*. USA: Cambridge University Press, 2006. ISBN: 0521862051.

[9] Renato Martins et al. "Extending Maps with Semantic and Contextual Object Information for Robot Navigation: a Learning-Based Framework Using Visual and Depth Cues". In: *Journal of Intelligent Robotic Systems* 99.3-4 (Feb. 2020), pp. 555–569. ISSN: 1573-0409. DOI: 10.1007/s10846-019-01136-5. URL: http://dx.doi.org/10.1007/s10846-019-01136-5.

[10] Á. Nagy, G. Csorvási, and D. Kiss. "Path planning and control of differential and car-like robots in narrow environments". In: *2015 IEEE 13th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2015, pp. 103–108.

[11] Ian Parberry. *Introduction to Game Physics with Box2D*. 1st. USA: CRC Press, Inc., 2013. ISBN: 1466565764.

[12] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[13] C. Rösmann, F. Hoffmann, and T. Bertram. "Kinodynamic trajectory optimization and control for car-like robots". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 5681–5686. DOI: 10.1109/IROS.2017.8206458.

10 Appendix

10.1 Installation Guide

From fresh Ubuntu 18.04 LTS install

1. Install CUDA, follow instructions exactly and fully from https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html. In this project NVIDIA driver 455 and CUDA 11 was used, but other versions might work as well.

2. Install opencv (We used 3.4.6 other versions produced errors for some reaseon) as well as opencv-contrib as it is needed for arena2-real and make sure to checkout the correct version:

   - https://docs.opencv.org/master/d7/d9f/tutorial_linux_install.html
   - https://github.com/opencv/opencv_contrib

3. SDL2 is also requiered for arena2-real: https://wiki.libsdl.org/Installation

4. as well as pytorch:

   - pip install torch==1.5.0+cpu torchvision==0.5.0+cpu -f

   https://download.pytorch.org/whl/torch_stable.html

5. Next we need to install ROS melodic from: http://wiki.ros.org/ROS/Installation and set up the workkspace: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

6. For the Intel Realsense first install the Realsense SDK: https://github.com/IntelRealSense/librealsense/blob/master/doc/distribution_linux.md#installing-the-packages and then clone and install the roswrapper from: https://github.com/IntelRealSense/realsense-ros.

7. Last install the darknet-ros node (YOLO) from: https://github.com/tom13133/darknet_ros. If you encounter computing compatability errors you might have to comment out the line -gencode arch=computeXXXXXXX in the CMakeLists.txt and or set the correct architecture type for your cpu (see: https://en.wikipedia.org/wiki/CUDA#Supported_GPUs).

   After that you have to change the topic name in config/ros.yaml to /camera/color/image_raw and then incooperate the weights and config for yolov3-tiny from https://pjreddie.com/darknet/yolo/. yolov4-tiny should work as well but in our graphics gard has not enough memory for CUDA to then run properly.

10.2 Results

10.2.1 Standard Settings

The following settings are the base for nearly all training sessions shown in sec. 7.

```
video{
        resolution_w = 800 # window width
        resolution_h = 600 # window height
        window_x = -1 # window x position on startup -1 for centered
        window_y = -1 # window y position on startup -1 for centered
        maximized = 0 # start window maximized 1/0
        msaa = 4 # multisampling anti-aliasing 2,4,8,16
        vsync = 0 # vertical synchronization 1/0
        fps = 60 # video frames per second
        fullscreen = 0 # fullscreen enabled 1/0
        enabled = 1 # video mode enabled 1/0; if set to 0
}

gui{
        font_size = 16 # gui font size
        show_robot = 1 # show/hide the robot
        show_stage = 1 # show/hide the stage
        show_laser = 1 # show/hide the laser
        show_stats = 1 # show/hide the statistics (e.g. #episodes)
        show_goal = 1 # show/hide goal
        show_goal_spawn = 0 # show/hide the spawn area for the goal
        show_trail = 1 # show/hide robot trail
        camera_follow = 0 # if == 1 camera follows robot, (if == 2, rotation is
            ↪  also taken into acount)
        camera_x = 0.0 # initial position of camera
        camera_y = 0.0 # initial position of camera
        camera_zoom = 0.45 # initial zoom of camera (zoom actually means
            ↪ scaling of the view -> camera_zoom < 1 means zooming out)
        camera_rotation = 0.0 # view rotation in degree
        camera_zoom_factor = 1.3 # how does scaling increase/decrease with each
            ↪  zoom step
}

keys{
        up = KEY_UP # key for moving forward
        left = KEY_LEFT # key for moving left
        down = KEY_DOWN # key for moving backward
        right = KEY_RIGHT # key for moving right
        reset = KEY_r # key for resetting robot
        play_pause_simulation = KEY_SPACE # key for playing/pausing simulation
}

physics{
        time_step = 0.1 # physics time step
        step_iterations = 5 # how often to perform a physics update per step
        fps = 60 # how many times per second a simulation step is performed
            ↪ with step_iterations sub steps
```

```
        position_iterations = 8 # position iterations for each time step (
            ↪ higher value increases simulation accuracy)
        velocity_iterations = 12 # velocity iterations for each time step (
            ↪ higher value increases simulation accuracy)
}

training{
        max_time = 100.0 # maximum time per episode (actual time, so physics.
            ↪ time_step influences maximum number of steps per episode)
        episode_over_on_hit = 1 # if set to 1 episode ends if an obstacle is
            ↪ hit
        reward_goal = 100.0 # reward for reaching goal
        reward_towards_goal = 0.1 # reward when distance to goal decreases
        reward_away_from_goal = -0.2 # reward when distance to goal increases
        reward_hit = -100.0 # reward for hitting obstacle
        reward_time_out = 0.0 # reward when episode timed out (after max_time
            ↪ seconds)
        num_envs = 1 # number of parallel environments
        num_threads = -1 # number of threads to run in parallel, if set to -1
            ↪ number of cpu cores will be detected automatically
        agent_class = "Agent" # name of class in agent python script
        reward_human = -100.0 # reward for hitting a human
        safety_distance_human = 0.8 # safty distance to human, which should be
            ↪ always fullfilled
        reward_distance_to_human_decreased = -2.0 # reward when distance to
            ↪ human decreases
        reward_distance_to_human_increased = 2.0 # reward when distance to
            ↪ human increases
        num_obs_humans = 0 # maximum number of humans the agent can observe
            ↪ inside the camera view
        reward_function = 1 # choose between different distance reward
            ↪ functions (1 = constant + only observed humans in camera angle; 2
            ↪  = constant + for all humans; 3 = linear dependent + for all
            ↪ humans; 4 = linear dependent + only observed humans in camera
            ↪ angle)
}

robot{
        laser_noise = 0.0 # random, uniformly distributed offset with a maximum
            ↪  of +/- laser_noise*distance_measured (a value of 0 means perfect
            ↪  laser data -> no noise)
        laser_max_distance = 3.5 # maximum distance the laser can recognize
        laser_start_angle = 0.0 # angle in degree of first sample
        laser_end_angle = 359.0 # angle in degree of last sample
        laser_num_samples = 360 # number of laser samples
        laser_offset{ # offset of laser from base center
                x = 0.0
                y = 0.0
        }
        base_size{ # width(x) and height(y) of robot base
                x = 0.13
```

```
                y = 0.13
        }
        wheel_size{ # width(x) and height(y) of robot wheels
                x = 0.018
                y = 0.064
        }
        wheel_offset{ # offset of wheels from edge of base
                x = 0.0
                y = 0.034
        }
        bevel_size{ # size of bevel along x/y axis at the base corners
                x = 0.025
                y = 0.025
        }
        forward_speed{ # velocity for forward action
                linear = 0.2
                angular = 0.0
        }
        left_speed{ # velocity for left action
                linear = 0.15
                angular = 0.75
        }
        right_speed{ # velocity for right action
                linear = 0.15
                angular = -0.75
        }
        strong_left_speed{ # velocity for strong left action
                linear = 0.0
                angular = 1.5
        }
        strong_right_speed{ # velocity for strong right action
                linear = 0.0
                angular = -1.5
        }
        backward_speed{ # velocity for backward action
                linear = -0.1
                angular = 0.0
        }
        camera_angle = 90.0 # camera view in degree
}

stage{
        random_seed = 0 # seed for pseudo random generator
        initial_level = "random" # level loaded on startup (-1 for none)
        level_size = 4.0 # width and height of default levels
        max_obstacle_size = 1.0 # maximum diameter of static obstacles
        min_obstacle_size = 0.1 # minimum diameter of static obstacles
        num_obstacles = 8 # number of static obstacles
        dynamic_obstacle_size = 0.3 # size of dynamic obstacle
        num_dynamic_obstacles = 4 # number of dynamic obstacles in
            ↪ static_dynamic level
```

```
        obstacle_speed = 0.08 # in m/s for dynamic obstacles
        max_time_chatting = 50.0 # maximum time for chatting between two
            ↪ wanderers
        goal_size = 0.1 # diameter of circular goal to reach
        svg_path = "svg_levels/" # path to folder where svg files are stored
        static_map_ros_service_name = "/static_map" #name of map service
            ↪ provided ros map server.
}
```

10.2.2 Settings of the DQN agent

Only the MEAN_REWARD_BOUND was changed between some training sessions to get longer or shorter trained agents. Especially when the reward values are large the MEAN_REWARD_BOUND has to be larger as well to not end a training to early.

```
### hyper parameters ###

MEAN_REWARD_BOUND = 120.0 # training is considered to be done if the mean
    ↪ reward reaches this value
NUM_ACTIONS = 7    # total number of discrete actions the robot can perform
DISCOUNT_FACTOR = 0.99  # discount factor for reward estimation (often denoted
    ↪ by gamma)

SYNC_TARGET_STEPS = 2000 # target net is synchronized with net every X steps
LEARNING_RATE = 0.00025  # learning rate for optimizer
EPSILON_START = 1   # start value of epsilon
EPSILON_MAX_STEPS = 10**6 # how many steps until epsilon reaches minimum
EPSILON_END = 0.02   # min epsilon value
BATCH_SIZE = 64    # batch size for training after every step
TRAINING_START = 1000  # start training only after the first X steps
MEMORY_SIZE = 1000000  # last X states will be stored in a buffer (memory),
    ↪ from which the batches are sampled
N_STEPS = 2
DOUBLE = True
########################
```

10.2.3 Training Results Table

Tabelle1

Agent	Level	Test		Result								Settings										Training - agent best (*last)			Date
		Test number	short description	Success rate [%]	Standard deviation [%]	Human hit rate [%]	Standard deviation [%]	Wall hit rate [%]	Standard deviation [%]	Timeout rate [%]	Standard deviation [%]	safety distance rate [%]	physics step_lite rations	reward_ human	safety_d istance_ human	reward_distan ce_to_human decreased	reward_distan ce_to_human increased	num_o bs_hu mans	reward_ function	stage: num_ob stacles	robot: camera_ angle	trained episodes	mean success [%]	name of training folder
DQN	random	1		94,4	2,1	0	1,9	1,2	3,7	1,7		5	-100	0,8		-0,3	0,3	0	1	8	90	217508	100	training_Mon07_00-01-23
	random	2	test random -human	90,2	3,1	2,2	1,5	5,3	2,1	2,3	1,5	20,7	5	-100	0,8	-0,3	0,3	0	1	8	90	217508	100	training_Mon07_00-01-23
	random--human	3	human_reward 0	93,9	2,3	1,2	1,2	2,5	1,5	2,5	1,3	21,6	5	0	0,8	-0,3	0,3	0	1	8	90	103605	98	training_Tue06_17-33-43
	random--human	3v2	human_reward 0	92,2	2,5	1,5	1,1	3,5	1,8	2,8	1,7	20,4	5	0	0,8	-0,3	0,3	0	1	8	90	89526	99	training_Fri09_18-03-55
	random--human	4	human_reward -100	89,9	2,7	1,2	1,1	3,6	1,8	5,3	2,3	20,5	5	-100	0,8	-0,3	0,3	0	1	8	90	143164	98	training_Tue15_12-52-23
	random--human	5	rf 2; dist_reward 0,3	87,1	3,3	3,1	1,7	5,6	2,3	4,3	1,9	19,4	5	-100	0,8	-0,3	0,3	0	2	8	90	63464	98	training_Wed30_19-07-20
	random--human	10	rf 1; camera_angle 360; dist_reward 0,3	94	2,3	0,5	0,7	2,5	1,5	2,9	1,7	20,7	5	-100	0,8	-0,3	0,3	3	1	8	360	353475	100	training_Fri30_17-19-32
	random--human	15	rf 1; camera_angle 90; dist_reward 0,3	90,9	2,9	1	0,9	2,7	1,8	5,4	2,1	20,2	5	-100	0,8	-0,3	0,3	3	1	8	90	149412	99	training_Fri16_19-37-52
	random--human	19	rf 2; camera_angle 90; dist_reward 0,3										5						2	8	90			
	random--human	12	rf 1; camera_angle 360; dist_reward 5	11,4	3,6	1	1	11,7	3,5	76	4,4	22,5	5	-100	0,8	-5	5	3	1	8	360	23902	18	training_Tue20_15-55-10
	random--human	6	rf 2; dist_reward 2	80,5	3,4	2,4	1,5	3,4	1,8	13,8	3,1	20,6	5	-100	0,8	-2	2	0	2	8	90	221578	88	training_Mon26_14-24-54
	random--human	11	rf 1; camera_angle 360; dist_reward 2	89,8	2,8	0,4	0,6	3,4	1,8	6,4	2,6	16,2	5	-100	0,8	-2	2	3	1	8	360	95725	96	training_Wed21_00-39-14
	random--human	16	rf 1; camera_angle 90; dist_reward 2	76,8	3,9	2,4	1,6	5,1	2,1	15,7	3,6	19,1	5	-100	0,8	-2	2	3	1	8	90	763918	87	training_Mon26_14-29-12
	random--human	20	rf 2; camera_angle 90; dist_reward 2	86,1	3,6	0,9	0,9	4	2,2	9	2,8	17	5	-100	0,8	-2	2	3	2	8	90	499775	98	training_Fri30_17-02-16
	random--human	7	rf 3; dist_reward 10	94,3	2,2	0,9	0,9	2,1	1,4	2,7	1,5	21,1	5	-100	0,8	-10	10	0	3	8	90	123465	99	training_Sun18_01-37-44
	random--human	9	rf 3; dist_reward 100	84	3,4	1	1	2,9	1,7	12,1	3,2	15,5	5	-100	0,8	-100	100	0	3	8	90	131081	89	training_Sun18_16-13-22
	random--human	14	rf 4; camera_angle 360; dist_reward 100	87,1	2,8	0,3	0,6	3,6	1,7	9	2,4	12,8	5	-100	0,8	-100	100	3	4	8	360	272080	91	training_Fri30_17-13-25
	random--human	18	rf 4; camera_angle 90; dist_reward 100	72,6	3,2	1,6	1,2	6	2	19,8	3,1	19,5	5	-100	0,8	-100	100	3	4	8	90	454132	69	training_Fri30_17-14-41
	random--human	21	rf 3; camera_angle 90; dist_reward 100	79	3,7	1,6	1,1	5,6	2,6	13,8	3,6	14,3	5	-100	0,8	-100	100	3	3	8	90	357793	88	training_Fri30_17-16-01
	random--human	8	rf 3; dist_reward 50	92,5	2,3	1	1	2,5	1,5	4,1	1,8	19,1	5	-100	0,8	-50	50	0	3	8	90	587045	99	training_Mon26_14-17-45
	random--human	13v2	rf 4; camera_angle 360; dist_reward 50	88,9	2,8	0,4	0,6	4,6	2,1	6,1	2,2	15,5	5	-100	0,8	-50	50	3	4	8	360	*50837	*86	training_Tue20_09-22-35
	random--human	13	rf 4; camera_angle 360; dist_reward 50	87,6	3	0,6	0,8	2,8	1,6	9	2,6	14,5	5	-100	0,8	-50	50	3	4	8	360	51649	91	training_Mon19_22-26-16
	random--human	17	rf 4; camera_angle 90; dist_reward 50	50	4,5	3,1	1,9	7,4	2,6	39,6	4,7	17,6	5	-100	0,8	-50	50	3	4	8	90	*65244	*65	training_Mon26_14-20-17
	random--human	22	rf 3; camera_angle 90; dist_reward 50	93,2	2,6	0,5	0,7	2,4	1,4	3,9	2,2	17,2	5	-100	0,8	-50	50	3	3	8	90	724762	99	training_Mon26_14-22-25
	custom	23		50,4				14,2		35,4			5	-100	0,8	-0,3	0,3	0	1	8	90	130688	57	training_Tue08_19-18-21
	custom	24		87,4	3,2	0	0	5,4	2,4	7,2	2,4		5	-100	0,8	-0,3	0,3	0	1	6	90	238658	99	training_Mon12_17-06-06
	custom--human		old level custom	61,3	4,2	1,7	1,3	13,4	3,1	23,6	3,7		5	-100	0,5	-0,3	0,3	3	1	8	90	121534	67	training_Wed02_00-10-24
	custom--human			77,9	3,6	1,6	1,2	10,8	2,9	8,6	2,5		5	-100	0,8	-0,3	0,3	3	2	6	90	284628	92	training_Thu01_15-42-13
	custom--human		trained on weights above	77,5	4,5	1,1	1,1	12,5	3,6	8,9	2,6		5	-100	0,8	-0,3	0,3	3	2	8	90	157165	91	training_Sun04_19-34-55
	maze	25		81,6	3,2	0	0	10,3	2,6	8,1	2,6		5	-100	0,8	-0,3	0,3	0	1	0	90	174849	85	training_Mon12_17-06-10
	random--human	7	rf 1; camera_angle 90; dist_reward 0,3; (wrong angle)	91,8	2,4	1,3	1,1	3,4	1,6	3,5	1,8	20,8	5	-100	0,8	-0,3	0,3	3	1	8	90	115524	96	training_Tue08_14-15-26
	random--human		rf 1; camera_angle 360; dist_reward 0,3; (wrong angle)	64,3	4,5	3,6	1,7	8,3	2,7	23,8	4,2	20,9	5	-100	0,8	-0,3	0,3	3	1	8	360	123654	64	training_Tue06_17-31-04
	random--human		rf 2; dist_reward 1	93,5	2,1	1,1	1	2,9	1,5	2,5	1,4	28	5	-100	1	-1	1	0	2	8	90	90488	99	training_Thu15_11-54-00
	random--human		rf 1; camera_angle 90; dist_reward 1	59,8	5,2	3	1,6	5,2	2,3	32	5,6	19,2	5	-100	0,8	-1	1	3	1	8	90	74202	84	training_Thu15_23-23-28
DQN - LSTM	random	26		86,4	3,1	0	0	8,2	2,7	5,4	1,8		5	-100	0,8	-0,3	0,3	0	1	8	90	33876	99	training_Fri02_18-51-17
	random			83,9	3,1	0	0	7,1	2,3	9	2,7		2	-100	0,8	-0,3	0,3	0	1	8	90	11841	97	training_Mon12_17-17-55
	random--human			88,6	3	2,6	1,5	5,4	2,3	3,4	1,9	20,8	5	0	0,8	-0,3	0,3	0	1	8	90	87060	99	training_Fri09_18-05-03
	random--human			82,5	2,9	4,1	1,9	8,7	2,5	4,8	1,8	20,9	5	-100	0,8	-0,3	0,3	0	1	8	90	66117	93	training_Mon05_16-00-38
	random--human			87,2	2,7	2,8	1,6	5,3	2,1	4,7	1,9	21	5	-100	0,8	-0,3	0,3	3	1	8	90	53687	96	training_Wed07_19-05-38
DQN - GRU	random	27		91,5	2,5	0	0	4,4	1,9	4,2	1,9		5	-100	0,8	-0,3	0,3	0	1	8	90	41682	98	training_Fri02_19-11-52
	random--human			87,6	3,1	2,8	1,5	7,3	2,4	2,2	1,6	21,1	5	0	0,8	-0,3	0,3	0	1	8	90	79038	97	training_Fri09_18-05-46
	random--human			73,9	3,8	5,6	2,3	9,2	3	11,3	3,4	20,4	5	-100	0,8	-0,3	0,3	0	1	8	90	32152	83	training_Thu08_17-16-04
	random--human			90,2	2,7	2,2	1,4	5,6	2,2	2	1,3	20,8	5	-100	0,8	-0,3	0,3	3	1	8	90	72452	97	training_Thu08_17-19-05

Seite 1

Figure 15: Table containing all training results.