Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Master Thesis

Applying Deep Reinforcement Learning in the Navigation of Mobile Robots in Static and Dynamic Environments

Ronja Güldenring

6guelden@informatik.uni-hamburg.de
Study Program: Informatics
Matr.-No.: 6970986

Primary Supervisor: Prof. Dr. Jianwei Zhang
Secondary Supervisor: Dr. Norman Hendrich
Advisor: Michael Görner

Submission: April 2019

Abstract

Nowadays mobile robots operate reliably in clean static environments like industry setups, but rather fail in complex dynamic environments, like e.g. airports or shopping centers. Those environments are more crowded and narrow than industry-like environments and contain moving objects — mainly humans. Traditional navigation approaches treat all objects as static objects, resulting in a non-reasonable behavior. Mobile robots should be able to cope with dynamic objects as well as dynamic crowds.

In this thesis, local planning is realized with the state-of-the-art Deep Reinforcement Learning (DRL) approach Proximal Policy Optimization (PPO). The RL-agent is trained in a 2D-simulation environment, where it collects experiences to update the Deep Neural Network, that serves as a function approximator. First, several RL-agents are trained in a static industry-like task setup and compared to traditional navigation approaches. Second, profiting from the knowledge of the static training, agents were trained in a dynamic environment with simulated humans, behaving according to Helbing's Social Force Model. Two different behaviors worth mentioning have been evolved. One agent learned a policy, that avoids individual humans, but stops and waits if the robot faces unsolvable situations like crowds or blocked passages. The other agent learned a more aggressive policy. It can push pedestrians by driving very slowly towards them until they give way.

Zusammenfassung

Mobile Roboter werden heutzutage stabil und erfolgreich in statischen und übersichtlichen industrielle Umgebungen eingesetzt. Sobald diese Roboter in komplexeren Umgebungen mit dynamischen Hindernissen wie z.B. Flughäfen oder Einkaufszentren agieren, ist ihr Verhalten unzureichend, da traditionelle Navigationsansätze dynamische und statische Objekte gleich behandeln. Die Navigation der Roboter müsste dahin optimiert werden, dass ein adaptiveres Verhalten gegenüber dynamischen Hindernissen erreicht wird. Desweiteren ist der Roboter Situationen mit Menschenmengen ausgesetzt, die wenig Raum zum Navigieren bieten.

In dieser Masterarbeit wird das lokale Navigieren mit dem state-of-the-art Deep Reinforcement Learning (DRL) Ansatz Proximal Policy Optimization (PPO) realisiert. Der RL-Agent wird in einer 2D-Simulationsumgebung trainiert, wo dieser Erfahrungen sammelt, um das Deep Neural Network nach und nach zu optimieren. Im ersten Schritt werden verschiedene RL-Agenten in einer einfachen statischen Umgebung trainiert und mit den traditionellen Ansätzen verglichen. Aufbauend auf den Erkenntnissen vom statischen Training, werden weitere RL-Agenten in einer dynamischen Umgebung mit Menschen, die sich entsprechend Helbing's Social Force Model bewegen, trainiert. Dabei kristallisieren sich zwei relevante Verhalten heraus. Ein Agent weicht einzelnen Menschen und kleinen Gruppen aus, hält aber an und wartet, wenn es keinen Ausweg gibt wie z.B. in Menschenmengen oder bei engen Passagen, die von Menschen blockiert werden. Ein zweiter Agent hat ein aggressiveres Fahrverhalten und fährt in unlösbaren Situationen sehr langsam auf Personen zu um sie dazu zu bringen, dem Roboter Platz zu machen.

Contents

List of Figures

List of Tables

1. Introduction

This chapter provides a general introduction to the topic of the thesis. In chapter 1.1, a motivation for the use of intelligent learning algorithms during navigation of mobile robots is given. In chapter 1.2, the most relevant related work for Deep Reinforcement Learning in general and Deep Reinforcement Learning in robotic applications – especially mobile robotics – are covered. In chapter 1.3 the MiR100 robot is presented with its sensors, actors, and today's traditional navigation approach.

1.1. Motivation

The use of robots in the industry has grown drastically in the last decades and has increased efficiency and accuracy in predefined task sequences. Mobile robots in industry operate in clean static environments. They shuttle between fixed goals and avoid path blocking static obstacles, such as pallets and containers. In addition, a high demand for mobile robots in dynamic, more complex environments, like e.g. hospitals, airports, and shopping centers, is expected. Those environments contain moving objects like humans and other robots – thus a more intelligent behavior of the robot is necessary. It is expected that the robot is able to cope with dynamic obstacles and that it adapts its driving behavior to it.

Nowadays, mobile robots operate in a very satisfactory manner in industry-like static environments. Their driving behavior is very smoothly and considers new unknown objects, that are not registered by the global map. Still, there is room for improvement, especially in dynamic environments. The navigation is often not designed for dynamic environments, resulting in non-reasonable behavior regarding dynamic objects. As an example, the MiR100 robot is given, that tries to avoid the obstacles as if they are static objects, abandons and re-plans globally after failing to avoid the object. An adaptable behavior regarding moving objects is therefore heavily demanded.

Deep Reinforcement Learning (DRL) is a machine learning discipline and showed great success in controlling tasks within the last years. The DRL-agents are trained in appropriate environments and learn complex behavior by interacting with the environment on a trial-and-error basis. Deep Reinforcement Learning is heavily applied in the fields of video games and trained agents are able to play games on human-level and higher. The promising results are motivating to apply DRL in controlling tasks of robotics, although the field has more challenges due to real world constraints. Depending on the problem's

complexity and the available resources, the RL-training can last up to several days and is for that reason infeasible in the real world.

The objective of this thesis is to investigate the use of Deep Reinforcement Learning as path planning method at the MiR100 robot. The outcome provides a proof-of-concept and evaluates to what extent further investments should be made in this field.

1.2. Related Work

The popularity of Deep Reinforcement Learning (DRL) increased immensely in the past four years. It started with two success stories in 2016, that combined Deep Neural Networks with Reinforcement Learning (RL) and achieved impressive and promising results. First, the DeepMind group developed a single RL-agent, that was able to play several Atari 2600 video games on human level [3]. Based on raw input images of the game, an action is chosen among a number of discrete actions, while the score of the game serves as the reward. The applied approach is well known as Deep Q-Network (*DQN*). It uses a Deep Neural Network as function approximator in Q-learning and addresses the instability problem, that has been previously experienced by combining RL with function approximators [4]. *AlphaGo* [5] was the second success story in 2016. They developed a hybrid DRL agent, that was able to beat the world champion in the Chinese board game Go. Go provides a large search space so that it is difficult to solve artificially. In the first stage, the *AlphaGo*-agent was trained supervised by learning from recorded amateur games. In the second stage, the agent played against itself applying Reinforcement Learning.

The published approaches of the past four years can be categorized in Value-Based and Policy-Based methods. DQN is a Value-Based method: It approximates a value function, that determines the value for each action a in state s. On top of that sits a policy, that chooses the finally taken action based on the appropriate action values. DQN has been investigated intensely, resulting in many improvement proposals [6], [7], [8], [9] and [10]. Those improvements are compared in [11] and combined to a *Rainbow DQN*, that outperforms the classical DQN and their improvements. DQN and a selection of improvements are discussed in more detail in chapter 2.3.1.

In Policy-Based methods, the policy is learned directly, resulting in a more stable and smooth convergence versus a maximum. Still, they often make use of the value function to learn the policy by applying a so called Actor-Critic Architecture. The foundation of Policy-Based methods provides the REINFORCE algorithm [12] from 1992. It learns stochastic policies by applying gradient ascent during the update step. Today's state-of-the-art Policy-Based approaches are Trust Region Optimisation (TRPO) [13], Generalized Advantage Estimation (GAE) [14] Proximal Policy Optimization (PPO) [15], Deep Deterministic Policy Gradient (DDPG)[16] and Asynchronous Advantage Actor-Critic (A3C)

[17]. A selection of these approaches is further discussed in chapter 2.3.2.

Deep Reinforcement Learning is especially interesting for Robotics because it allows learning control policies from raw input data. In the last years, it has been applied to all the different robotic domains like robotic manipulation [18] [19] [20], locomotion [21] [22], self-driving cars [23] [24] and autonomous navigation. Classical navigation of autonomous robots works well in static environments. The classical approaches rely strongly on the global planner that determines a plan, leading through previously known global objects. Unseen new obstacles that are not considered by the global planner are handled by the local planner. The local planner gets — especially in environments with moving objects like other robots or humans — easily stuck and fails frequently. Applying learned policies to those situations seems promising. It is desired, that the robot learns to behave more dynamically and even adapts social behavior. Today's Reinforcement Learning approaches are very time consuming and millions of experiences need to be collected to learn complex tasks properly. As a consequence, most of the successfully robotic RL-agents are trained in a simulation environment. The training process can be automated easily and dangerous situations, caused by trial-and-error, are withheld from the real world. One common approach is to only consider 2D laser scan sensor data, because their simulation is easy and gets closest to the real world sensor data. The following paragraphs address RL solutions in the navigation of autonomous robots, that have been trained in a simulation environment.

The publications of [25] and [26] show that it is sufficient to train RL-agents in static environments with spatial laser sensors with seven to twelve data points.

Long and Fan [27] address a decentralized multi-robot scenario. The task of the robots is to drive vs. a certain goal and meanwhile avoid the other robots. The robots do not communicate directly. Instead, they decide only based on their current observation that includes the past three raw laser scans, the relative goal position and their current velocity. The action space includes continuous velocity commands that are determined by a 4-hidden-layer Neural Network. The network is trained with an extended PPO-algorithm that is adapted to parallel agents. Each agent acts according to a centralized policy and generates new experiences. The PPO-algorithm uses all samples from all robots to update the centralized policy. Furthermore, it has been trained in two stages to speed up training. Stage one includes a simple environment that has 20 robots and no static obstacles. In the second training stage, the number of robots is increased and a more complex static scenario is used. The trained agent has a remarkable success rate and its behavior is very convincing, as demonstrated in the provided video.

Coping with humans or dynamic objects, that do not behave according to the same policy, is more challenging. The encountering agents behave differently and it is not guaranteed, that they avoid in the same manner. Those dynamic environments are ad-

dressed in [28] and [29]. But also a second publication [30] builds on top of the previously explained approach dealing with multi-robot scenarios and applies it to an environment, crowded with humans. The planner developed in [27] is used as the local planner, i.e. it is supposed to follow a global plan and to react and to navigate among the crowd. Furthermore, crowded environments often cause problems in classical lidar-based localization. It fails especially if there is no distinct match between global map and lidar scan. They apply an Actor-Critic based recovery method that should navigate the robot to a close recovery point, that provides rich landmark features. By reaching one of those points, the classical localization can overtake again and re-localize.

Xie et. al. [31] proposed one of the few approaches that trains an RL-agent with raw RGB-images as input data. As the agent is trained in a simulation environment, the images are corrupted with noise and blur to be able to generalize better over real world data. Particularly, they use a Convolutional Neural Network to estimate depth data from a single RGB image. Compared to 3D sensors, the depth estimation is rather inaccurate. A DQN approach combined with two improvement strategies *dueling DQN* and *double DQN* – named D3QN – is supposed to handle those inaccurate depth informations. The network in the D3QN approach gets a stack of four depth images as input and provides velocity commands as output.

The related field of self-driving cars is also researching the usage Deep Reinforcement Learning in the controlling of cars. Most of the publications are only based on simulation environments [32], [33], while Folkers even applies the trained RL-agent to a real vehicle [34]. The RL-agent is trained with the state-of-the-art PPO-algorithm and is supposed to maneuver through a parking lot, avoiding simple static objects.

Although simulated laser scan data gets close to the real world data, the performance of an agent in the real world is mostly worse than in the simulation. It is difficult to construct realistic situations that the agent can learn from. Especially human behaviors, like movement patterns and reactions regarding the robot, are difficult to imitate. Additionally, it is desirable to use RGB- and depth-images as sensory data, because they provide more relevant information. Imitation Learning (IL) approaches like e.g. DAGGER [35] or GAIL [36] make the agent learn from expert demonstrations. IL is much more sample efficient, but it is still time-intensive if humans are supposed to generate those demonstrations. Additionally, IL is likely to overfit demonstrations during training time. Combining Reinforcement Learning and Imitation Learning could reduce the gap between real world and simulation. Besides, the advantages of both approaches can contribute: IL speeds up the training process, while RL generalizes better over all different kind of data.

Inverse Reinforcement Learning (IRL) [37] uses expert demonstration to find a reward function that explains the expert's behavior. It is assumed that the expert behaves optimally, i.e. always picks the best possible action. IRL prevents researchers from designing

and tweaking reward functions until the desired behavior is reached. In [38] and [39] IRL is applied in the context of navigation at autonomous driving.

DDPG from demonstration [40] builds on the DDPG algorithm, that stores collected experiences in a so called replay buffer and samples data from that during learning. It integrates demonstration data, by adding them to the replay buffer, so that the agent learns from both, demonstrations and self-generated experiences. They provide experiments in simulation and real world. Inserting demonstration data speeds up learning, especially when space rewards are provided. Another way to combine both techniques – IL and RL – is to pre-train the Neural Network with Imitation Learning in the first stage. In the second stage, the Reinforcement Learning takes places, building on the pre-trained network [41], [42].

1.3. The MiR100 Mobile Robot

The mobile robot MiR100 of the company Mobile Industrial Robots ApS is shown in figure 1.1. It has a rectangular footprint and a differential drive, consisting of two independently powered wheels that are positioned around the center point of the robot. For stability, another four passive wheels are positioned in the corners. It is equipped with two Sick safety Laser Scanners S300 in the front left and back right corner, a 3D-camera Intel RealSense in the front as well as multiple Ultrasonic sensors. The laser scanners cover 270 ° each, with an increment of 0.5 °, so that the whole 360 °-field around the robot is covered (see figure 1.2). They are connected to an independent safety system that is triggered if the sensors detect an obstacle in a minimum distance, depending on the speed of the robot. Besides, it has internal sensors such as a gyroscope, an accelerometer, and motor as well as wheel encoders.

Figure 1.1.: MiR100 robot of the company Mobile Industrial Robots ApS [1].

[1] accessed 2019-01-27: http://www.mobile-industrial-robots.com/de/products/mir100/

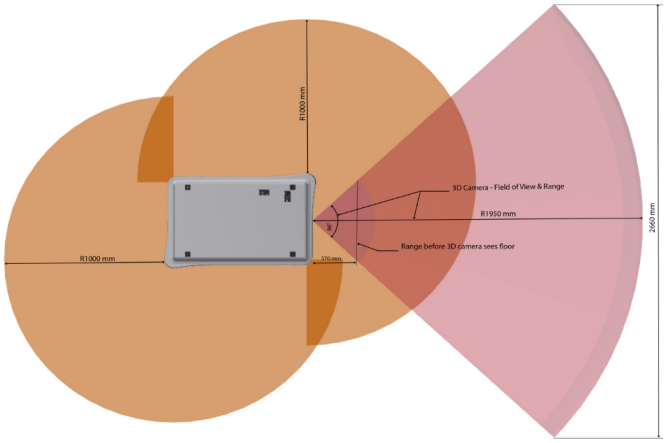


Figure 1.2.: Sensor setup of the mobile MiR100 robot. In the front left and back right corner a Sick safety Laser Scanner S300 is positioned. Together they cover the whole 360°-field of the robot (orange). In the front of the MiR100 sits a 3D-camera Intel RealSense (pink). [2]

In the following, a short introduction to the navigation software, that is applied to the MiR100, will be given. It will be referred to as traditional navigation software from here on.

1.3.1. Localization

Localization is resolved with Adaptive Monte Carlo Localization (AMCL) [43], that is based on the Particle Filter: Each particle represents a possible solution for the position of the robot. The Particle Filter iterates over the following steps.

Do

1. Sample a particle from the previous particle distribution and move it according to the physical system.
2. Place the particle in the binned state space and increase the number of non-empty bins k, if the particle was placed in an empty bin.
3. Weigh the particle according to the recent sensor data. Particles that accord strongly with the sensor data are weighted higher than particles that accord less strong.
4. Adapt the sample size bound M_x to the number of non-empty bins k. The smaller k, the more the particles agree and the smaller the final sample size n.

while $n < M_x$

After a few iterations, the particles converge towards the most probable position of the

[2] accessed 2019-01-27: MiR100 User Guide

robot. To estimate the correspondence in step 3, the sensor data will be compared to the environment of the particle, that can be retrieved from the global map. Therefore it is likely, that AMCL fails in crowded, dynamic areas. People cover significant features in the environment so that no distinct correspondence can be found.

1.3.2. Navigation

The navigation is based on the Navigation Stack in ROS [44]. It contains a global planner that determines long-distance and optimal paths from A to B based on a global costmap. A costmap is an occupancy grid map, where each cell value gives a probability of how unsafe it is to be at that position. The global costmap is mainly based on the provided global map of the world, but also considers sensor data. That means new objects that are not listed in the map of the world can be detected with sensors and integrated into the global costmap.

For the global planner, a variant of the SBPL (search-based planner) lattice planner provided by ROS is used. It applies graph-search methods to determine the global plan. First, the state space is transformed into a discrete graph, where each node represents one possible state (x, y, yaw) of the robot. In addition, the node is marked as valid, if the costmap has a low probability at that state, else invalid. Discretizing the state space makes the path-finding process more efficient, but can also lead to unusual looking paths. For example, if the robot has to drive along a corridor, that has a certain angle to the global coordinate system, the path can have a zick-zack pattern. As search-algorithm, the ARA*-algorithm [45] is applied. It applies the A*-algorithm with weighted heuristic, that produces a sub-optimal path. The parameter ϵ defines the extent of sub-optimality: The length of the sub-optimal path is not larger than ϵ times the length of the optimal path. ARA* executes A* several times while decreasing ϵ and reusing the information from the previously produced path. Like this, it guarantees a sub-optimal path in a short amount of time and if a certain time threshold is not yet exceeded, it can spend the remaining time to improve that path.

The local planner solves short-distance path planning and re-plans on an on-going basis during the navigation along the global plan. The local planner follows the global plan as well as avoids local obstacles, that are detected on the global path. Those objects were not present in the global costmap and are therefore not considered in the global plan. The local planner is supposed to avoid local obstacles and to find back to the global plan afterward. The local planner takes a local costmap into consideration, that is low in size and only represents the area around the robot. It is regularly updated according to the input sensor data.

The local planner is realized as a mixture of the pure pursuit and the Vector Field Histogram (VFH*) motion planning approach [46]. The pure pursuit takes care of basic global path following while the VFH* avoids local objects on that path. The VFH* is

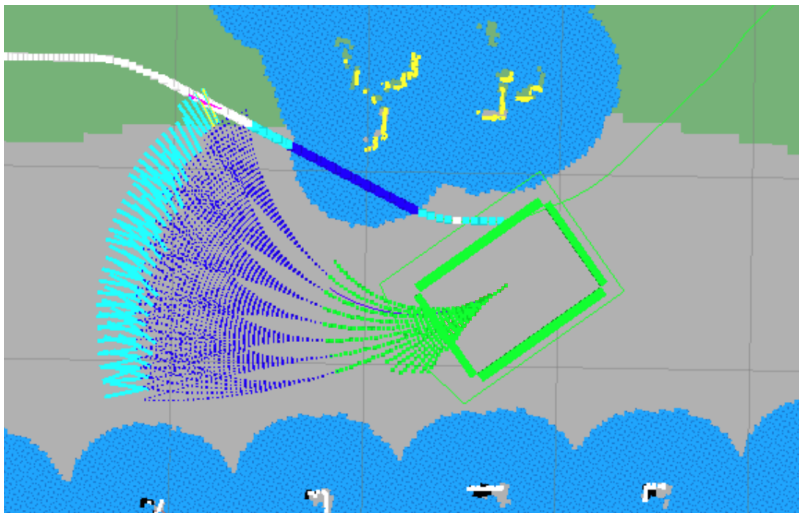


Figure 1.3.: Two-step VFH*: The green fan is the first VFH* expansion, arising from the robot (green rectangle). On each valid arc a second finer VFH* expansion (dark blue) is added. The cyan lines are simple straight extensions of each blue arc. Finally, the expansion that leads closest back to the path is chosen.

triggered, as soon as an obstacle is closer than a certain distance threshold and it uses the local costmap to determine openings with the lowest cost that are passable for the robot. VFH* allows the robot only to move on a number of discrete arcs. Local obstacles block all arcs with trajectories that lead through the direction of the obstacle. Moreover, a variant of the VFH* is applied in the MiR100 robot, that is further called two-step VFH* and is illustrated in figure 1.3. In the first step, a discrete number of possible arcs (green) is spread out, arising from the robot. Each valid arc (i.e. does not collide with an inflated light blue obstacle) is extended with a second finer VFH* expansion (dark blue). Finally, the expansion with the closest distance back to the path is chosen.

In the following two variants of the traditional navigation approach will be referred to: 2S-VFH*-R and 2S-VFH*. 2S-VFH*-R is closest to the original software. It applies the two-step VFH* and pure pursuit for the local planner and adds a recovery method on top. The recovery method takes place if the robot gets stuck and is not able to solve the situation with the normal local planner. Note that the recovery method uses global re-planning, while new local objects are considered during re-planning. In this thesis, the consideration of local objects is disabled to isolate the performance of the local planner. 2S-VFH* isolates the local planner even more strongly by disabling the recovery methods completely. That means 2S-VFH* does not allow global re-planning at all.

2. Fundamentals

This chapter summarizes the relevant fundamentals for further understanding of the topic of the thesis. In chapter 2.1, basic concepts of Artificial Neural Networks are presented, while a special focus is set on Convolutional Neural Networks. Chapter 2.2 covers the fundamentals of traditional Reinforcement Learning. Basic algorithms like Monte Carlo and Temporal Difference Methods are presented. Finally, the knowledge of 2.1 and 2.2 is combined in chapter 2.3 about Deep Reinforcement Learning. Three selected state-of-the-art DRL-algorithms are discussed: Deep Q-Network(DQN), Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO).

2.1. Artificial Neural Networks

The artificial neuron is inspired by the biological neuron from the animal brain. The human brain consists of approximately 100 billion neurons to process sensory information like vision, touch, and acoustics. A single neuron has several inputs – called dendrites – coming from other preceding neurons. The neuron processes the inputs and if a certain action potential is reached, the neuron "fires" through its single output – called axon. The output of the axon will be forwarded to all following connected neurons.

An artificial neuron (also called perceptron) models the biological neuron in a simplified way. Each artificial neuron has n input connections. The neuron processes the inputs by taking the weighted sum, adding a bias b and applying an activation function: $f(\sum^n \theta_i x_i + b)$. Figure 2.1 illustrates the parallels of a biological and an artificial neuron.

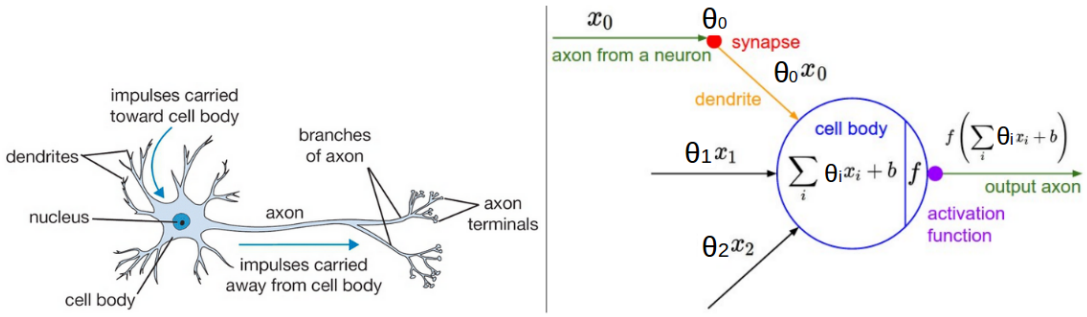


Figure 2.1.: Biological neuron (left) vs. artificial neuron (right). The artificial neuron models the dendrites as weighted inputs and processes the sum through an activation function $f(\sum \theta_i x_i + b)$. [47]

Neural networks approximate a non-linear function $f^*(x)$ by composing multiple neu-

rons in a chain. The parameter set θ that contains all weights θ_i of all neurons needs to be adjusted in such a manner that the Neural Network results in the best possible function approximation. The process of finding a good parameter set θ is called learning. A feedforward Neural Network organizes the artificial neurons in different layers. The neurons in the layers are connected to each other in a forwarding manner. There are no connections that are fed back to previous neurons. Each network has one input layer, that processes the raw input data, and one output layer, that contains the approximated result. In between those two layers can be one or more hidden layers where the relevant computing is happening. Figure 2.2 shows a Fully-connected, feedforward Deep Neural Network. It is fully-connected because all neurons of the outgoing layer are connected to all neurons of the incoming layer. This is not absolutely necessary. [48]

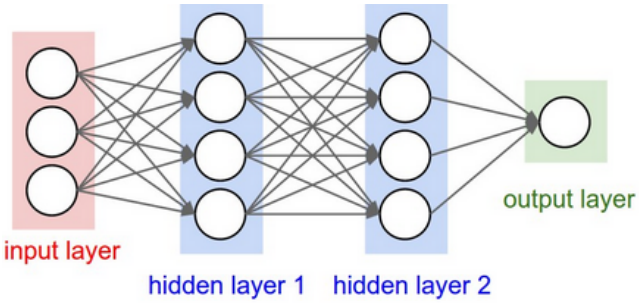


Figure 2.2.: Fully-connected, feedforward, Deep Neural Network with one input, one output and two hidden layers. [47]

2.1.1. Learning Process

The goal of the learning process is to find a parameter set θ that results in the best possible function approximation. In supervised learning, the true output Y of a certain input X is given and can be used to update the parameters θ. It is an iterative process, consisting of the following steps.

1. **Forward Pass**. The input X is forwarded through the network and one gets the predicted output $Y_{pred} = f(X, \theta)$.

2. **Loss**. The predicted output Y_{pred} is compared to the true output Y by computing the loss $L(\theta)$. The choice of the loss function depends on the learning task. In the following, relevant loss functions are listed.

   – Mean-square-error [49]. It is widely used and computes the L2-distance between Y_{pred} and Y.

   $$L(\theta) = \frac{1}{2}\sum_{i=1}^{n}(Y_i - Y_{pred,i})^2 \tag{2.1}$$

   – Logistic Loss function [49]. The logistic loss function punishes points that are classified correctly with a low confidence. Still wrongly classified samples are

punished more strongly.

$$L(\theta) = \sum_{i=1}^{n} \log(1 + \exp(-Y_i \cdot Y_{pred,i})) \tag{2.2}$$

3. **Back-propagation**. The global gradient of loss $\nabla L(\theta)$ is computed and back-propagated through the network. The back-propagation algorithm, introduced by [50], provides local gradient of loss to all hidden neurons. The algorithms underlying concept is the chain rule. The chain rule computes derivatives of composed function by multiplying local derivatives. Given the function $y = g(x)$ and $z = f(g(x))$, the derivative $\frac{\partial z}{\partial x}$ can be computed according to equation 2.3. [48]

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{2.3}$$

The chain rule is used to propagate the global gradient loss $\frac{\partial L(\theta)}{\partial \theta}$ back through the network. It flows in the opposite direction of the forward pass. Figure 2.3 shows a neuron with the function $z = f(x, y, \theta)$. Its local derivatives are $\frac{\partial z}{\partial x}$, $\frac{\partial z}{\partial y}$ and can be determined during the forward pass. The local gradient of loss is computed during back-propagation by multiplying the local derivative with the local gradient of loss of the connected neuron of the next layer $\frac{\partial L}{\partial z}$. As result, one gets a local gradient of loss for each input of the neuron: $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial y}$. They will be further back-propagated to the other preceding neurons. In case a neuron is connected to several neurons in the next layer, the gradients are simply added up.[47]

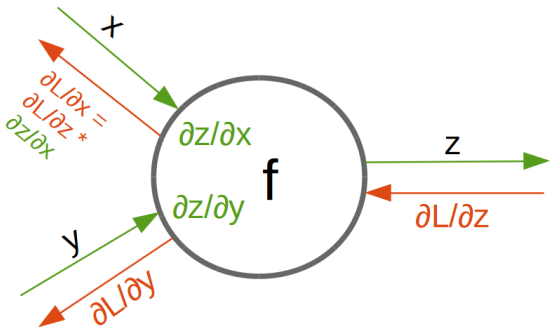


Figure 2.3.: The local gradient of loss for the input x and y can be computed by applying the chain rule. The local gradient of loss of the next neuron $\frac{\partial L}{\partial z}$ is multiplied with the local derivatives $\frac{\partial z}{\partial x}$, $\frac{\partial z}{\partial y}$. As result, one gets the local gradients of loss $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial y}$.

4. **Update**. The weights of all neurons are updated. A common optimizer is stochastic gradient descent (SGD). It combines Batch Learning from section 2.1.4 with gra-

dient descent. Gradient descent changes the weights in the negative direction of the gradient of loss so that the function approximation approaches closer to the minimum in each iteration. It is expected that after a number of iterations, a local minimum is reached. Equation 2.4 shows the corresponding update rule of gradient descent. α is the learning rate parameter that defines how quickly the minimum should be approached. If the learning rate α is too high, there is a risk that the minimum cannot be reached, because the taken steps are too big and will be overshot.

$$\theta_i \leftarrow \theta_i + \alpha \nabla_\theta L(\theta) \tag{2.4}$$

2.1.2. Regularization

The final goal of learning is that the function approximation generalizes over the presented data, i.e., it shows similar performance on new, unseen data. A good compromise between under- and overfitting needs to be found. Underfitting occurs when the approximated function is too simple. It generalizes on the data, but the prediction error is too high for all data points. Underfitting often results from insufficient, small training data sets with a lack of diversity. Overfitting describes the contrary: the approximated function is too complex. It represents the training points really well, but unseen points are predicted poorly.

Regularization is an approach to prevent overfitting. The loss function will be extended with a regularization term $\Omega(\theta)$, that tries to keep the approximated function as simple as possible. Equation 2.5 shows the extended regularized objective loss function $\tilde{L}$. λ is the regularization factor, that weighs the regularization term $\Omega(\theta)$ against the original loss function. If $\lambda = 0$, there is no regularization. [51] [48]

$$\tilde{L}(\theta) = L(\theta, Y, Y_{pred}) + \lambda \Omega(\theta) \tag{2.5}$$

Equation 2.6 shows the L^2-regularization. It adds up the squared sum of the weights θ to the objective loss function. The regularization method aims to keep the weights small. [48]

$$\tilde{L}(\theta) = L(\theta, Y, Y_{pred}) + \lambda \frac{1}{2} ||\theta||^2 \tag{2.6}$$

Equation 2.7 shows the L^1-regularization. In contrary to the L^2-regularization, the weights are only punished linearly. Large weights are not punished stronger, so that it is possible to have large weights if at the same time several small weights get zero. The regularization method leads to a sparser solution. [48]

$$\tilde{L}(\theta) = L(\theta, Y, Y_{pred}) + \lambda \frac{1}{2} ||\theta|| \tag{2.7}$$

2.1.3. Activation Functions

There are three different commonly used activation functions: sigmoid, tanh and ReLU, that will be discussed in this chapter.

The **sigmoid** function is shown in equation 2.8. It maps the input value x between the range of 0 and 1. Large negative values become 0 and large positive values become 1.

$$\text{sigm}(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

The sigmoid function is less used because it has some crucial disadvantages. If the neuron's output saturates at 0 or 1, the local gradient gets almost zero. At back-propagation, the global gradient will be multiplied with the local gradients, so that the product ends up zero as well. Eventually, the weights will not change, and the network is not capable of learning effectively. It is especially problematic if the network is initialized with weights that directly end up in saturating outputs. Another disadvantage is that the output of the sigmoid function is not zero-centered. [47]

The **tanh** function is shown in equation 2.9. It zero-centers the sigmoid function. Still the disadvantage of saturation remains.

$$\tanh(x) = 2 \cdot \text{sigm}(2x) - 1 \tag{2.9}$$

The **ReLU** function is the most popular function and shown in equation 2.10. It does not allow the output to get smaller than zero. It has a non-saturating form and allows the gradient to converge faster during training. Another advantage is that it is a simple function with a small computational cost. [47]

$$\text{ReLU}(x) = \max(0, x) \tag{2.10}$$

2.1.4. Batch Learning and Normalization

The idea of **Batch Learning** is to process a set of m training samples (mini-batches) instead of just a single training example. The gradient is averaged over all m processed training examples. This can lead to a more accurate gradient with less variance, resulting in reduced training time. Besides, Batch Learning speeds up the training when using a graphics processing unit (GPU). All training samples can be processed independently, i.e., in parallel. [47]

Batch normalization [52] applies normalization over the whole batch by zero-centering and rescaling the data. It is expected that the mean of the normalized data is close to zero and the variance is close to one. In a batch H with m samples, each value h_i is

normalized over the whole batch according to equation 2.12. The mean μ (equation 2.13) and variance σ (equation 2.14) is computed element-wise for each spatial position across the whole batch. $\delta > 0$ is a small value to avoid division by zero. The normalized value $y_i^{'}$ will be further processed by equation 2.11. γ_{bn} and α_{bn} are parameters of the batch normalization (bn) layer and are learned along with the original parameter set θ of the Neural Network. The additional learning dynamics increase the network's expressive power. [48]

$$y_i = \gamma_{bn} y_i^{'} + \beta_{bn} \tag{2.11}$$

$$y_i^{'} = \frac{h_i - \mu}{\sigma} \tag{2.12}$$

$$\text{with } \mu = \frac{1}{m} \sum_{i=1}^{m} h_i \tag{2.13}$$

$$\text{with } \sigma = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (h_i - \mu)^2 + \delta} \tag{2.14}$$

Applying the batch normalization to the input data as well as the output of any hidden layers leads to a regularizing effect during learning and prevents overfitting. Another advantage is that it speeds up the training time. It has to be noted that batch normalization is only applicable, if the exact position of the features is not relevant, but rather whether the feature exists in the input.

2.1.5. Convolutional Neural Networks

Convolutional Neural Networks are inspired by the receptive field in the brain, that processes sensor input data and is sensitive to certain stimuli, e.g., edges in the visual system. They handle large input data efficiently and are consequently widely used in state-of-the-art approaches in the fields of Computer Vision, like e.g. object detection [53] [54] [55] or image segmentation [56] [57].

Figure 2.4 shows the LeNet-5 [1] that recognizes digits in images. It provides a typical architecture of Convolutional Neural Networks, consisting of stacks of Convolutional Layers, followed by a subsampling Pooling Layer. The final hidden layers of the network are normally fully-connected to compute the final low-dimensional output of the network. It can be assumed, that in the early stages of the network low-level features like edges and corners are learned, while in later layers those features are combined to high-level features.

Convolutional Layer

The Convolutional Layer builds on the discrete convolution operation, that applies a square filter f of the size $[m \times m]$ with $m = 2k + 1$ to an input matrix g at position $[x, y]$ by computing the dot product. The discrete convolution operation is shown in equation

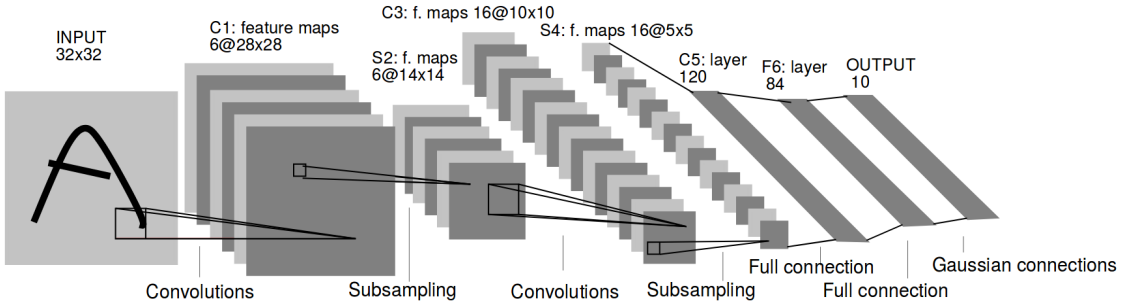


Figure 2.4.: To illustrate a typical architecture for Convolutional Neural Network, the LeNet-5 [1] is presented. It recognizes digits on images. The input image is processed by two stacks of Convolutional Layers, each followed by a subsampling Pooling Layer. The last three layers are fully-connected to map the high-level features to the final digit classification.

2.15. [48]

$$h[x,y] = f * g[x,y] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} f[u,v]g[x-u,y-v] \tag{2.15}$$

One neuron in a Convolutional Layer is represented by a filter of the size $[m \times m \times d]$. The weights of the neuron are the filter values as well as a bias b. The filter will be shifted over the input matrix g with depth d and produces an output $h[x,y]$ for each position $[x,y]$. Note that the input matrix g and the filter f have the same depth d. To produce an output h that has the same size as the input g, zero padding can be applied. Zero padding extends the input matrix g by $(m-1)/2$ rows or columns with zero-values on each side. A set of different filters (= neurons) forms the Convolutional Layer. All filters have the same size, but different filter values, and are all applied to the same input g, producing a so called *activation map*. The final output of the layer is a stack of all activation maps.

It is common to shift the filter with a constant stride S over the input so that every Sth position of the input will be convolved. It results in a reduction of the data size in the next layer.

Compared to a Fully-connected Layer, the number of weights in a Convolutional Layer is kept small and the computation in the layer is more efficient. Generally, the filter size is kept much smaller than the input data, leading to the detection of small, low-level features. Small filters require fewer parameters as well as fewer operations in the convolution operation. In addition, the filter is applied to the different locations of the input due to filter shifting. The intuition behind this is that the same features can appear at different locations of the input and can be detected by the same neuron. For this reason, feature detection with Convolutional Layers is invariant in translation.[48]

Pooling Layer

The Pooling Layer has a subsampling function in the spatial dimensions width and height by applying a downsampling filter to the input. Common pooling filters are max- and

average-pooling. At max-pooling, a filter of size $[m \times m]$ slides over the input and only the maximum value remains in the output. Figure 2.5 provides an example: A $[2 \times 2]$-filter is shifted over the 2-dimensional input with a stride of 2. The resulting output size is a quarter of the input size. In average-pooling the average of each position $[x, y]$ and its neighbors is computed by the filter.

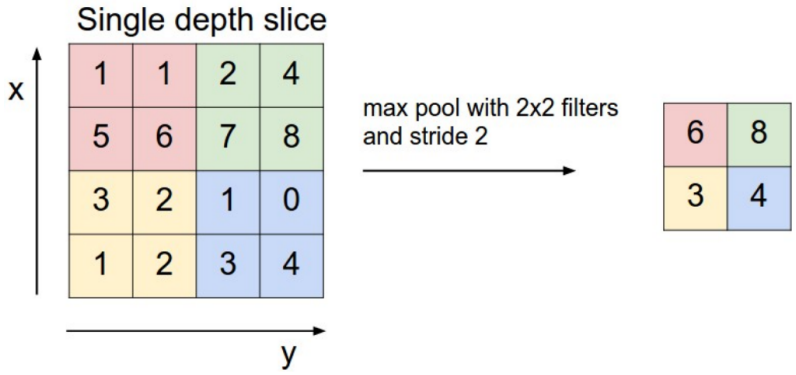


Figure 2.5.: A max-pooling filter of size $[2 \times 2]$ with a stride of 2 is applied to a 2-dimensional input with the size $[4 \times 4]$. The maximum value remains in the output and the output size is reduced by 4. [47]

Pooling reduces the data size, leading to an improvement of the network efficiency. It is useful if the exact feature position is not relevant but rather whether a certain feature exists in the input at all. [48]

2.2. Reinforcement Learning (RL)

In Reinforcement Learning, an agent is supposed to learn a specific behavior by trial-and-error. The agent interacts with its environment to collect experiences. Figure 2.6 illustrates the basic concept behind Reinforcement Learning. At each time step $t = 1, 2, 3, ...$ the agent is in a certain state $s_t \in \mathcal{S}$ and takes one of the possible available actions $a_t \in \mathcal{A}(s)$. The action changes the environment and the agent ends up in a new state s_{t+1}. Furthermore, it receives a reward R_{t+1} from the environment that serves as a feedback about how good it was to take action a_t in state s_t.

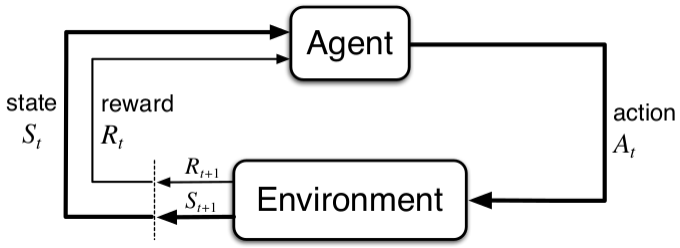


Figure 2.6.: General idea of Reinforcement Learning: The agent interacts with the environment to learn from experiences. At each time step t the agent is in a certain state $s_t \in \mathcal{S}$ and takes action $a_t \in \mathcal{A}(s)$. As result, it switches to a new state s_{t+1} and receives a reward R_{t+1}. [58]

This chapter gives an introduction to the basic concepts of classical Reinforcement Learning and serves as the foundation for the advanced Deep Reinforcement Learning approaches, discussed in chapter 2.3. The whole chapter is based on the well-known book *Reinforcement Learning - An Introduction* from Sutton and Barto [58].

2.2.1. Markov Decision Process

The *Markov Assumption* assumes an independence of past and future states, meaning that the state and the behavior of the environment at time step t are not ninfluenced by the past agent-environment interactions $a_1, ..., a_{t-1}$. If the RL-task can fulfill the *Markov Assumption*, it can be formulated as five-tuple *Markov Decision Process* $(\mathcal{S}, \mathcal{A}, P^a_{s,s'}, R^a_{s,s'}, \gamma)$.

- Set of states $\mathcal{S}$
- Set of actions $\mathcal{A}$. $\mathcal{A}(s)$ is the set of available actions in state s.
- Transition probabilities $P^a_{s,s'} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \rightarrow [0, 1]$. It is the probability of the transition from s to s' when taking action a in state s at time step t.
- Reward probabilities $R^a_{s,s'} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \rightarrow IR$. It defines the immediate reward the agent receives after the transition from s to s'.
- Discount factor $\gamma \in [0, 1]$ for computing the *discounted expected* return.

The Markov Decision Process (MDP) is finite if the set of states $\mathcal{S}$ and actions $\mathcal{A}$ is finite.

2.2.2. Discounted Expected Reward

To train an effective agent, its goal should be to maximize the reward in the long run instead of just caring about the immediate return. Consider the environment of figure 2.7 with six different rooms. The agent starts in room one and the goal is to end up in room five. The immediate reward is the negative distance of the agent to room five. If the agent just cares about maximizing the immediate reward, it changes to room three because the immediate reward is higher than in room one or two. Unfortunately, it is not able to reach

room five from there and it remains in room three. It will never reach the final goal. On the contrary, if the agent's effort is to maximize the *expected return*, it accepts to receive a lower immediate reward in room two, followed by higher immediate rewards in room four, six and five. The sum of immediate rewards is maximized.

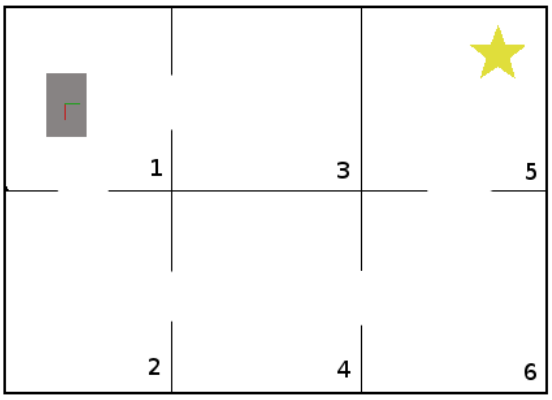


Figure 2.7.: Immediate reward vs. expected return. Suppose the agents start in room one, its goal is to end up in room five and its reward is the negative distance between the agent and room five. If the agent only cares about the immediate reward, it would switch directly to room three and never reach room five. If the goal of the agent is to maximize the expected return, it will first accept a lower immediate reward in room two, followed by higher rewards in room four, six and five.

The *discounted expected return* is the cumulative sum of possible future rewards and can be found in equation 2.16. The discount factor $\gamma \in [0,1]$ rates the future rewards and defines how far in the future rewards are considered. If $\gamma = 1$ all rewards of the future are considered with the same weight. If $\gamma = 0$ just the immediate reward is taken into account.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.16}$$

It can be differentiated between *episodic* and *continuous tasks*.

- Episodic: The training procedure can be divided into episodes. When the agent reaches a terminal state, the episode is over, the scene will be reset and the agent will restart in the next episode. The terminal state has an immediate reward of 0 and can be reached in T finite time steps.
- Continuous: The problem cannot be formulated in episodes and is a continuous ongoing problem so that $T = \infty$.

2.2.3. Policy and Value Functions

A policy π tells the agent how to behave. It models a probability distribution $\pi(a|s)$ over the number of available actions $a \in \mathcal{A}(s)$ for each state s, i.e. $\pi(A_t|S_t)$ is the probability

of taking action A_t in state S_t. The agent samples its next action from that probability distribution $\pi(a|s)$.

The value function $v_\pi(s)$ is an estimate of how good it is for the agent to be in state s. $v_\pi(s)$ is the expected discounted return of state s, if the agent behaves according to policy π (see equation 2.17).

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \text{ for all } s \in \mathcal{S} \tag{2.17}$$

Above all, equation 2.17 can be formulated recursively. The recursive form is called Bellman Equation for v_π and is shown in equation 2.18. In the Bellman equation, the value of state s is only dependent on the next possible states s' while each state is weighted by the transition probability $P^a_{s,s'}$. Many Reinforcement Learning solutions approximate the optimal Bellman equation by approximating the value of the next states.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma v_\pi(s')] \end{aligned} \tag{2.18}$$

The action-value function $q_\pi(s, a)$ is an estimate of how good it is to take action a in state s. $q_\pi(s, a)$ is the expected return of taking action a in state s and thereafter behaving according to policy π.

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \\ &\text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \end{aligned} \tag{2.19}$$

Reinforcement Learning aims to find an optimal policy π_*. A policy is better than another policy $\pi \geq \pi'$ if the value function of the new policy is better $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. If the state-value function is optimal, an optimal policy was used by the agent. It is possible that there are multiple optimal policies, that lead to the same optimal state-value function. The optimal state-value function v_* can be defined as followed:

$$v_*(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \tag{2.20}$$

Furthermore, optimal policies result in the optimal action-value function q_*.

$$q_*(s,a) = \max_\pi q_\pi(s,a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s) \tag{2.21}$$

$$= \mathbb{E}[R_{t+1} + \gamma v_*(s') | S_t = s, A_t = a] \tag{2.22}$$

Finally, the *Bellman optimality equation* in equation 2.23 can be derived from the previously introduced equations.

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(s') | S_t = s, A_t = a] \\ &= \max_a \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma v_*(s')] \\ &= \max_a \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma \max_{a'} q_{\pi_*}(s',a')] \end{aligned} \tag{2.23}$$

2.2.4. Monte Carlo Method

The Monte Carlo method is an approach that aims to solve reinforcement problems with episodic tasks, where no model of the environment exists. It is an iterative approach and converges with the increasing number of episodes towards the optimal policy.
There is a Q-table that holds the action-value for each possible state-action pair. The value is the average over all returns, that has been collected in all episodes. An entry of the table is updated each time a state-action pair is met by the agent. The update equation is shown in equation 2.24, where $N(S_t, A_t)$ is the number of visits of the state-action pair (S_t, A_t).

$$Q(S_t, A_t) = Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t)) \tag{2.24}$$

The different returns can be weighted by α. Instead of taking the true average, recent returns can be weighted more or less strongly (see equation 2.25).

$$\begin{aligned} Q(S_t, A_t) &= Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)) \\ &= (1-\alpha)Q(S_t, A_t) + \alpha G_t \end{aligned} \tag{2.25}$$

The Monte Carlo method iterates over episodes. During the *Evaluation step*, the agent acts according to policy π for one episode. When the episode is finished, the agent collected a sequence of experiences $S_1, A_1, R_2, S_2, ..., S_T$ and can update the Q-table according to it. For each state-action pair (S_t, A_t) in the sequence, the expected return G_t is retrieved and the corresponding entry in the Q-table is updated according to equation 2.25. During the *Improvement step*, the policy π will be updated according to the recent Q-table. In the next iteration, the *Evaluation step* is performed with the new, updated policy. Figure 2.8

illustrates the concept of the Monte Carlo method.

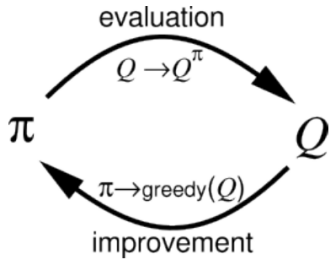


Figure 2.8.: Concept of the Monte Carlo method. **Evaluation**: The agent experiences one episode and updates all visited state-action pairs (S_t, A_t) in the Q-table with the expected return G_t according to equation 2.25. **Improvement**: The policy π is updated according to the new Q-table. [58]

The most conservative policy update is the so called *greedy policy*. The agent always chooses the action with the maximum action-value. Greedy actions exploit the current knowledge; the agent can get stuck and end up in a non-optimal policy. To avoid that situation, explorative actions should be taken from time to time. A policy is called ϵ-*greedy policy*, if the greedy action is taken with a probability of $(1 - \epsilon)$ and a random action is taken with a probability of ϵ to explore the action space. The equation of the ϵ-*greedy policy* is shown in equation 2.26.

$$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}}\, Q(s,a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{2.26}$$

ϵ is a value between 0 and 1 and weighs the relation between exploitation and exploration. It stands to reason to explore the action space stronger in the beginning because the agent does not have reliable knowledge yet. The more experiences the agent gains, the more learned knowledge should be exploit. An ϵ that decreases over time to a minimum value of ϵ_{min} models that behavior.

2.2.5. Temporal Difference Methods

The advantage of Temporal Difference (TD) methods is that they are applicable to online learning and continuous RL-tasks. The policy is updated in each time step t so that there is no need of completed episodes. Moreover, no model of the environment is required. Experiments showed that TD methods tend to converge more quickly than the Monte Carlo method.

In TD-methods, the true expected return G_t is not available and is approximated with the *TD-target*. The TD-target can also be seen as an approximation of the Bellman equation 2.23. The TD-target is determined by taking the sum of the immediate return and the discounted expected value of the next state: $R_{t+1} + \gamma V(S_{t+1})$. The pseudo-code of

Data: policy π, $\alpha \in (0, 1]$
Initialize V(s), for all $s \in \mathcal{S}$ arbitrarily, except $V(terminal) = 0$;
for *each episode* **do**
  Initialize S_t **do**
    $A_t \leftarrow$ action given by π for S_t;
    Take action A_t, observe R_{t+1} and S_{t+1};
    $V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD-target}} - V(S_t)]}_{\text{TD-error}}$;
    $S_t \leftarrow S_{t+1}$
  while *S is not terminal*;
end

Algorithm 1: General algorithm of TD methods: The expected return G_t is approximated with the sum of the immediate return and the discounted expected value of the next state (TD-target): $R_{t+1} + \gamma V(S_{t+1})$. [58]

the general concept of TD-methods can be found in algorithm 1. In each iteration of each episode, an action A_t is retrieved from policy π for a given State S_t. The action A_t is executed and the agent is rewarded with R_{t+1} and transitioned to the next state S_{t+1}. Finally, the value-table V is updated with the *TD-error*, which is the difference of the TD-target and $V(S_t)$.

Sarsa

Sarsa is an on-policy TD control method and stands for $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. Those parameters are needed in each iteration to update the action-value $Q(s, a)$ of the state-action (S_t, A_t) in the Q-table. The agent takes action a in state s and receives the immediate reward R_{t+1}. Afterwards policy π is used to determine action A_{t+1}, that will be taken in the next state. The sum of the immediate return R_{t+1} and the Q-value of the next state-action pair S_{t+1}, A_{t+1} represents the TD-target and approximates the true expected return G_t. The action-value update rule is shown in equation 2.27.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.27}$$

Q-Learning

Q-Learning is a popular off-policy TD control method, i.e. the policy π is not used for updating the Q-table. The main difference to Sarsa is, that instead of using the action-value of the next state-action pair $Q(S_{t+1}, A_{t+1})$, only the maximum Q-value of the next state S_{t+1} is taken. The action-value update rule of Q-learning is shown in equation 2.28.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.28}$$

2.3. Deep Reinforcement Learning (DRL)

In the classical Reinforcement Learning, discussed in chapter 2.2, all approaches have a tabular setting, leading to essential disadvantages. On the one hand, the usage of a table limits the classical approaches to tasks with a low number of states and actions. In real world problems, the state space can quickly get large. It is not feasible to visit all possible states to retrieve the value for all action-state pairs. Furthermore, the size of the table is limited due to memory constraints in hardware. On the other hand, knowledge about similar states is not shared. This could lead to better representation and lower training times.

To overcome the mentioned restrictions, a common approach is to replace the value table with a Deep Neural Network as function approximator. Their ability to approximate non-linear functions and to extract relevant features from raw inputs makes it possible to generalize over unseen states.

2.3.1. Value-Based Methods

Value-Based Methods build on Temporal Difference Methods from classical Reinforcement Learning discussed in chapter 2.2.5. The idea is to replace the value table one-to-one and to approximate it with a Deep Neural Network. The network's output provides probabilities for each possible action. A traditional policy lays on top of the network output to choose the final action (e.g., ϵ-greedy policy).

Deep Q-Network (DQN)

Combining Q-Learning with non-linear function approximation has been investigated in the past decades and did not lead to great success because of unstable learning. In 2015, the DeepMind group [3] presented an approach – called deep Q-Network (DQN) – that showed a great success. They combined the model-free, off-policy Q-Learning with Deep Neural Networks. As input data, high-dimensional raw sensory input with no previously hand-crafted features are used. This end-to-end architecture allows the network to extract relevant features by itself. The output of the Q-network provides a probability distribution over all possible discrete actions. This allows one to determine the best action for a given state with a single forward pass. Particularly, the problem of unstable learning has been improved with two additional mechanisms, called *experienced replay* and *frozen target network*, that will be further explained in the following.

The idea of *experienced replay* is to store the agent's experiences $S_t, A_t, R_{t+1}, S_{t+1}$ in a buffer that can hold n_{buf} experiences in total. In each training step, a batch of experiences is uniformly sampled from the buffer and fed to the network. *Experienced replay* removes the correlations in the data sequences and feeds the network with independent data. It also

ensures that old experiences are repeated from time to time. It has a smoothing effect over changes in the data distribution.

In DQN the network is updated according to the loss function from equation 2.29. The loss function is computed by taking the squared TD-error.

$$L_i(\theta_i) = \hat{\mathbb{E}}_t[(R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \theta_i^-) - Q(S_t, A_t, \theta_i))^2] \tag{2.29}$$

In equation 2.29, the second mechanism *frozen target network* is introduced as well. Two networks with the same structure, but different weights are used: θ for the Q-network and θ^- for the target network. The Q-network is regularly updated according to the loss function from equation 2.29, while the target network is updated by copying the parameters of the Q-network to the target network $\theta^- = \theta$ every C time steps. Thus, the weights of the target network θ^- are held frozen for C time steps. It smooths oscillating policies and leads to more stabilized learning.

Improvements of DQN

After the publication of the successful DQN approach, it has been investigated widely and several publications with improvements followed. Rainbow [11] compares all relevant improvements with the original DQN approach and even applies a combination of all improvements called *Rainbow DQN*. In the following paragraphs, three major improvements are shortly and intuitively introduced.

Double DQN. Double DQN [7] tries to handle the overestimation of Q-values. Especially in early stages of the learning process, it is likely that wrong actions have the highest Q-value. Using two different Q-networks θ and θ^- for estimating the TD-target results in more robust learning. As DQN already holds two different networks, Double DQN can easily make usage of them by modifying the loss function from equation 2.29 to equation 2.30.

$$L_i(\theta_i) = \hat{\mathbb{E}}_t[(R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a, \theta_i), \theta_i^-) - Q(S_t, A_t, \theta_i))^2] \tag{2.30}$$

Prioritized Experienced Replay. The idea of Prioritized Experienced Replay [8] is to prioritize experiences, that contain more important information than others. Each experience is stored with an additional priority value, so that experiences with higher priority have a higher sampling probability and have the chance to remain longer in the buffer than others. As importance measure, the TD-error can be used. It is expected that if the

TD-error is high, the agent can learn more from the corresponding experience, because the agent behaved better or worse than expected. Prioritized Experienced Replay was able to speed up the learning process by a factor of two.

Dueling DQN. Dueling DQN [6] proposes a new network architecture shown in figure 2.9. They decouple the Q-value estimation in two streams: One stream estimates how good it is to be in state $V(s)$ and the other stream estimates the advantage of taking action an in that state $Adv(s,a)$. Both streams build on the same convolutional basis and are finally fused together to represent the final Q-value. The outcome of that architecture is that the state value can be learned separately, without getting confused by the influence of the action advantage. This leads especially to the identification of state information where actions have no effect on.

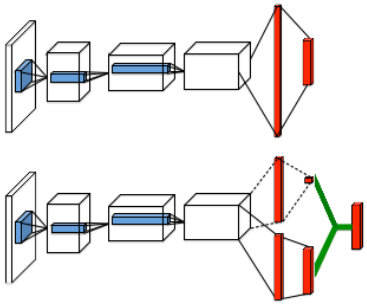


Figure 2.9.: Original DQN network architecture (top) vs. dueling network architecture (bottom). On top of the Convolutional Layers two streams are added, that estimate the state value $V(s)$ and the action advantage $Adv(s,a)$ separately. The final output is the aggregation of both streams and represents the Q-value. [6]

2.3.2. Policy Gradient Methods

Policy Gradient Methods optimize the policy $\pi(a|s,\theta)$ directly instead of learning a value function and choosing the actions based on it (e.g ϵ-greedy policy). The quality of each policy can be measured by the policy's performance measure $J(\theta)$. The objective function of Policy Gradient Methods in equation 2.31 maximizes the scalar value $J(\theta)$.

$$\theta^* = \underset{\theta}{\text{argmax}} J(\theta) \tag{2.31}$$

The policy's parameter θ are updated via gradient ascent. Gradient ascent is the inverse of gradient descent and updates the parameters θ_t in the positive direction of the gradient of the policy's performance measure $\nabla_\theta J(\theta)$ (see equation 2.32). Furthermore, the learning rate α defines, how strongly one steps in the gradients direction.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.32}$$

One advantage of Policy Gradient Methods is their stable convergence property because the policy is updated directly and thus improves smoothly at each time step. Value-based methods update the value function at each time step. A small change in the value function can lead to a drastic change in the policy output. Hence, value-based methods often deal with big oscillations during training. Especially, Policy Gradient Methods can deal with infinite and continuous action spaces. Instead of determining a Q-value for each possible discrete action, the action can be estimated directly, e.g. the speed of the mobile robot is estimated directly by the agent. The third advantage of Policy Gradient Methods is their ability to learn stochastic policies, i.e., actions are chosen with a certain probability. It is especially necessary for uncertain, partially observable environments. The big disadvantage of Policy Gradient Methods is that they rather converge to a local maximum than to the global optimum. [58]

Actor-Critic Architecture

A Policy Gradient Method that makes use of the value function $v(s)$ to learn the policy parameters θ is called Actor-Critic Architecture. Figure 2.10 illustrates the basic idea of the architecture: The Actor represents the current policy and generates an action a for a given input state s. The Critic represents the value function $v(s)$ and computes the expected value for a given input state. A common practice is to update both networks with the TD-Error, discussed in chapter 2.2.5. The expected values of the current and the next state that contribute to the TD-Error are estimated with the Critic. Thus the Critic's output contributes to the Actor's update essentially.

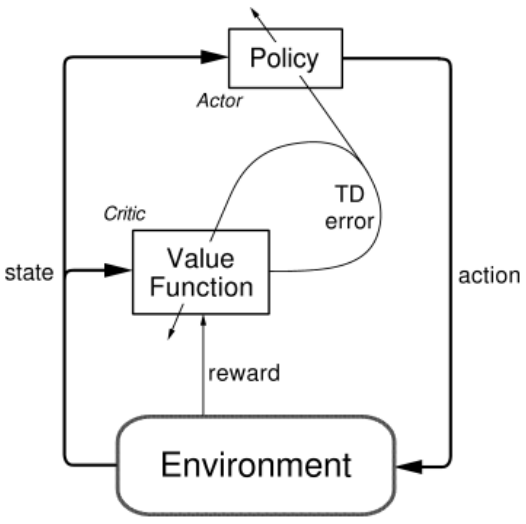


Figure 2.10.: Actor-Critic Architecture: The Actor represents the policy and maps the input state to an output action. The Critic represents the value function. Both networks can be updated with, e.g. the TD-error, in which the Critics output contributes. Thus the Actor makes use of the Critic during the learning process. [58]

REINFORCE algorithm

The REINFORCE algorithm is one of the first and simplest Policy Gradient Method introduced by [12] in 1992. In the following, a variant of the original algorithm will be presented to demonstrate the key procedure of a Policy Gradient Method.

A trajectory τ is defined as a state-action sequence with the length T: $S_0, A_0, S_1, A_1, ...S_T, A_T, S_{T+1}$. The difference between a trajectory and an episode is that the last state of a trajectory does not need to be final. The policy performance measure $J(\theta)$ in equation 2.33 is defined by the expected return of all trajectories τ. The contribution of each trajectory τ to the expected return is the product of the cumulative reward $R(\tau)$ and the probability of its occurrence $\pi_\theta(\tau)$ under policy π_θ.

$$J(\theta) = \mathbb{E}[\sum_{\tau} R(\tau)\pi_\theta(\tau)] \tag{2.33}$$

The derivative of $J(\theta)$ can be determined by applying the Policy Gradient Theorem that has been derived in [58]. This results in equation 2.34 .

$$\nabla J(\theta) = \mathbb{E}[\sum_{\tau} R(\tau)\nabla_\theta \log \pi_\theta(\tau)] \tag{2.34}$$

The REINFORCE algorithm approximates $\nabla J(\theta)$ with equation 2.35. Only one trajectory $\tau^{(i)}$ is used per iteration i to approximate the gradient of the objective function. Instead of using the raw cumulative, discounted reward $R(\tau)$, the advantage $Adv^{\pi_\theta}(S_t, A_t)$ is used. It compares the true, cumulative, discounted reward $\sum_{k=0}^{T-t} \gamma^k R_{t+k}$ to the expected return $V^\pi(S_t)$, that is estimated by a Critic network. If the true return is higher than the expected return, the advantage is positive, and the Actor will be updated in such a manner, that it is more likely to choose action A_t in state S_t. If the true return is smaller than the expected reward, the probability of taking action A_t in state S_t will be decreased.

$$\nabla_\theta J(\theta) \approx \hat{g} = \sum_{t=0}^{T} Adv^{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t|S_t) \tag{2.35}$$

$$\text{with } Adv^{\pi_\theta}(S_t, A_t) = \sum_{k=0}^{T-t} \gamma^k R_{t+k} - V^\pi(S_t) \tag{2.36}$$

Finally, the approximated gradient $\hat{g}$ is used to apply gradient ascent to update the policy parameters θ.

$$\theta \leftarrow \theta + \alpha\hat{g} \tag{2.37}$$

The REINFORCE algorithm is summarized with pseudo code in algorithm 2.

for *n_iter* **do**
1. Collect trajectory $S_0, A_0, S_1, A_1, ...S_T, A_T, S_{T+1}$ with length T.
2. Compute the approximated gradient $\hat{g}$:
   $\nabla_\theta J(\theta) \approx \hat{g} = \sum_{t=0}^{T} Adv^{\pi_\theta}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t)$
   wit $\text{Adv}^{\pi_\theta}(S_t, A_t) = \sum_{k=0}^{T-t} \gamma^k R_{t+k} - V^\pi(S_t)$
3. Update the policy's weights with gradient ascent.
   $\theta \leftarrow \theta + \alpha \hat{g}$

end

Algorithm 2: Pseudo Code for REINFORCE algorithm

Proximal Policy Optimization (PPO)

PPO [15] is a popular state-of-the-art Policy Gradient Method. It is supposed to learn relatively quickly and stable while being much simpler to tune, compared to other state-of-the-art approaches like TRPO [13], DDPG [16] or A3C [59] . This makes PPO often the first choice when it comes to solving a problem for the first time.

PPO strongly builds on Trust Region Policy Optimization (TRPO) [13]. It applies the key concepts of TRPO like *Importance Sampling*, that provides better data efficiency as well as an extended version of TRPO's *KL penalty*, that controls the update size in the optimization step. Moreover, PPO presents an alternative, simpler method called *Clipped Surrogate Objective* for controlling the optimization step size.

Importance Sampling. In the REINFORCE algorithm, at each time step a new trajectory is generated, the policy learns from it and the trajectory is thrown away. To achieve better data efficiency, importance sampling is applied in PPO. Trajectories that has been collected with older policies are reused in newer updated policies. Importance Sampling estimates the expected value of f(x) for distribution p (new policy) by sampling from q (old policy), i.e. X_i has been sampled from the data distribution q (see equation 2.38).

$$\mathbb{E}_p[f(x)] \approx \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)p(X_i)}{q(X_i)}, X_i \sim q \tag{2.38}$$

Applying Importance Sampling leads to a new objective function in equation 2.39 – called *surrogate function* $L(\theta)$.

$$L(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{old}}(S_t, A_t)} Adv^{\pi_\theta}(S_t, A_t)] \tag{2.39}$$

Adaptive KL Penalty. To ensure stable updates, the step size in the optimization step can be controlled with *Trust Region* from [13]. It prevents the optimization from taking

too big steps and from overshooting the maximum. In each step, the difference between the updated π_θ and the old policy $\pi_{\theta_{old}}$ is measured through the KL Divergence in equation 2.40.

$$D_{KL}(\pi_\theta||\pi_{\theta_{old}})[s] = \sum_{a\in A} \pi_\theta(a|s) \log \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{2.40}$$

In equation 2.41, the KL Divergence is added to the *surrogate function* $L(\theta)$ as penalty, multiplied by a factor β. If the difference between the old and new policy is big, the objective function is punished strongly.

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{old}}(S_t, A_t)} Adv^{\pi_{\theta_{old}}}(S_t, A_t)] - \beta D_{KL}(\pi_\theta||\pi_{\theta_{old}})[S_t] \tag{2.41}$$

Difficulties have been experienced with finding a good constant β, that performs well over the whole training process. That is why an adaptive β is applied in PPO according to equation 2.42. If the KL divergence d is very small, β gets smaller as well and if it is high, β gets bigger. Still the initial β and the target distance d_{targ} have to be set.

$$\beta \leftarrow \begin{cases} \beta/2 & \text{if } d < d_{targ}/1.5 \\ \beta \cdot 2 & \text{if } d > 1.5 d_{targ} \end{cases} \tag{2.42}$$

Clipped Surrogate Objective. PPO presents a second method to control stable updates by restricting the update size from one policy to another. Instead of *adaptive KL penalty*, it applies a much simpler and more effective way to realize the restriction by introducing the *clipped surrogate function*. It clips the probability ratio $r_t(\theta) = \frac{\pi_\theta(S_t,A_t)}{\pi_{\theta_{old}}(S_t,A_t)}$ between the range of $[1-\epsilon, 1+\epsilon]$ (see equation 2.43).

$$L^{clip}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta) Adv^{\pi_{\theta_{old}}}(S_t, A_t)\ , \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) Adv^{\pi_{\theta_{old}}}(S_t, A_t))] \tag{2.43}$$

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [16] is based on the Deterministic Policy Gradient Algorithms from [60] and uses an Actor-Critic Architecture. Furthermore, it adapts concepts from DQN [3] to stabilize learning, but addresses continuous action space at the same time and is, therefore, more applicable for controlling tasks.

It makes use of an *experienced replay* buffer to provide independent and uncorrelated data to the Deep Neural Networks. In addition, a variant of the *frozen target network* mechanism has been integrated, called *soft target update*. Instead of copying the weights to the target, the target networks (both Actor and Critic) are updated continuously according to equation 2.44 and slowly approach the original parameters. τ_s defines the tracking speed,

i.e., the smaller τ, the slower the target weights approach the original weights.

$$\theta^- \leftarrow \tau_s \theta + (1 - \tau_s)\theta^- \tag{2.44}$$

DDPG is built on an Actor-Critic architecture with Deep Neural Networks. The Actor $\mu(s|\theta^\mu)$, as well as the Critic $Q(s, a|\theta^Q)$, have both a duplicate target network with older weights to stabilize learning. They are denoted with a hyphen: $\mu^-(s|\theta^\mu)$ and $Q^-(s, a|\theta^Q)$. Both networks are trained by forwarding uniformally sampled experiences from the replay buffer $(S_t, A_t, R_{t+1}, S_{t+1})$. The Critic is updated by back-propagating the gradient of the squared TD-error from Q-learning (see equation 2.45 and 2.46).

$$L_i(\theta_i^Q) = \hat{\mathbb{E}}_t[(R_{t+1} + \gamma Q^-(S_{t+1}, \mu^-(S_{t+1}|\theta_i^{\mu-})|\theta_i^{Q-}) - Q(S_t, A_t|\theta_i^Q))^2] \tag{2.45}$$

$$\theta_{i+1}^Q = \theta_i^Q + \alpha_c \nabla_{\theta^Q} L_i(\theta_i^Q) \tag{2.46}$$

The Actor is updated with the Deterministic Policy Gradient [60] by applying gradient ascent to the output of the Critic with respect to the state s and the estimated action by the actor $\mu(S_t|\Theta_i^\mu)$ (see equation 2.47 and 2.48).

$$L_i(\theta_i^\mu) = \hat{\mathbb{E}}_t[Q(S_t, \mu(S_t|\theta_i^\mu)|\theta_t^Q)] \tag{2.47}$$

$$\theta_{i+1}^\mu = \theta_i^\mu + \alpha_a \nabla_{\theta^Q} L_i(\theta_i^\mu) \tag{2.48}$$

In order to apply exploration to continuous action space, noise $\mathcal{N}$ is added to the actor's output $\mu(s|\Theta^\mu)$. The noise is generated by the Ornstein-Uhlenbeck process [61].

$$\mu_{\mathcal{N}}(s) = \mu(s|\Theta^\mu) + \mathcal{N} \tag{2.49}$$

3. Simulation Environment

This chapter introduces the used simulation environment, that fuses the two pre-existing simulation environments Flatland [62] and Pedsim_ros [63]. The modeling of all static and dynamic objects as well as the robot are presented in detail.

The training process has been restricted to a 2D world. Eventually, just 2D laser scan sensors are considered. It is assumed that all relevant objects can be successfully detected in the working plane of the laser scan sensors.

The main advantage of restricting the problem to two dimensions is that the simulated world can approximate the real world better. Recent 3D simulators like Vrep [64] or Gazebo [65] do not provide sufficiently accurate 3D worlds so that the resulting images of camera sensors are too simplistic and not even close to real world images. Hence, a transfer from simulation to real world is expected to be easier considering a 2D world problem. Using 2D laser scanners as input source has two more relevant advantages: The number of raw input points is smaller and the computational consumption of the simulator is lower than in 3D simulators. Both advantages contribute to faster training.

As base 2D simulator, the Flatland simulator has been chosen. To create a dynamic environment of walking humans, the PedSim simulator has been integrated into Flatland. All relevant elements of the final simulation will be explained in the following subsections.

3.1. Pedsim Crowd Simulator

PedSim [1] is a 2D-simulator that is specialized on pedestrian crowd simulation. The underlying concept of the pedestrian movement patterns is the social force model from [66]. Different forces can influence the behavior of the pedestrians. The sum of all influencing forces defines in which direction and with which acceleration the pedestrians move (see equation 3.1).

$$F_{sum} = f_{des} + \sum_{j} f_{ij} + \sum_{W} f_{iW} \tag{3.1}$$

The social force model considers different summands that are further explained in the following listing:

- **Desired Force** f_{des}: The desired force points in the direction of the current goal of

[1] accessed 2019-04-08: http://pedsim.silmaril.org/

the agent. Furthermore, the force is needed to reach the maximal velocity v_i.

- **Pedestrian Force** f_{ij}: The force avoids collisions with other pedestrians. The force is estimated from the minimal allowed distance to each pedestrian.
- **Wall Force** f_{iW}: The force avoids collisions with obstacles like walls and is estimated from the distance to each wall.

With the original social force model, an undesired behavior of pedestrians – frequently walking into the robot footprint – is produced. To prevent this behavior, the exponential repellent force coming from the walls is also implemented for the robot f_r, named **Robot Force** in the following. It is a circular repulsion arising from the robot center regardless of the robot shape. The weight of the force needs to be adjusted in such a manner, that it is guaranteed, that the pedestrians don't walk through the footprint of the robot.

In the following paragraphs, example pedestrian-obstacle interactions are provided.

In PedSim, obstacles are defined with straight lines, i.e., round objects or splines can only be approximated with a high number of small lines. Each line has a repellent influence on the agent. The strength of the Wall Force increases exponentially with the decreasing distance between the wall and the agent. Figure 3.1a shows a 2D-agent approximated with two round red legs, that tries to reach the waypoint behind the obstacle (red rectangle). It does not approach the wall closer than ~ 0.3 m. The social force model requires a careful definition of waypoints for the pedestrians. Especially in more complex environments with many obstacles, it is very likely that pedestrians end up in a local minimum. In figure 3.1a, it never reaches the waypoint behind the obstacles, because the repellent force coming from the obstacles and the attracting force coming from the waypoint (star) point in the exact opposite directions and the pedestrian ends up in a position, where the forces are evened out. To overcome this problem, a finer waypoint definition, leading around the obstacle, would be necessary.

Figure 3.1b, 3.1c and 3.1d show different situations, where a pedestrian (two round red circles) and the robot (gray rectangle) interact with the additional Robot Force. In figure 3.1b the pedestrian walks between two waypoints forth and back while avoiding the robot. The red dotted line shows the taken path. Thus it needs to be mentioned, that the pedestrian does not behave as forward-looking as humans would do. In figure 3.1c, the pedestrian again ends up in a local minimum, because the robot is positioned in such a manner that the center of the robot, the pedestrian and the next waypoint are perfectly aligned, resulting in having the Desired Force and the Robot Force point exactly in opposite directions. In Figure 3.1d, the robot is positioned on the waypoint. The pedestrian walks around the robot and again ends up at a position where forces are evened out and waits until the robot leaves that position.

For this thesis different pedestrian behaviors have been implemented, that are presented in the following:

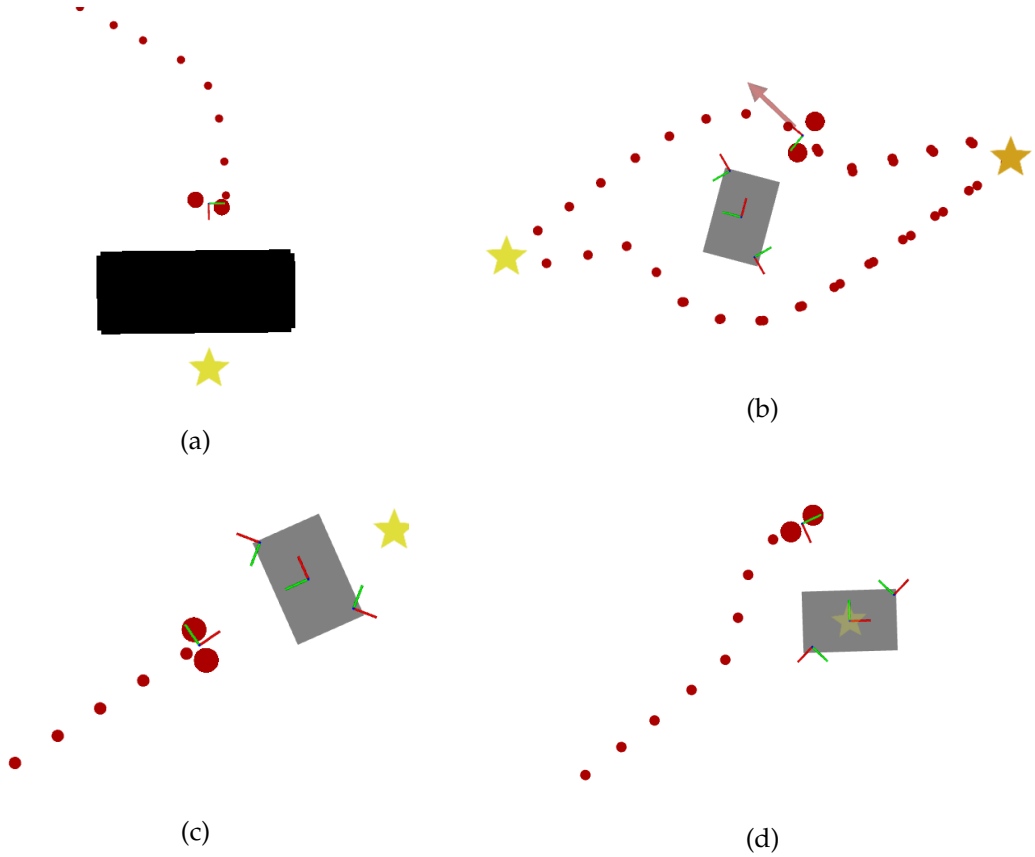


Figure 3.1.: Example situations that illustrate the behavior between a pedestrian (two red circles) and the robot (grey rectangle) in the PedSim-simulator. **(a)** The pedestrian tries to reach the goal (star) behind the obstacle (black rectangle). The Wall Force is strong enough to prevent that the agent walks against or through the wall. **(b)** The pedestrian walks from waypoint 1 (yellow star) to waypoint 2 (orange star) and back, while the robot stands in the direct way. The red dotted line shows which path has been taken by the pedestrian instead. **(c)** The pedestrian tries to reach the next waypoint. The robots center, the agent and the waypoint are aligned. The agent is not able to walk around the robot because the Robot Force and the Desired Force point in the exact opposite direction. **(d)** The robot is positioned on a waypoint. The pedestrian tries to reach the waypoint by walking around the robot and ends up waiting at one position.

- **Polite Pedestrian**. The Robot Force is applied, resulting in a reliable avoidance of the robot.
- **Impolite Pedestrian**. No Robot Force is applied. The pedestrian walks into the robot if it does not make room.
- **Semi-polite Pedestrian**. The Robot Force is only applied if the velocity of the robot falls below a certain threshold $v_{reaction,max}$ for at least $t_{reaction}$ seconds. This behavior simulates a pedestrian that in general expects the robot to make room, but in case the robot moves relatively slow, the pedestrian avoids the robot after a certain reaction time.

The EU-funded SPENCER project integrated the original Pedsim library in ROS, called Pedsim_ros [63]. The position and velocity of the robot is frequently updated according to the published transform from the odometry frame to the robot_base frame. The state of all pedestrians like position, velocity and operating forces are made available in ROS by publishing them on a ROS-topic. Additionally, they extended the basic social force model with group walking behavior by simulating several agents walking together. Each agent has the desire to not be too far from its social group. If an agent gets lost, it aims to get back to its group.

3.2. Flatland Simulator

Flatland is a 2,5D simulator, integrated into the ROS framework [44]. It is mainly built on the Box2D physics engine [2], that was originally developed for game development. The Box2D library is wrapped in Flatland, so that the basic ROS concepts like parameter loading through yaml-files, dynamic loading of customized plugins with pluginlib and publishing transforms on the *tf* topic can be applied. The simulator provides visualization by publishing markers of all defined objects and displaying them in Rviz.

Flatland has been chosen over the widely used, in ROS integrated, player/stage 2D-simulator [67] and the stdr_simulator [68], because it provides clean code with structured documentation. It already offers relevant plugins like the laser-plugin and the diff_drive-plugin. It is also possible to implement new, customized plugins and easily integrate them in the Flatland simulator via *ROS pluginlib*. Another big advantage is the configurable simulation speed: real time factor $=$ step size $\cdot$ update_rate. This allows one to run the simulation faster than in real time and speeds up the training time.

3.2.1. Static Environment

The static environment can be created by loading a defined map via a yaml-file, containing all relevant information of the map like the floor plan, resolution and origin. This

[2] accessed 2019-02-18: https://box2d.org/about/

(a) (b) (c) (d)

Figure 3.2.: Static obstacles, that are spawned and removed dynamically in the world. **(a)** represents a cylinder, **(b)** represents the MiR500 robot, **(c)** represents a wagon or a small table with four legs and **(d)** represents an EPAL-palett.

makes it possible to load real 2D maps of high complexity. Besides, additional static objects, composed of primitives, can be loaded and removed dynamically. Only four different kinds of common objects are considered, displayed in figure 3.2. Object 3.2b represents a non-moving MiR500 robot [3], while 3.2d represent an EPAL-palett. They have different dimensions. Object 3.2c is supposed to be a representative for a 4-legged wagon or a table.

3.2.2. Pedestrian

To get the pedestrians from PedSim into Flatland, a pedestrian-plugin has been implemented, that synchronizes the pedestrian state of the PedSim-simulator with the Flatland simulator. Each pedestrian gets attached a pedestrian-plugin. The plugin models each pedestrian as a body, consisting of two circles of a variable radius and offset, modeling the legs of the person (see figure 3.1 for examples). Moreover, it takes into account the speed and position of the appropriate pedestrian and applies the same velocities and pose transformations to the body in Flatland.

The pedestrians in Flatland have been further extended with a simple walking pattern. One leg at a time swings according to a triangular velocity function. The leg accelerates to a maximum velocity, that is four times faster than the global body velocity $v_{i,leg,max} = 4v_i$, followed by a constant deceleration to 0 m/s. During that period of time, the other leg remains in the same position. Figure 3.3 illustrates the explained walking model pattern.

3.2.3. Mobile Robot

The model of the robot consists of a single polygon body, that represents the footprint of the real robot. It does not contain any physics like joints, motors, and friction properties, because they are not assumed to be relevant for the defined task. Particularly, modeling physics consumes a relevant portion of computation power.

The front and back laser scan sensors of the mobile robot are simulated with the pre-existing laser-plugin in Flatland. They are attached to the polygon body and consider the whole environment (static environment, pedestrians, other robots) in their data output.

[3] accessed 2019-04-10: https://www.mobile-industrial-robots.com/de/products/mir500/

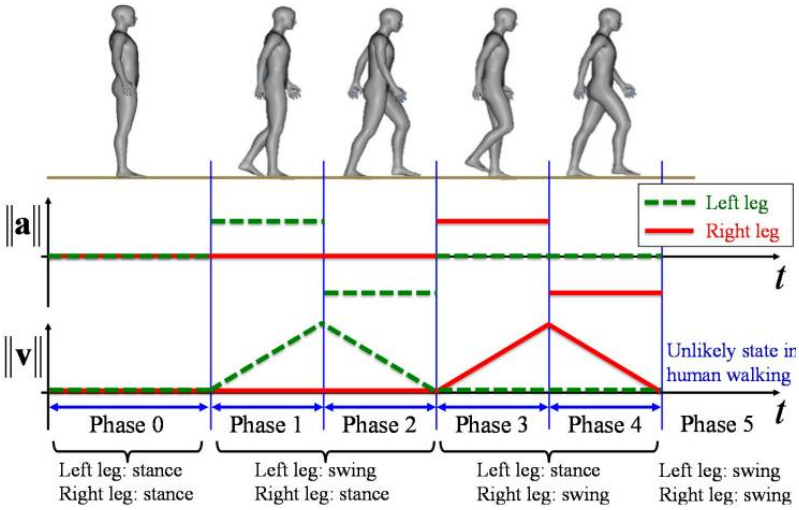


Figure 3.3.: Different phases of a simple leg movement model. One leg swings at a time and follows a triangular velocity function, while the other leg remains on the same position. [69]

The noise of the sensor data is modeled with Gaussian noise. Furthermore, two artificial laser scans "ped_laser" and "static_laser" are added to the center of the robot. They either represent static objects or pedestrians in their data. This makes it possible to distinguish between both obstacle types and can be useful in the reward function.

Moreover, a modified diff_drive-plugin has been attached to the robot. It models the response of the MiR100 robot to external velocity commands. Flatland's original diff_drive-plugin has been extended with the relevant dynamic behavior of the real mobile robot. A discrete transfer function is applied to the input velocity commands and shows effects on velocity changes, i.e., acceleration or deceleration. To obtain the transfer function, Matias Vinsten (working at Mobile Industrial Robots ApS) performed experiments on the real robot: He applied a step function to the robot, measured the output via the encoders and retrieved the transfer function from the delay, settle time and system gain. The extended diff_drive-plugin models a time delay and a slight overshoot during acceleration.

4. Methods and Setup

The chapter presents the used methods and the concrete training setup. Section 4.2 covers in detail how the tasks for each episode are generated, while a static and a dynamic task setup is considered. Moreover, four different global maps of different complexities are introduced. Section 4.3 presents the RL-agent specific setup that includes different observation spaces, action spaces, reward functions and Neural Network Architectures. Furthermore, the used library stable-baselines [2] is shortly introduced.

4.1. Navigation Stack Setup

The RL-agent will be plugged in the traditional navigation software, introduced in chapter 1.3.2. In other words, the SBPL lattice planner still provides the global plan, serving as a guideline for the RL-agent, that replaces the traditional local planner. The RL-agent is supposed to provide velocity commands with a frequency of 10 Hz. Note that the AMCL localization has been disabled and perfect localization is provided to isolate the performance of the global planner from other possible error sources.

4.2. Task Setup

The task of the agent is to navigate successfully from a start to a goal position along a given global plan. Furthermore, the agent is supposed to react on local objects, that have not been considered by the global plan. As input the agent gets a representation of its environment and interacts with the environment by controlling the robots translational and rotational velocity (v, ω). It will be differentiated between three kinds of objects: global static objects (black), local static objects (green) and pedestrians (red). Global static objects are fixed objects that are listed in the global map and therefore, are considered by the global plan and should not challenge the agent if it follows the global plan correctly. Local static objects are non-moving objects that are not present in the global map. They are detected by the laser scan sensors of the robot. Pedestrians are likewise detected by the laser scan sensors and are the most challenging objects because they move and require a forward-looking behavior of the agent. It is desired, that the agent takes into consideration the walking speed and direction.

In this thesis, two different task setups are considered: a static setup and a dynamic setup. Both setups are based on a global map as world (see figure 4.2) providing global

static objects. For each episode a new random start and goal position is generated on free space in the global map, resulting in a global plan computed by the global sbpl-planner. Obstacles are spawned randomly on or near the path and the RL-agent is supposed to avoid them during the navigation along the path. An episode is successfully fulfilled if the robot reaches the final goal within 0.4 m. It is not expected to reach the goal exactly, because it would increase the complexity of the learning problem and the traditional navigation software already provides a well-functioning mode for approaching the goal precisely. Furthermore, an episode will be stopped, if the RL-agent drives into an obstacle, deviates from the path further than 4 m or exceeds the maximum time of 65 seconds during task completion.

Static Setup

For each episode static objects are spawned along the current path. The spawning process is randomized as follows.

- The number of static objects is determined randomly, but restricted by a maximum quantity, that is dependent on the path length.
- The object type is selected randomly from all static objects in figure 3.2.
- The position of each static object $p_{so,i}$ is generated by selecting a random position $p_{gp,i}$ on the global path in the first place and secondly choosing a random position $p_{so,i}$ in a radius of 1 m around $p_{gp,i}$.

Figure 4.1a shows example episodes of the static task setup with the global plan (blue), global static objects (black) and local static objects (green). There are episodes with single isolated static objects, but also episodes with object compositions, that lead to new object shapes. Episodes with local static objects positioned close to global static objects provide narrow sections. The agent should learn which of those are passable and if they are not, to avoid the obstacle from the other side. Unfortunately, unsolvable episodes can be generated like in the second top row. It happens rather rarely because the agent is trained in open spaces.

Dynamic Setup

For each episode, pedestrians, that either cross the path, walk along the path or stand around close to the path, are spawned. The spawning process is randomized, described in more detail in the following:

- The number of pedestrians is determined randomly, but restricted by a maximum quantity, that is dependent on the path length.
- $\sim$ 10 % of the total number of pedestrians are standing around. The position $p_{p,i}$ is generated by selecting a random position $p_{gp,i}$ on the global path and finally choosing a random position $p_{p,i}$ in a radius of 2 m around $p_{gp,i}$

- $\sim$ 52 % of the total number of pedestrians are walking along the path. The pedestrian walks back and forth between two waypoints close to the path. The waypoints have a minimum distance of 5 m and their positions are generated in the same manner as explained in the previous bullet point.

- $\sim$ 38 % of the total number of pedestrians are crossing the path. The pedestrian walks back and forth between two waypoints, that have a distance of $\sim$ 8 m and are on different sides of the global path. If the static global objects do not allow a distance of 8 m, the distance is decreased accordingly. Again, a random position $p_{gp,i}$ on the global path is selected. The waypoints of the pedestrian $p_{p,i}$ are generated by randomly selecting two opposite position on the circle with a center at $p_{gp,i}$ and a radius 4 m, that has an angle $45° < \beta_1 < 135°$ and $-145° < \beta_2 < -45°$ to $p_{gp,i}$.

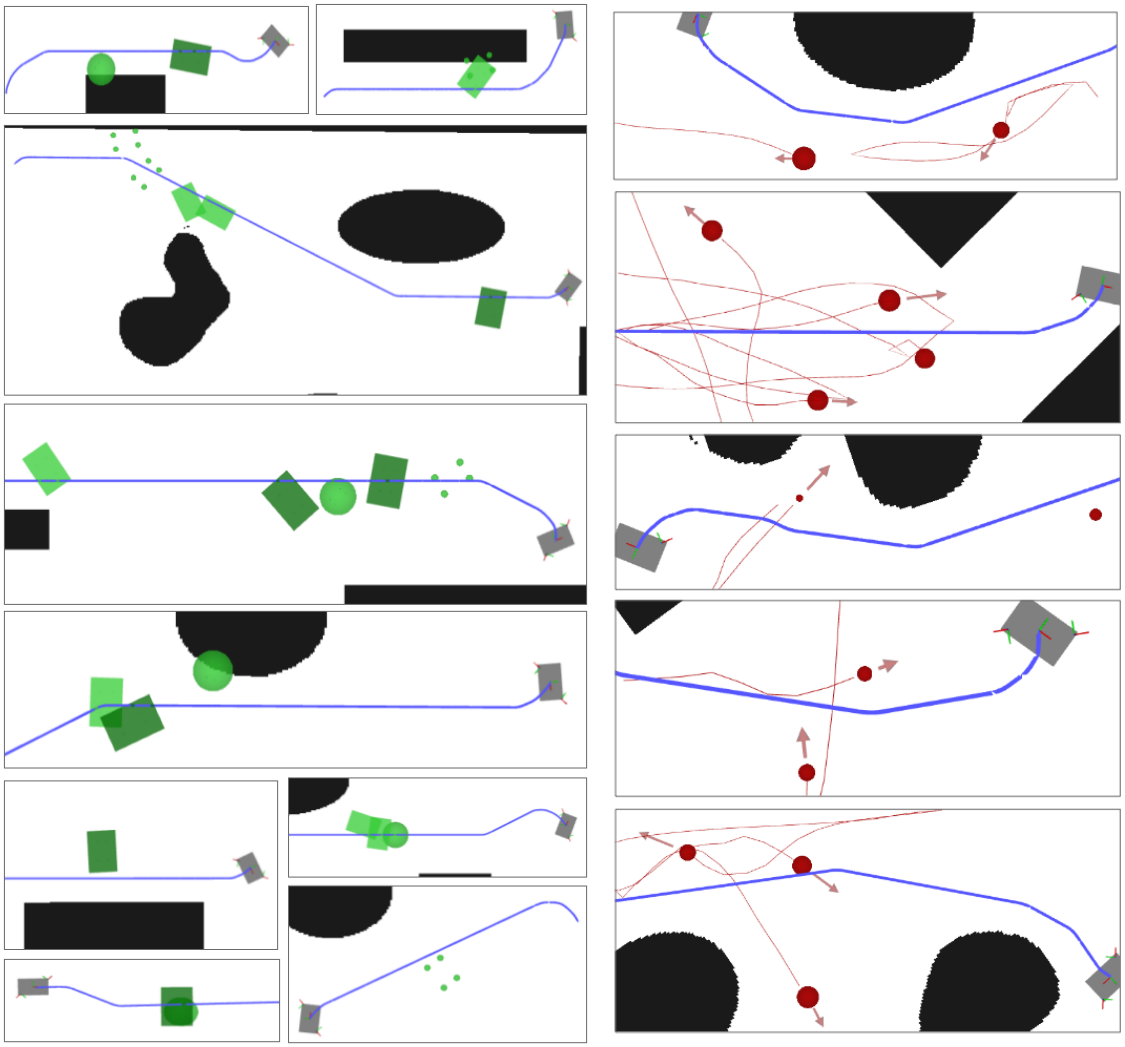
The number of spawned pedestrians is in general higher than the number of spawned static objects, because it is not guaranteed, that the RL-agent encounters all pedestrians. To break down the problem complexity, each pedestrian is modeled as a single circle moving around with a random speed generated by a normal distribution. The mean of the normal distribution is slightly lower than the maximum translational velocity v_{max}. This allows the robot to overtake some people.

Figure 4.1b shows example episodes of the dynamic task setup with the global plan (blue), global static objects (black) and pedestrians (red). The red circle represent the pedestrians, the arrows show the walking direction and speed and the fine red lines represent their walking history.

Global World Setup

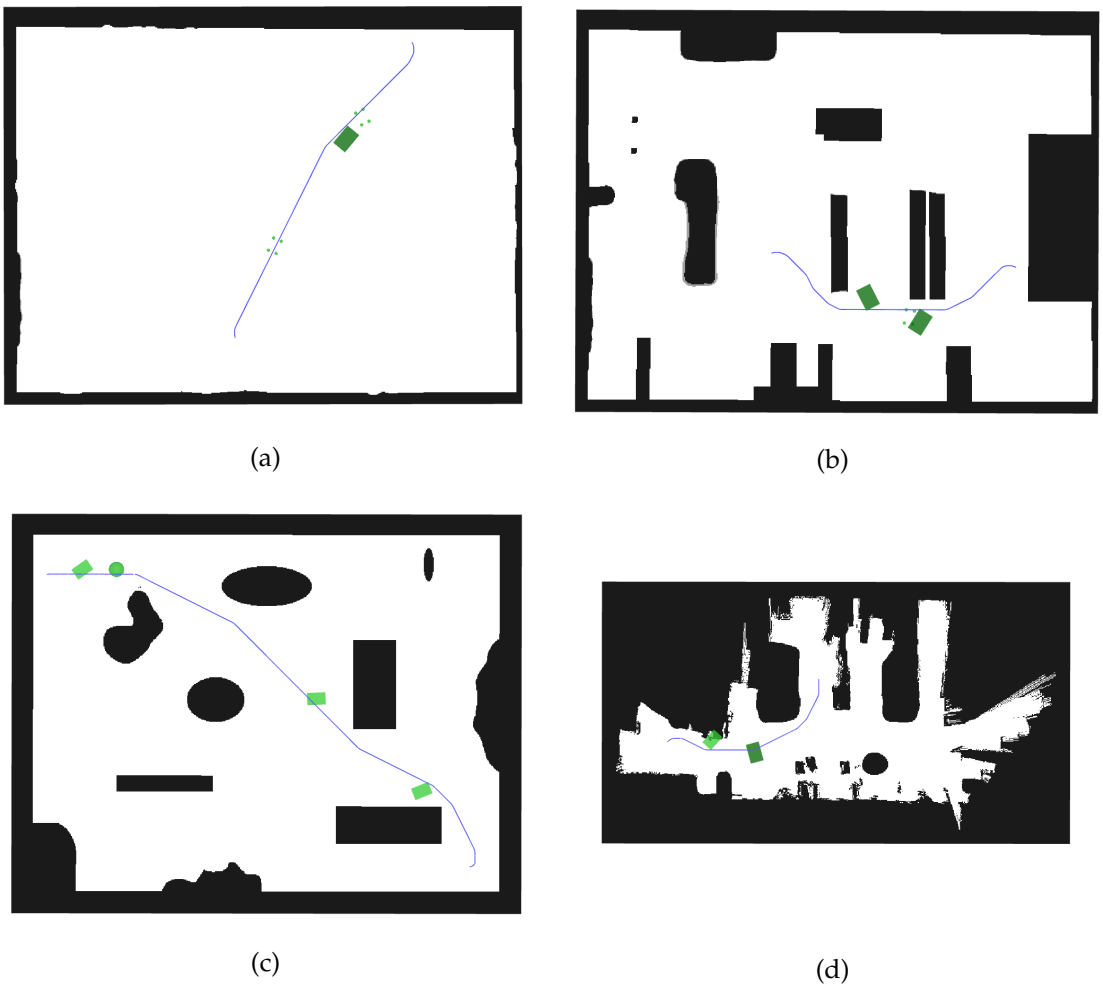
During this thesis, four different maps of different complexity shown in figure 4.2 are considered. The map 4.2a is empty and provides enough room to the left as well the right during obstacle avoidance. Map 4.2b and 4.2c have an average complexity. The distribution of global static objects provides mostly enough room to at least one side. Nevertheless, narrow situations, as well as unsolvable situations, can occur. Map 4.2d is the most complex map because the map is in general smaller and provides more narrow passages as well as one-way corridors.

Map 4.2b is used during training and provides an adequate complexity, while the number of unsolvable generated tasks is kept low. There is no need for map variation during training. The RL-agent does not overfit to the provided map, because the additional spawned objects provide enough variation. Map 4.2a, 4.2c and 4.2d are used during testing. Each agent will be evaluated in environments of increasing complexity.



(a) Static Setup (b) Dynamic Setup

Figure 4.1.: The agent is trained on two different task setups: static setup and dynamic setup. **(a)** The static setup includes the global plan (blue), global static objects (black) and local static objects (green), that are randomly spawned on the global path. **(b)** The dynamic setup includes the global plan (blue), global static objects (black) and pedestrians (red), that are randomly spawned on the global path. The pedestrians are either walking along the path, crossing the path or standing around. A single circle represents the pedestrians while the arrow shows the walking direction and speed and the fine line shows the walking history.



(a) (b)

(c) (d)

Figure 4.2.: Maps of different complexities. **(a)** An empty map that provides enough room to both sides for obstacle avoidance. **(b)** and **(c)** A map with an average complexity. The randomly generated tasks mostly provide enough room for avoidance to one side, but also can generate narrow or unsolvable tasks. **(d)** A map of a high complexity with many narrow situations as well as a higher probability unsolvable task in case of random task generation.

4.3. RL-Agent Setup

As Reinforcement Learning approach, Proximal Policy Optimization (PPO), described in chapter 2.3.2, is used. The implementation from the open source project stable-baselines [2] are applied. Stable-baselines provides a set of improved RL-algorithms like PPO, DQN, TRPO, DDPG and more. Details about the general PPO training parameter settings can be found in the appendix in table A.3 and B.4. The deep learning part is realized in the Tensorflow framework [70] and allows to define custom policy-networks.

Some algorithms in the stable-baselines library even support multiprocess training, i.e., training an RL-agent on n environments using n processes. All n agents in the different environments act according to the same policy π and collect episodes to update and improve the network policy. Moreover, it is possible to plug in a custom simulation environment that needs to implement the OpenAI-Gym interface *gym.Env*. To integrate the simulation environment with the existing move_base implementation as well as the stable-baselines library, a wrapper has been created, that on one side communicates with a local planner of move_base fulfilling the interface of *nav_core::BaseLocalPlanner*, and on the other side provides the *gym.Env*-interface for the stable-baselines library.

4.3.1. Observation Space

The RL-agent needs to get all relevant information about the current state of the environment to be able to fulfill the task successfully. The following listing provides the raw data that has been identified as relevant.

- Waypoints. The global plan is downsampled to a number of waypoints with a distance of $d_{lookahead}$ to each other. A vector of n_{wp} sequential waypoints is needed, while the first waypoint is the closest to the robot. Each waypoint position is given in Euclidean space with the (x, y)-coordinate in the robot frame.
- Laser scan data. The laser scan data provides information about obstacles and free space. The back- and front-laser scan data is merged together, resulting in a merged data vector of length l_{laser_vec}, that covers 360 $^\circ$ and provides a distance value for each angle increment. The vector is discretized to a resolution *res*.
- Robot velocity. The current linear and angular velocity (v, ω) of the center of the robot are considered to be relevant only in the dynamic setup.

The previously listed relevant information can be fed to a Neural Network in various different representations. In this thesis I investigate four different data representations, that will be presented in the following paragraphs.

Raw Data Representation. The most obvious is to feed the raw data directly. Specifically the laser scan vector of normalized distance values and the normalized (x, y)-positions

of the n_{wp} waypoints is fed directly. Note that the laser scan vector provides an indirect form of polar coordinates, while the waypoints are given in cartesian coordinates.

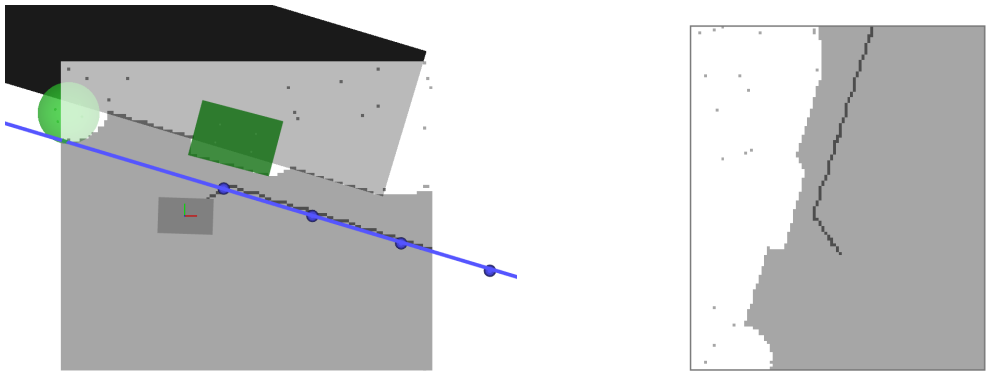
Polar Representation. It is assumed, that providing data in the same format should result in better training performance. For that reason, the waypoints have been transformed into a vector of the same length as the laser scan vector l_{laser_vec}. For each waypoint, the distance and the angle are computed with respect to the robot position. Each waypoint distance is entered at the appropriate angle position in the vector, while all remaining values are zero. E.g. if four waypoints are provided, the vector has maximal four non-zero entries, while each entry represents the distance to a waypoint. If several waypoints have the same angle, the shortest distance will be considered. Finally, the normalized waypoint- as well as the normalized laser scan vector are fed to a Neural Network.

X-Image Representation. Convolutional Neural Networks have been particularly successful with images in the past years. For that reason, the raw input data is transformed into an image. The image has a size of $[(width\_front + width\_back) \times height]$, a resolution of *res* m/px and is generated from the laser scan vector and the waypoint positions. Figure 4.3a shows a sample scene with one global static object (black), two local static objects (green), the global path and its next four waypoints (blue) as well as the robot (small grey rectangle). Obviously, the scene is overlayed with the input image, that is again shown solely in figure 4.3b. The input image represents the environment of the robot, while the robot location is always at pixel $[width\_back, height/2]$ pointing to the front. A black line is drawn from the robot location along the waypoint vector. Free space is marked with grey pixels and occupied space as well as unknown space is marked with white pixels. The white pixels are generated from the laser scan vector by drawing a line from each scan point pointing in the appropriate direction using the Bresenham's algorithm [71]. It is also possible to feed a stack of X images to the Convolutional Neural Network, where each image represents a different time stamp.

X-Image Speed Representation. This representation builds on the X-Image Representation, but further considers the robot velocity (v, ω). The data has again two different formats: The waypoint- and laser scan-data are provided in the form of an image, while the robot velocity is provided in its raw format. This representation is only used in the dynamic setup because the robot's own velocity is crucial for estimating the movement of dynamic objects.

4.3.2. Action Space

Continuous as well as discrete action space have been investigated. For the continuous action space, limits for the translational velocity $[0, v_{max}]$ and angular velocity $[-\omega_{max}, \omega_{max}]$ are defined. The discrete action space allows six discrete actions, that are combinations



(a) (b)

Figure 4.3.: Generation of the input image from the laser scan and waypoint vector. **(a)** An example scene is given with a global static object (black), two local static objects (green), the global plan and its next four waypoints (blue) as well as the robot (small grey rectangle). The scene is overlayed with the generated input image. **(b)** The input image of the scene (a) is shown solely and with the true orientation. It includes a black line from the robot location along the waypoint vector as well as a white line from each scan pointing away from the robot in the direction of the appropriate scan angle.

of translational and angular velocity:$[0.0, -\omega_{max}]$, $[v_{max}, 0.0]$, $[0.0, \omega_{max}]$, $[v_{max}, \omega_{max}/2]$, $[v_{max}, -\omega_{max}/2]$ and $[0.0, 0.0]$.

4.3.3. Reward Functions

Reward function shaping is challenging and critical for successful learning. Many different reward functions have been investigated during this thesis and the relevant ones will be presented in the following. Rewards are computed each time step t after taking action a_t.

Reward function 1

This reward function is especially successful for the static training setup and is shown in equation 4.1. It considers three summands regarding the closest waypoint (wp), the obstacles (o) and the final goal (g).

$$r_t = r_t(wp) + r_t(o) + r_t(g) \tag{4.1}$$

The reward $r_t(g)$ contributes with a positive constant R_g, if the robot reaches the final goal within a radius of D_g. $d(p_r, p_g)$ is the distance function, that computes the distance

between the robot position $p_{r,t}$ at time step t and the goal point p_g.

$$r_t(g) = \begin{cases} R_g & \text{if } d(p_{r,t}, p_g) < D_g \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The reward $r_t(o)$ contributes with a negative constant R_o, if the robot collides with any kind of obstacle $\in O$.

$$r_t(o) = \begin{cases} -R_o & \text{if collision with an obstacle } \in O \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

The RL-agent at position $p_{r,t}$ is awarded for getting closer to the next waypoint $p_{wp,t}$ in equation 4.7, but it is punished for driving away from the next waypoint $p_{wp,t}$ in equation 4.8. Both phenomena can be weighted differently. The diff()-function in equation 4.6 determines the difference between the distance from the RL-agent to the next waypoint of the previous time step $d(p_{r,t-1}, p_{wp,t-1})$ and the distance from the RL-agent to the next waypoint of the most recent time step $d(p_{r,t}, p_{wp,t})$. Additionally it gets a slightly higher positive reward R_{wp} for reaching a waypoint within D_{wp} in equation 4.9. R_{wp} is only given once for each waypoint, because after reaching it the next waypoint is automatically triggered. Furthermore $r_t(wp)$ is disabled, if the robot is close to an obstacle $\in O$ within D_o. It motivates the RL-agent to privilege obstacle avoidance over path following. Equations 4.4 to 4.9 formalize the reward $r_t(wp)$ completely.

$$r_t(wp) = \begin{cases} 0 & \text{if } \min\limits_{o_i \in O}(d(p_{o_i,t}, p_{r,t})) < D_o \\ r'_t(wp) & \text{otherwise} \end{cases} \tag{4.4}$$

$$r'_t(wp) = r_{1t}(wp) + r_{2t}(wp) + r_{3t}(wp) \tag{4.5}$$

$$\text{diff}(p_{r,t}, p_{wp,t}) = d(p_{r,t-1}, p_{wp,t-1}) - d(p_{r,t}, p_{wp,t}) \tag{4.6}$$

$$r_{1t}(wp) = \begin{cases} w_1 \cdot \text{diff}(p_{r,t}, p_{wp,t}) & \text{if diff}(p_{r,t}, p_{wp,t}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

$$r_{2t}(wp) = \begin{cases} w_2 \cdot \text{diff}(p_{r,t}, p_{wp,t}) & \text{if diff}(p_{r,t}, p_{wp,t}) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

$$r_{3t}(wp) = \begin{cases} R_{wp} & \text{if } d(p_{r,t}, p_{wp,t}) < D_{wp} \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Reward function 2

The reward function in equation 4.10 is applied during dynamic training. It builds on function 4.1, but distinguishes between the static and dynamic object types. It considers rewards regarding the closest waypoint (wp), the obstacles (o), the final goal (g) and the velocity of the robot (vel).

$$r_{t,2} = r_t(wp) + r_{t,2}(o) + r_t(g) + r_t(vel) \tag{4.10}$$

The reward for approaching the closest waypoint (see equation 4.4 to 4.9) and for reaching the goal (see equation 4.2) is estimated in the same way like in reward function 4.1. The summand $r_{t,2}(o)$ distinguishes between static objects (so) and pedestrians (ped). This enables the agent to approach static objects closer while moving objects should be avoided with a higher safety distance. If the RL-agent collides with a static object $\in SO$, it results in a negative constant R_{so} (see equation 4.12). To train the RL-agent to avoid pedestrians with more space, the RL-agent is already rewarded negatively, if it ends up in the circular area of radius D_{ped} around any pedestrian. Since the pedestrians are moving as well, it is not always possible for the RL-agent to keep the demanded distance to all pedestrians in, e.g. in crowded or narrow situations. For that reason, the RL-agent does not get punished negatively for being too close to a pedestrian if it has been driving slowly ($< |v_{reaction,max}|$) in beforehand for a duration of $t_{reaction}$. This motivates the RL-agent to foresee critical situations and wait for the pedestrians to pass them. Equation 4.13 summarizes the rewarding of interactions with pedestrians $\in PED$.

$$r_{t,2}(o) = \min(r_t(so), r_t(ped)) \tag{4.11}$$

$$r_t(so) = \begin{cases} -R_{so} & \text{if collision with a static obstacle } \in SO \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

$$r_t(ped) = \begin{cases} 0 & \text{if } \min\limits_{ped_i \in PED}(d(p_{ped_i,t}, p_{r,t})) > D_{ped} \\ & \textbf{or } v \leq v_{reaction,max} \text{ for a duration of } t_{reaction} \\ -R_{ped} & \text{otherwise} \end{cases} \tag{4.13}$$

Equation 4.14 shows the estimation of $r_t(vel)$ in more detail. It punishes the RL-agent for not moving forward, while standing completely still is punished differently with $-R_{vel1}$ than turning in place with $-R_{vel2}$.

$$r_t(vel) = \begin{cases} -R_{vel1} & \text{if } v_t = 0 \text{ and } \omega_t = 0 \\ -R_{vel2} & \text{if } v_t = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

4.3.4. Neural Network Architectures

The design of Neural Networks varies in the number of layers, the different layer sizes, different activation functions, different layer types, etc.. To reduce the search space, the focus is not set on network shaping in this thesis; instead, network architectures, that have already shown success in similar learning tasks, are applied. Three different network architectures are used and are applied for both, the Actor as well as the Critic Network. The networks are initialized with orthogonal matrix initialization [72].

1D Convolutional Neural Network

Table 4.1 shows a 1D-Convolutional-Network, that is used in combination with the Raw Data as well as the Polar Representation and is inspired by the work of Long and Fan [27]. They trained a multi-robot scenario, where each robot had to drive to a certain goal and avoid static and dynamic obstacles on their way. They provide a similar task scenario, except that the robot is not supposed to follow a given global path to the final goal. The first hidden layer is a Convolution Layer with 32 filters, a filter size of $[5 \times 1]$ and a stride of 2. The filters only consider data in one dimension along the 1D-input vector and are therefore shifted only along the vector. Those Convolutional Layers are called 1D-Convolution. The second layer is a 1D-Convolution with 32 filters, a filter size of 3 and a stride of 2. The third layer is a fully-connected layer with 256 neurons and the fourth layer is a fully connected layer with 512 neurons. To all hidden layers, a ReLU activation function is applied. The fourth hidden layer finally is mapped to the output size, that varies according to the action space size: The output size of the continuous action space is 2 and the output size of the discrete action space is 6. The input layer depends on the used input data representation. In the case of the Polar Representation, the vector of the size $[2, l_{laser_vec}]$ is simply fed to the first layer. In case of the Raw Data Representation, the laser scan vector is processed by layer 1 of the network, while the n_{wp} closest waypoints are concatenated with the output of layer three and then forwarded to layer four.

Layer	Type	Activation	Size	Filter Size	Filter Stride
1	Convolution	ReLu	32 Filter	$[5 \times 1]$	$[2 \times 0]$
2	Convolution	ReLu	32 Filter	$[3 \times 1]$	$[2 \times 0]$
3	Fully-Connected	ReLu	256 Neurons	-	-
4	Fully-Connected	ReLu	128 Neurons	-	-
5	Fully-Connected	Linear	Output Size	-	-

Table 4.1.: Raw data network.

Layer	Type	Activation	Size	Filter Size	Filter Stride
1	Convolution	ReLu	32 Filter	$[8 \times 8]$	$[4 \times 4]$
2	Convolution	ReLu	64 Filter	$[4 \times 4]$	$[2 \times 2]$
3	Convolution	ReLu	64 Filter	$[3 \times 3]$	$[1 \times 1]$
4	Fully-Connected	ReLu	512 Neurons	-	-
5	Fully-Connected	Linear	Output Size	-	-

Table 4.2.: 4-layered image network for the static setup.

2D Convolutional 4-layered Neural Network

Table 4.2 provides the network architecture for images as input and is the default 2D-Convolutional Network of stable-baselines [2] and inspired by DQN [3]. It solved diverse Atari games successfully and the complexity of our task scenario is comparable to some of them. The first hidden layer is a 2D-Convolution with 32 filters with a filter size of $[2 \times 2]$ and a stride of 4. The second and third layer are as well 2D-Convolution with 64 filters, while the second layer has a filter size of $[4 \times 4]$ and a stride of 2 and the third layer has a filter size of $[3 \times 3]$ and a stride of 1. The fourth and last hidden layer is a fully-connected layer with 512 neurons. All hidden layers apply the ReLu activation function. The output size depends as well on the used action space size as mentioned in the previous subchapter. The input depends again on the data representation. In the case of the X-Image Representation a stack of $n_{stack} = X$ state images are processed by the first layer. In the case of the X-Image Speed Representation, the velocity of the robot is additionally provided by concatenating it with the flattened output of layer three.

2D Convolutional 6-layered Neural Network

The 2D Convolutional Neural Network in table 4.3 is very similar to the network from table 4.2, but provides one extra Convolutional Layer and one extra Fully-connected Layer. The network is only used during the dynamic training setup to be able to learn more complex time-dependent tasks from the used 4-Image Speed Representation.

Layer	Type	Activation	Size	Filter Size	Filter Stride
1	Convolution	ReLu	64 Filter	$[8 \times 8]$	$[4 \times 4]$
2	Convolution	ReLu	64 Filter	$[4 \times 4]$	$[2 \times 2]$
3	Convolution	ReLu	32 Filter	$[3 \times 3]$	$[1 \times 1]$
4	Convolution	ReLu	32 Filter	$[2 \times 2]$	$[1 \times 1]$
5	Fully-Connected	ReLu	512 Neurons	-	-
6	Fully-Connected	ReLu	216 Neurons	-	-
7	Fully-Connected	Linear	Output Size	-	-

Table 4.3.: 6-layered image network for the dynamic setup.

5. Evaluation

This chapter presents different training setups for a static and a dynamic scenario. Each training setup will be evaluated in a quantitative as well as in a qualitative way. During the quantitative evaluation, the training, as well as the test results, are presented and discussed. During the qualitative evaluation, the learned policy of the agents is discussed qualitatively by investigating example episodes from the test sets. They show how the agents behave in specific situations. Finally, a selection of relevant agents is applied to the real world and a qualitative assessment is given.

Since Reinforcement Learning has not been applied to the MiR100 robot before, this thesis serves as a proof of concept for using RL during navigation. After the MiR100 software has been successfully integrated with the simulation environment and the stable-baselines library [2], the investigation of an optimal training setup started. During the thesis, the defined goal has been approached slowly by increasing the problem complexity step by step. The following stages of problem complexity have been identified.

1. Following path only in map 4.2b.
2. Navigating along the path with local static obstacles in map 4.2a.
3. Navigating along the path with local static obstacles in map 4.2b.
4. Navigating along the path with moving simple obstacles (single circles) in map. 4.2a.
5. Navigating along the path with moving simple obstacles (single circle) in map 4.2b.
6. Navigating along the path with moving complex obstacles (two walking legs) in map 4.2b. (not solved)
7. Navigating along the path with moving complex obstacles (two walking legs) and local static obstacles in map 4.2b. (not solved)

While solving one stage after another, different variations in the training setup have been applied. Unfortunately, the number of possible variations is almost unmanageable. For that reason, only the evaluation of the most relevant trained agents of stage three and stage five are presented in this chapter.

	agent_1	agent_2	agent_3	agent_4
Action Space	discrete $v_{max} = 0.5$ $\omega_{max} = 0.5$	continuous $v_{max} = 0.5$ $\omega_{max} = 0.5$	discrete $v_{max} = 0.5$ $\omega_{max} = 0.5$	discrete $v_{max} = 0.5$ $\omega_{max} = 0.5$
State Input	1-Image Representation	1-Image Representation	Raw Data Representation	Polar Representation
Network architecture	table 4.2		table 4.1	
Reward function	equation 4.1			
Reward function parameters	table A.2			

Table 5.1.: Agents that are trained in map 4.2b with the static task setup.

5.1. Static Agents

The static agents are all trained in map 4.2b with a static task setup and reward function 4.1. In comparison to the dynamic agents, the static training setup is computational less expensive because there is no time-dependent component. Providing the most current state of the environment should be sufficient for learning the given static task, described in chapter 4.2.

Table 5.1 gives an overview of the static agents that will be evaluated. Agent_1 uses discrete action space and the 1-Image Representation as state input combined with the Neural Network Architecture of table 4.2. Agent_2 uses the same input state and network architecture as agent_1, but applies continuous action space instead. On the contrary, agent_3 and agent_4 both apply the same action space as agent_1, but use different state inputs: the Raw Data Representation and the Polar Representation, that are both combined with the Neural Network Architecture of table 4.1.

5.1.1. Quantitative Evaluation

All static agents have been trained with the PPO1-algorithm from the stable-baselines library with the same hyperparameters, that can be found in the appendix in table A.1, A.2 and A.3. Each agent has been trained for 10 million times steps on CPU only. Furthermore, three agents per training setup have been trained to gain knowledge about the variance of the training results. I am aware, that a higher number of agents per training setup provides a more reliable statement about reproducability, but unfortunately the total training time per agent did not allow more training sessions. The total training time for each agent is between 2 to 5 days, depending on the input state size. Table 5.2 gives an overview of the average training time for each agent setup.

agent	$\sim$ training time
agent_1	4d 18h 49m
agent_2	5d 15h 38m
agent_3	2d 17h 13m
agent_4	3d 0h 31m

Table 5.2.: Average training time for each training setup.

Figure 5.1 summarizes the learning progress of all static agents in comparison. Instead of presenting the reward over time, the success rate is shown to provide universal results. Like this, they can be compared with other approaches, using different reward functions. An episode is a success, if the agent reaches the goal while keeping a minimum distance of 0.56 m to all obstacles. The success rate is averaged over all agents per agent setup and finally, the moving average over 500 consecutive episodes is determined. The error band of each curve shows the variance over all agents per agent setup. Agent_1 achieves the highest success rate with $\sim$ 0.76. During the first 3M time steps, the success rate increases drastically. Thereafter, it starts to settle with a slight remaining increase. Note that the variance is low compared to the other agents so that a stable reproducibility can be expected. Agent_2 reaches only a success rate of $\sim$ 0.58 within 10M time steps. The learning curve does not have such a steep rise compared to the curve of agent_1. After the first settle phase at time step 4M, the learning curve continues to increase slowly. The reason for the slower learning of agent_2 is the higher problem complexity due to its continuous action space. Agent_4 reaches a similar success rate like agent_2, but uses the laser scan data directly with the Polar State Representation. Agent_3 provides the lowest final success rate with $\sim$ 0.2 and a relatively high variance due to unstable training over the different agents per agent setup. Considering agent_3 and agent_4, it can be assumed, that providing the laser scan and waypoint data in the same format – like in the Polar State Representation – is more effective than the Raw State Representation.

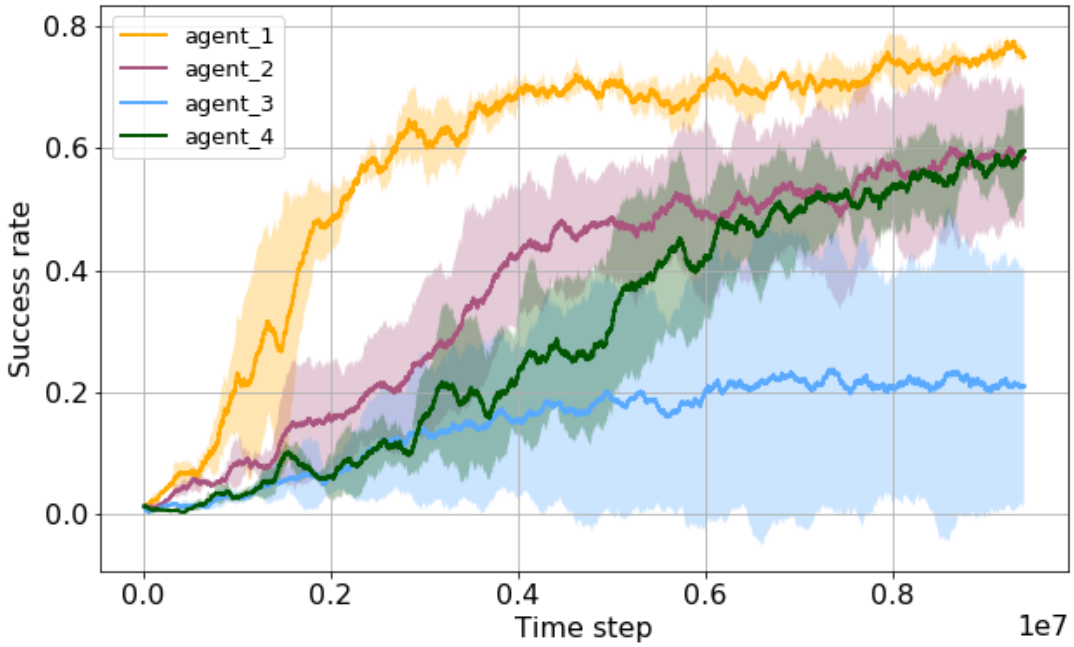


Figure 5.1.: Training results of the four different static agent setups. For each training setup, three agents are trained and the average success rate is taken. Finally, the moving average over 500 consecutive episodes is plotted, while the error band represents the variance over all three agents.

All trained agents have been tested in three worlds of different environment complexities: map 4.2a (simple), map 4.2c (average) and map 4.2d (complex). Each trained agent had to solve 300 static tasks per world. The tasks have been saved in an evaluation set so that each agent had to solve the exact same tasks with the same local static objects. Moreover, the test set only contains solvable episodes. Additionally, the two variants of the traditional local planner approach 2S-VFH*-R and 2S-VFH*, introduced in chapter 1.3, solved the same tasks. During testing an episode is a success, if the robot reaches the goal without colliding with any obstacles. The graph in figure 5.2 shows the average success rate for all agents in the three different worlds, while the dark bars represent the true success rate and the light bar represent the sum of the success and time-exceeded rate. In the simple world agent_1, agent_2 and agent_4 achieve with a true success rate over 0.95 a slightly better result than the traditional 2S-VFH*-R, that even uses global re-planning for recovery. As expected, the performance of agent_3 is relatively low with a true success rate of 0.52. The average-complex world provides more challenging tasks — objects cannot always be avoided from both sides. All success rates decrease from the simple to the world of average complexity, but the true success rate of agent_1 decreases only by 0.02, outperforming the rest. Since the agents are trained in a world with similar complexity to the average-complex world, the performance of agent_1, agent_2 and agent_3 drops stronger than the performance of 2S-VFH*-R from the average to the com-

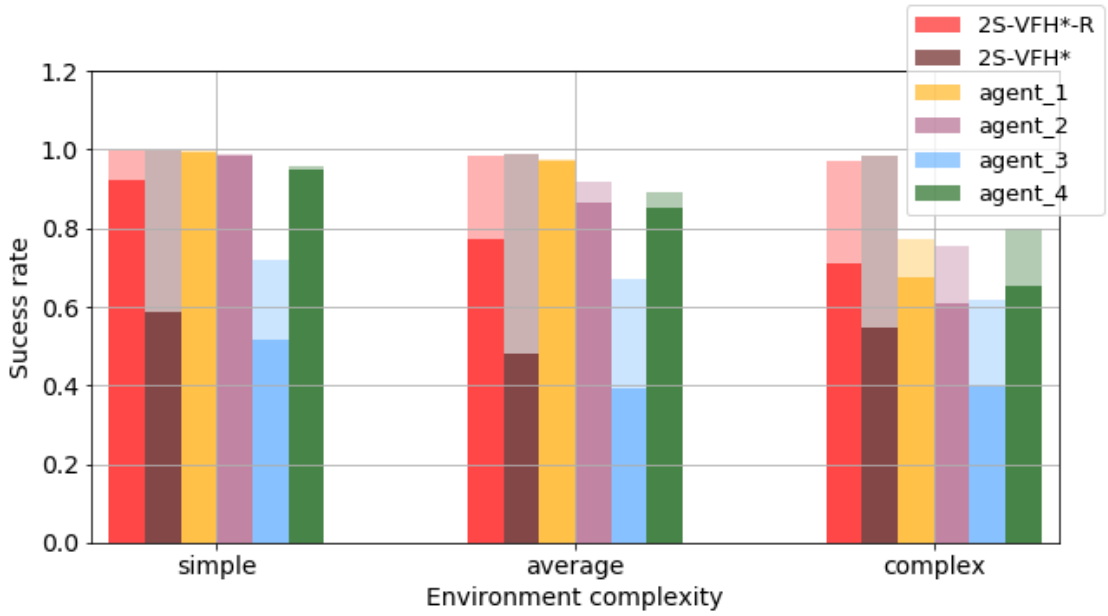


Figure 5.2.: Test results of the four different static agent setups as well as two variants of the traditional local planner 2S-VFH*-R and 2S-VFH*. All trained agents solved each 300 static tasks in three worlds of different complexities. For each world and each agent the average success rate is shown.

plex world. The complex world provides more unseen situations, to those the agents did not generalize. The main challenge are narrow passages that need to be passed to reach the goal. On the contrary, the performance of 2S-VFH*-R and 2S-VFH* does not change significantly, because the VFH* approach is able to handle narrow passages better. It is also striking that the traditional approaches have a relative high time-exceeded rate. One reason for such a low collision rate is an additional stopping mechanism on top of VFH* that stops the robot if the distance to the obstacles falls below a certain threshold. This stopping mechanism could easily be applied to the RL-agents and prevent a majority of collisions, but eventually also reduces the success rate.

5.1.2. Qualitative Evaluation

To fully evaluate the results of the agents, it is not sufficient to only consider the quantitative results. It is also interesting and necessary to look into the learned policy, i.e., driving behavior. In the course of work, identifying situations, in that the agent fails frequently, gave new ideas for an improved training setup. In the following, example tasks will be presented for the different agent setups. Note that each task includes the global plan (blue) and static objects (green and black).

Agent_1 vs. 2S-VFH*-R

Figure 5.3 provides example episodes of the driving behavior of agent_1 (orange) in comparison to the traditional planner with recovery 2S-VFH*-R (red). Episodes with isolated,

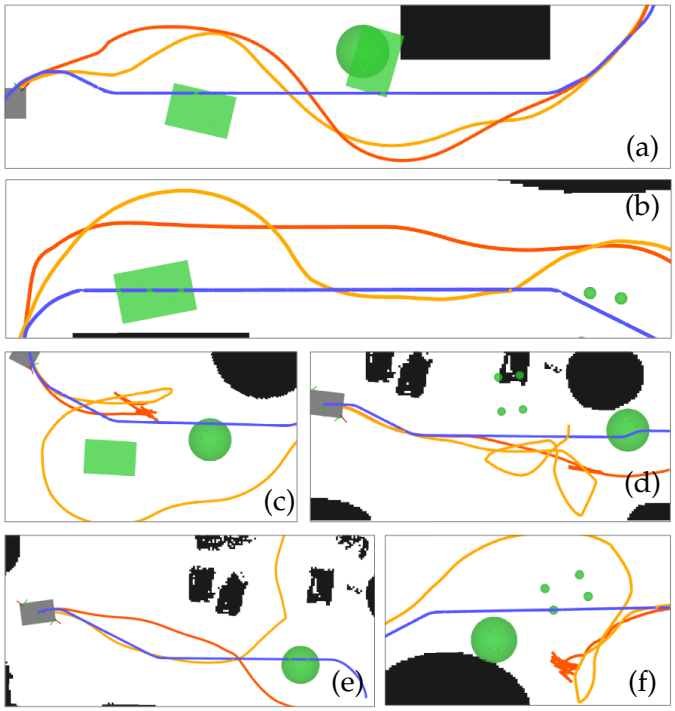


Figure 5.3.: A set of selected example episodes is presented to show the behavior of agent_1. Each test episodes includes the global plan (blue), leading from the left to the right, with global static and local static obstacles (black and green). The driven path of agent_1 (orange) is presented in comparison to the driven path of the traditional path planning approach 2S-VFH*-R (red).

single objects are resolved very robustly from both approaches. They both avoid the obstacles with a similar reasonable avoidance radius. Figure 5.3b presents a task with two single isolated objects with a rather high distance in between. After avoiding the first object 2S-VFH*-R does not drive back to the path but rather continues parallel to the original global path until it reaches the second obstacle. In contrary, agent_1 gets back to the global path, if the distance between the local static objects allows that. Its behavior can be explained with the applied reward function 4.1, because the agent gets a higher reward for driving through the waypoints on the path. Figure 5.3c and 5.3f provide tasks where the objects are only avoidable from one side. If 2S-VFH*-R chooses to avoid the obstacle from the wrong side, it gets stuck on the blocked side of the object and can not recover. Agent_1 is able to recognize if a passage is not passable and tries the other side of the object. Nevertheless, in some situations, agent_1 should be able to recognize the blocked side of the object in advance and directly choose the correct side for avoidance. That is not always the case. The task in figure 5.3d expects the robot to pass a narrow gap, that agent_1 is not capable of. Instead, it drives forth and back in front of the gap. In figure 5.3e agent_1 gets lost in a one-way passage. This happens frequently in the complex world (map 4.2d), because the complex world includes lots of one-way hallways, the robot has not been trained on in the training map 4.2b.

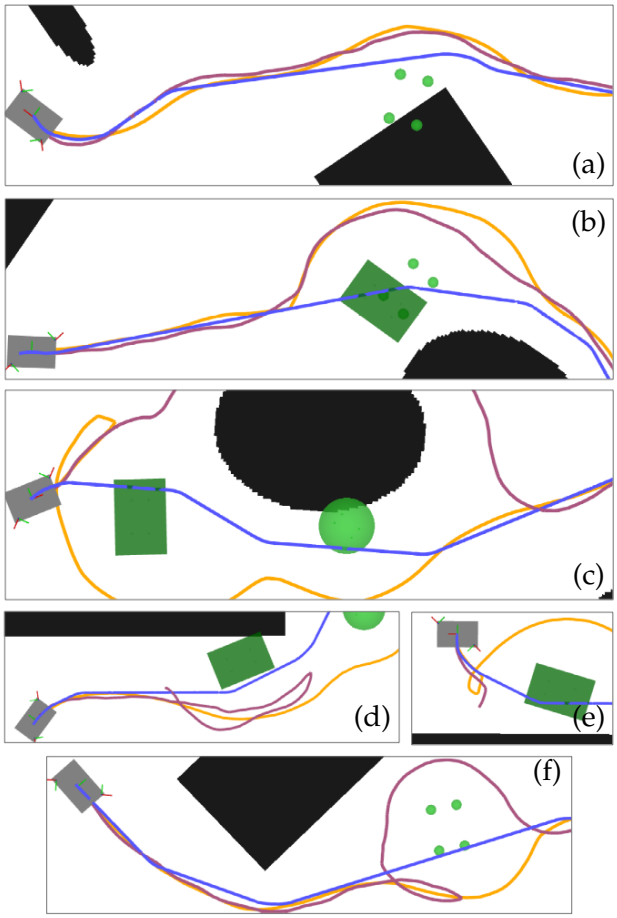


Figure 5.4.: A set of selected example episodes is presented to show the behavior of agent_2. Each test episodes includes the global plan (blue), leading from the left to the right, with global static and local static obstacles (black and green). The driven path of agent_2 (purple) is presented in comparison to the driven path of agent_1 (orange).

Agent_2 vs. Agent_1

Figure 5.4 summarizes the similarities and differences between agent_1 (orange) and agent_2 (purple) in their behavior of solving tasks. Figure 5.4a and 5.4b show, that agent_2 overcomes single isolated objects also very stable and reasonable. Agent_2 oscillates stronger during simple driving maneuvers, compared to agent_1. Figure 5.4c, 5.4d and 5.4f show that the recovery behavior of agent_2 is not well learned. In figure 5.4c agent_2 avoids the whole global obstacle instead of recovering and trying it from the other side of the local obstacle. This is especially problematic with non-closing global obstacles like in map 4.2d. In 5.4d and 5.4f agent_2 decides to drive back and avoid the obstacle from other sides, although it is not absolutely necessary. Note that agent_2 fails in general more tasks than agent_1 like in figure 5.4e.

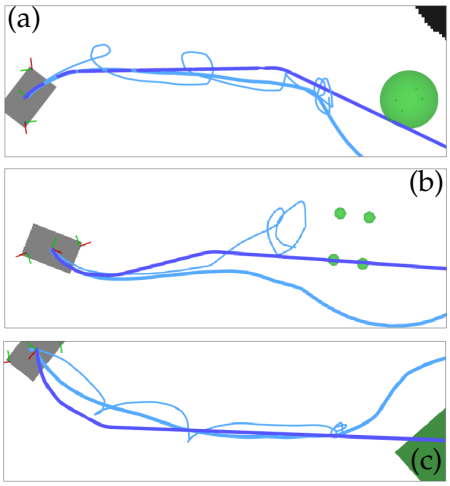


Figure 5.5.: A set of selected example episodes is presented to show the behavior of agent_3. Each test episodes includes the global plan (dark blue), leading from the left to the right, with global static and local static obstacles (black and green). Two trained agents from the same agent setup are presented, while one is displayed with a thin light blue line and the other with a thick light blue line.

Agent_3

The quantitative evaluation of agent_3 already showed that a low success rate is reached compared to the other agent setups. Besides, it shows the highest variance. In figure 5.5, two trained agents of the agent_3 setup are compared: Agent_3_1 is represented with a thin light blue line and agent_3_2 with a thick light blue line. While agent_3_2 has a reasonable driving behavior and overcomes simple tasks, agent_3_1 learned some unnatural movements. It seems that agent_3_1 ends up in a local minima because the agent does not improve for 6M time steps. Agent_3_1 has a stronger focus on "catching" waypoints than avoiding obstacles or driving along the path as fast as possible. Figure 5.5a and 5.5c show, that agent_3_1 tends to drive back when it overshoots a waypoint to catch it. In figure 5.5b it remains in front of the obstacle, because a waypoint is positioned right before the obstacle. It tries over and over to catch it without colliding with the local obstacle.

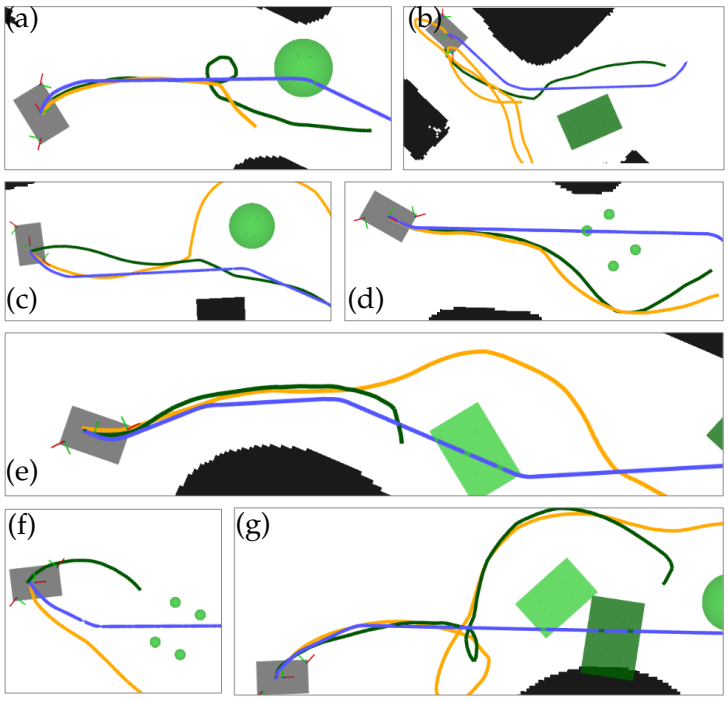


Figure 5.6.: A set of selected example episodes is presented to show the behavior of agent_4. Each test episodes includes the global plan (blue), leading from the left to the right, with global static and local static obstacles (black and green). The driven path of agent_4 (dark green) is presented in comparison to the driven path of agent_1 (orange).

Agent_4 vs. Agent_1

The main difference between agent_4 and agent_1 is that agent_4 is slightly better in driving through narrow gaps. Figure 5.6a to 5.6c show example tasks, where agent_4 was able to pass the narrow gap, while agent_1 collides or finds an alternative route. Agent_4 is more risky regarding narrow passages, leading to a higher collision rate. It also tries to pass narrow passages that are not passable like in figure 5.6e to 5.6g. Agent_4 collides more frequently with objects of the type 4-legged wagon. It can be supposed, that the object has a certain angle to the robot and agent_4 assumes that the gap is passable.

5.2. Dynamic Agents

The dynamic agent setup aims to learn how to drive along a path with walking and standing pedestrians as obstacles. For that reason, the dynamic task setup of chapter 4.2 is applied.

The evaluation of the static training gives insights about the best possible action space as well as the best possible state representation. Since agent_1 outperformed the other static agent setups, all dynamic agents are trained with discrete actions and a 4-Image Speed Representation as input state. By providing more than one image, a time compo-

	agent_5	**agent_6**	**agent_7**
Pedestrian behavior	Polite	Semi-polite	
Reward function	equation 4.10		
Reward function parameters	table B.2		table B.3
Action Space	discrete $v_{max} = 0.5$ $\omega_{max} = 0.7$		discrete $v_{max} = 0.5$ $\omega_{max} = 0.7$ + [0.09, 0]
State Input	4-Image Speed Representation		
Network architecture	table 4.3		

Table 5.3.: Dynamic agent setups trained in map 4.2a.

nent is integrated into the input data, and the agent should be able to cope with moving pedestrians. A higher input size results in longer training periods. To counter the increasing training time, the PPO2-algorithm from the stable-baselines library is applied. The main difference between PPO2 and PPO1 is that PPO2 allows training in parallel environments. Collecting episodes is accelerated and compensates the longer network updates.

Table 5.3 gives an overview of the dynamic agent setups that will be evaluated in this section. All agents have the same state input with the 4-Image Speed Representation fed to the network architecture in table 4.3. Like in the static setup, the dynamic agents are trained in the map 4.2b with average complexity. agent_5 is confronted with pedestrians that have a polite walking behavior, i.e., they always avoid the robot if it comes too close. Additionally, reward function 4.10 with the parameter set from table B.2 is applied. agent_6 applies the same parameter set like agent_5, but has to deal with semi-polite pedestrians. agent_6 has to stop for at least 0.8 seconds in advance to motivate the pedestrians to avoid the robot. Normally, reward function 4.10 with parameter set B.2 punishes the RL-agent for being closer than 0.85 m to a pedestrian and the episode is over. This punishment is disabled if the robot stops in time – 0.8 seconds before a pedestrian enters the critical zone. agent_7 deals as well with semi-polite pedestrians, but applies a different reward parameter set listed in table B.3, that is less strict and allows agent_7 to drive slowly ($\leq$ 0.1 m/s) while being close to pedestrians. Still, it is expected that the robot drives slow for at least 0.8 sec in advance to not get punished. Due to the expected slow robot velocity, an additional discrete action [0.09, 0.0] is made available for agent_7.

agent	$\sim$ training time
agent_5	2d 1h 44m
agent_6	2d 4h 47m
agent_7	1d 23h 2m

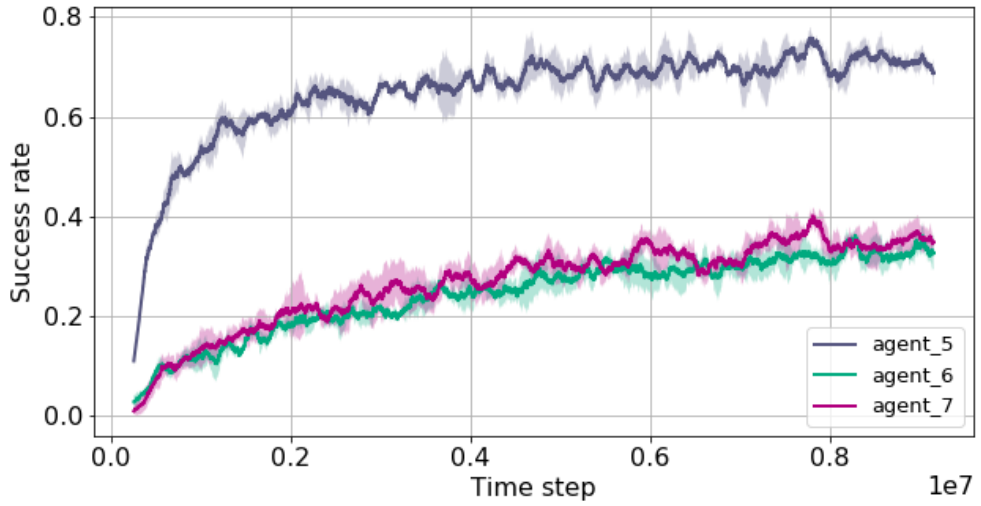
Table 5.4.: Average training time for each dynamic training setup.

5.2.1. Quantitative Evaluation

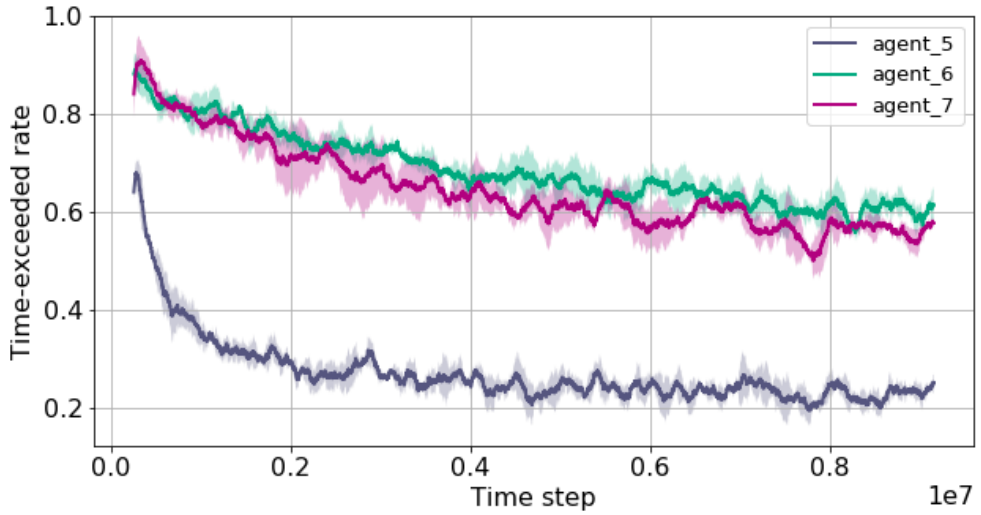
Each dynamic agent has been trained between 8 to 10 million time steps on CPU only. For each training setup three agents of the same agent setup have been trained to show reproducability. The training time of the dynamic agents is shown in table 5.4 and reduced significantly to 1 to 2 days. The main reason for the speedup is the usage of five simulation environments, that parallelizes the process of experience collection. Furthermore, I reduced the image size, while increasing its resolution (see table B.1). Note that the input size is still higher than during the static setup because a stack of four images is provided.

Figure 5.7 shows the training process of all three agent setups, while 5.11a shows the success rate and 5.11b shows the time-exceeded rate over time. An episode counts as success if the RL-agent navigates to the goal without violating any distance threshold to the static or dynamic objects. The time-exceeded rate contains all episodes in which the RL-agent did not finish in time, but also did not come to close to any obstacles. If the RL-agent deviates to strongly from the path, the episode is over and counts also as time-exceeded. The rates are averaged over all agents per agent setup and finally, the moving average over 500 consecutive episodes is determined. The error band of each curve shows the variance over all agents per agent setup. Agent_5 reaches a relatively high success rate of ~ 0.7 while agent_6 and agent_7 only reach a success rate between $\sim 0.3 - 0.4$. Nevertheless, the time-exceeded rate is high for all three agents, resulting in a low collision rate in the later period of the training.

All dynamic agents as well as agent_1 from the static training and the traditional navigation approach 2S-VFH*-R have been tested in a dynamic test setup in the worlds of simple and average complexity (map 4.2a and map 4.2c). They all solve the same 300 episodes per world with a dynamic task setup with semi-polite agents. The success rate during testing differs slightly to the one during training, because an episode counts as success if the robot drives successfully to the goal without colliding with any objects. That means no differentiation between dynamic or static objects is made during testing. Furthermore, the time span for finishing a task is increased and depends on the path length to enable the agents to finish episodes more often. Figure 5.8 shows the average success rate of the dynamic test setup, while the dark bars represent the true success rate and the light bars represent the sum of the success and the time-exceeded rate. Agent_5



(a) Success rate



(b) Time-exceeded rate

Figure 5.7.: Training results of the different dynamic agent setups. For each setup three agents are trained and the average success and time-exceeded rates are determined. Finally, the moving average over 500 consecutive episodes is plotted, while the error band represents the variance over all three agents.

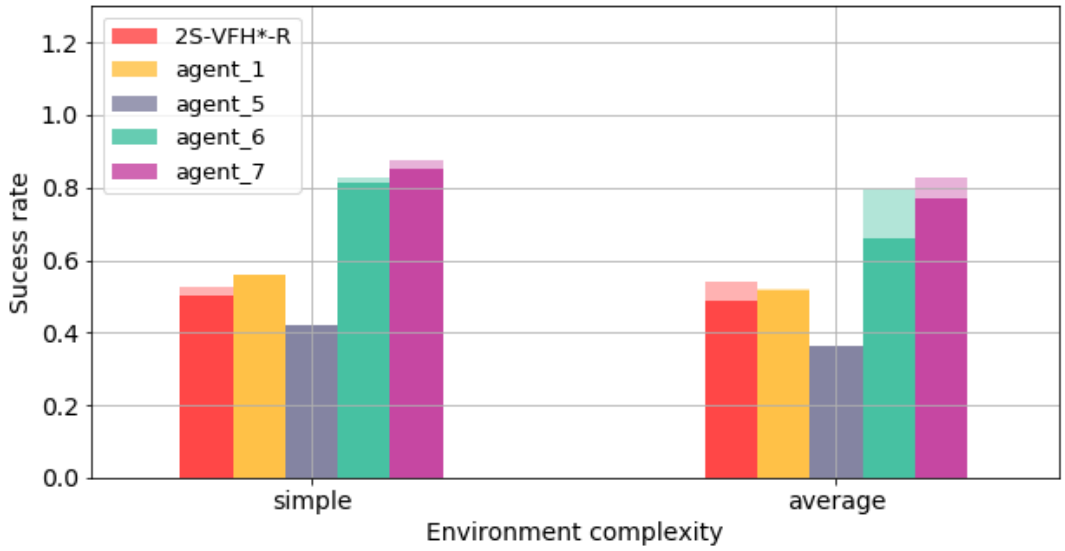


Figure 5.8.: Test results of the three different dynamic agent setups as well as the traditional local planner 2S-VFH*-R and agent_1 of the static agent setup. All approaches solved every 300 dynamic tasks in two worlds of different complexities. For each world and each agent the average success rate is shown.

has the lowest success rate in both worlds. Since agent_5 is trained on polite pedestrians, RL-agent simply drives along the path and expects the pedestrians to avoid the robot. The pedestrians are semi-polite in the test setup, so that agent_5 happens to collide with people more often. The success rates of the static agent_1 and 2S-VFH*-R are slightly better than agent_5 with ≥ 0.5. Agent_6 reaches a success rate of 0.81 in the simple world and 0.66 in the world with average complexity. Agent_7 reaches a higher score with a success rate of 0.85 in the simple world and 0.77 in the world with average complexity. The better results from agent_7 can be explained with the provided pushing mechanism. In a blocked situation, agent_7 can free itself by pushing the pedestrians away. In contrary, agent_6 needs to wait for the situation to resolve on its own. The resolution of critical situations is not always guaranteed or can be very slow, so that agent_6 can not finish those episodes in time. It results in a relatively high time-exceeded rate in the average-complex world of agent_6 with 0.14.

5.2.2. Qualitative Evaluation

Although the results of the qualitative evaluation already indicate which training setup leads to good performance, it is essential to investigate the qualitative driving behavior. In the following sections, the learned policy of each dynamic agent is presented qualitatively by showing a selection of solved tasks. Note that each task includes the global plan (blue) leading among static objects (black). The state of the robot (grey rectangle) and the pedestrians (red circle) is shown at a certain relevant time step. The red arrow

indicates the moving direction of the pedestrians, while the length shows the speed of the pedestrians. In addition, the red fine line indicates the pedestrian's walking path of the future.

Agent_5

The usage of polite pedestrians in the simulation environments influences the resulting driving behavior of agent_5 significantly. When running agent_5 in a world with semi-polite pedestrians, it is obvious that the agent did not learn to avoid humans confidently. Most of the time, the agent drives along the global plan and expects the pedestrians to give way to the robot. In clear situations agent_5 is able to avoid slow moving and non-moving pedestrians.

Agent_6

Figure 5.9 shows example episodes, while the mint green path is the taken path of agent_6. In figure 5.9a, agent_6 solves an episode with a high number of pedestrians, resulting in a constant deviation from the planned path. In figure 5.9b, agent_6 shows a forward-looking avoidance behavior, although it could be expected that the robot stays on the path because the two pedestrians are walking away from the path and they will be gone by the time the robot reaches there. The forth and back walking behavior of the simulated pedestrians can be the reason for that behavior. To be able to react in time on a change of direction, agent_6 keeps a certain safety distance. In figure 5.9c and 5.9d, the RL-agent stops and waits for the pedestrians to pass and finally continues. This behavior is essential in highly crowded situations with no safe way out. Figure 5.9e and 5.9f show two reasonable driving maneuver.

Figure 5.9g to 5.9j show, that agent_6 does not behave confidently regarding pedestrians crossing the path. It is more likely to fail in those situations. The agent does not seem to consider the walking direction of the pedestrian, because it tries to avoid the pedestrian from the same side it walks to. In figure 5.9i agent_6 first tries to avoid the pedestrian from the "wrong" side and after the pedestrian has crossed the path, it changes its avoidance strategy to the other side. This behavior happens very frequently.

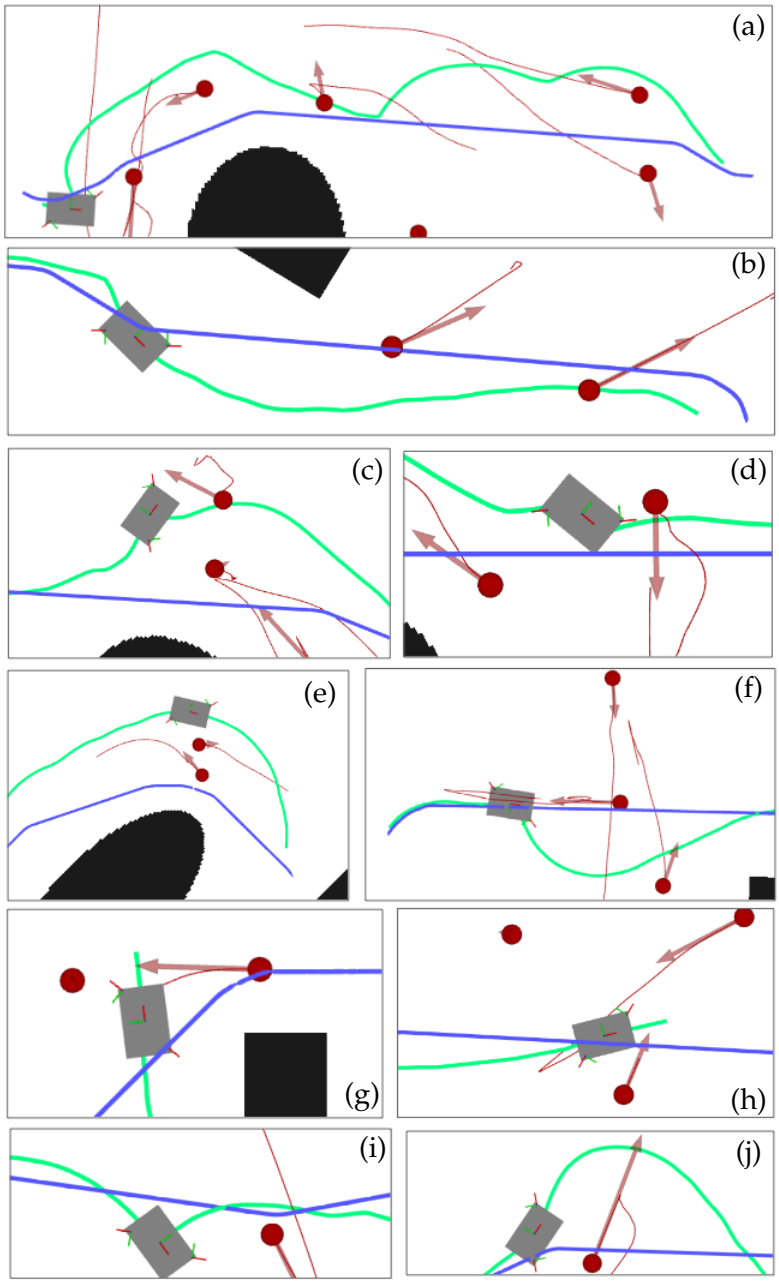


Figure 5.9.: A set of selected example episodes is presented to show the behavior of agent_6. Each test episodes includes the global plan (blue), leading from the left to the right, with global static (black) and moving pedestrians (red circle). The arrow indicates the moving direction, while the length shows the speed of the pedestrian. In addition, the fine red line indicates the future walking path for each pedestrian. The mint green line is the taken path of agent_6 (grey rectangle).

Agent_7

Compared to agent_6, the behavior of agent_7 is more aggressive. Due to the different parameter set of the reward function 4.10 as well as the additional discrete action $[0.09, 0.0]$, the agent is able to push people away if it drives slowly towards them. The agent is supposed to use this mechanism to free itself from crowded situations. Unfortunately, agent_7 pushes pedestrians not only in critical situations but also in situations, where a simple avoidance maneuver is more reasonable. Figure 5.10 shows sample episodes of agent_7, while the purple path represents the taken path of agent_7. Figure 5.10a to 5.10d show episodes, where agent_7 avoids the moving objects in a reasonable way and with an appropriate safety distance to the pedestrians. Figure 5.10e and 5.10f show two scenes, where the robot pushes pedestrians away. It is difficult to visualize but can be recognized by the future path of the pedestrians (thin red line), that is not straightforward like usual. In figure 5.10e, the pushing mechanism has been applied correctly because the passage is fully blocked, while in figure 5.10f agent_7 could have simply avoided the crowd from the left side.

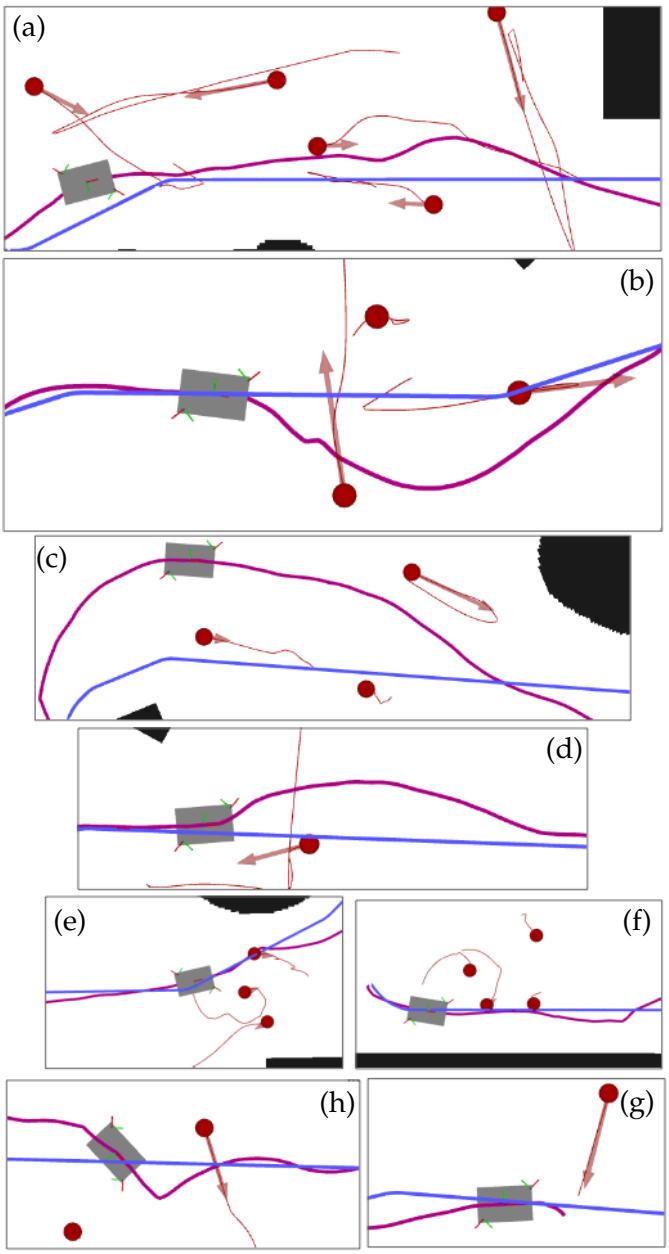


Figure 5.10.: A set of selected example episodes is presented to show the behavior of agent_7 (purple). Each test episodes includes the global plan (blue), leading from the left to the right, with global static obstacles (black) and moving pedestrians (red circle). The arrow indicates the moving direction, while the length shows the speed of the pedestrians. In addition, the red fine line indicates the future walking path for each pedestrian. The purple line is the taken path of agent_7 (grey rectangle).

5.3. RL-Agent in the Real World

The transition from the simulation environment to the real world is a particular challenge in the field of robotics, but absolutely necessary. Due to the heavy training times and

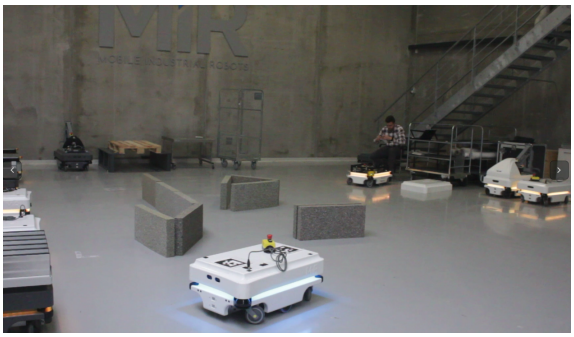
safety issues, it is not possible to train robots in the real world. Alternative approaches like Imitation Learning or a combination of Reinforcement Learning and Imitation Learning aim to speed up the training process.

In this thesis, pure Deep Reinforcement Learning is applied. To reduce the gap between the simulation environment and the real world, the simulation has been restricted to a 2D world. 2D laser scan sensors are used to perceive the environment because they get closer to real world data than other sensors like stereo or depth cameras. Furthermore, Gaussian noise is added to the simulated sensor data to approximate the real world sensors more realistically. During the simulation, physical properties like e.g. robot mass, motor friction, friction coefficients of the wheels have been ignored completely to simplify and speed up the simulation. Training in simulation environments offers more advantages: It is possible to distinguish between different obstacles in the scan data. The rewarding made use of it by allowing the robot to get closer to static objects than to dynamic object. Obviously, the distinction is not possible in the real world and consequently, it is not possible to continue the training in the real world with the exact same reward function. Another challenge is to create realistic tasks during training. The static task setup is very clean and spacious, while in industrial real world scenarios the environment is more cluttered and narrow. While creating a realistic static task setup is still fairly easy and solvable, constructing a real world dynamic setup is very challenging. Human behavior is not deterministic or physically describable. There are various scenarios that can be considered. The following categories have been identified:

1. Humans that block intentionally the way of the robot to explore its limits
2. Humans that expect the robot to give way.
3. Humans that naturally avoid the robot.
4. Group of humans that stands around.
5. Group of humans that walks together around.

In the simulation environment, I modeled category two and three. To speed up the simulation, humans have only been spawned near the planned path, walking back and forth between two waypoints to increase the chance of a clash between robot and pedestrian. Hence, the RL-agent often faces changes of direction, that seem rather rare in the real world. Also, the pedestrians are walking on a similar speed to the maximum robot speed of 0.5 m/s, that is a fairly low speed. Another difference to the real world is that the pedestrians are approximated with a single circle representing both legs instead of modeling two reasonable moving legs.

The strict time frame of the master thesis did not allow to setup a proper evaluation procedure for testing the different agent setups in the real world. Nevertheless, agent_1, agent_6 and agent_7 are applied to the MiR100 robot and a qualitative proof-of-concept of the behavior in the real world is provided in this chapter. Furthermore, a video is



(a) Example static task.

(b) Example dynamic task.

Figure 5.11.: Real world setup.

submitted that shows the robot in action and provides a selection of episodes for all three agents. The testing environment is a spacious room, that contains multiple robots and some boxes and pallets, standing close to the walls. Static scenes similar to the trained environments have been constructed by composing foam blocks. As dynamic object, I walked with a low speed of ~ 0.5 m/s along and across the planned path to force the robot to avoid me. Situations with more than one pedestrian have not been tested. Figure 5.11 shows an example static as well as an example dynamic task in the real world.

The behavior of agent_1 on the MiR100 is very similar to the behavior in the simulation. It avoids static objects reliably and is even able to overcome difficult situations. If objects are positioned close to a wall and the object is only avoidable from one side, agent_1 is able to identify the side with enough space. If the robot gets stuck on one side of an obstacle, the RL-agent is able to recover and tries the other side of the obstacle.

Although pedestrians are only modeled with a single circle, agent_6 is able to deal with pedestrians walking on low speed. When a pedestrian crosses the path, the robot either stops and waits for the pedestrian to pass or tries to avoid the pedestrian from the side the pedestrian walks to. At a certain point, it realizes to better avoid the pedestrian from the other side. It is the same undesired behavior as in the simulation (see figure 5.9i). It does not avoid pedestrians that walk along the path very comforting, because agent_6 tends to start the avoidance procedure too late. If a pedestrian is very close to the robot, agent_6 stops and waits for the pedestrian to give room.

Agent_7 does not apply as many avoidance maneuvers but rather makes use of the pushing mechanisms. At least it manages to slow down in time and continues its faster navigation as soon as the pedestrian left.

6. Conclusion and Future Work

In this thesis, Deep Reinforcement Learning (DRL) has been applied in the navigation of a mobile robot. The idea was to keep the traditional global planner, but to replace the traditional local planner with an RL-agent. Its objective is to react to local obstacles, that have not been considered by the global planner. A special focus is set on moving obstacles, that demand an adaptable driving behavior.

In this thesis, a learning infrastructure has been developed, that integrates the Navigation Stack infrastructure of the ROS framework with the DRL-library stable-baselines [2] and a simulation environment that fuses the two simulators Flatland and Pedsim. The state-of-the-art DRL algorithm Proximal Policy Optimization (PPO) has been applied during training.

In the first stage, different RL-agents are trained in an environment of lower complexity with exclusively static objects. Different training setups have been investigated in that stage to identify the optimal training setup. Continuous and discrete action spaces, as well as different representations of the input state, were compared. One static agent setup outperformed the others and applied a discrete action space as well as an image as state input, that contains information about the laser scan data and the global path.

In the second stage, the complexity of the training setup has been increased by modeling dynamically moving and standing pedestrians, behaving according to Helbing's Social Force Model. Moving objects imply time-dependent decisions; therefore the findings of stage one have been extended to a dynamic training setup: A stack of four images at different time steps as well as the robot velocity are provided as state input. Two agents worth mentioning have been trained with different reward function parameters and both achieved a success rate over 65 %, while the traditional local planner only achieves 50 %. The different reward function parameters lead to different learned behaviors. One agent is able to avoid individual humans, but stops and waits if the robot faces unsolvable situations like crowds and blocked passages. The other agent learned a more aggressive policy. It pushes pedestrians by driving very slowly towards them until they give way. Unfortunately, the agent applies the pushing mechanism not only in unsolvable situations but also when simple avoidance would be more reasonable.

Although the obtained results seem very promising, further improvements are necessary before a concrete deployment can be considered. A simplified dynamic task setup has been applied during this thesis. In future work, a more realistic design of the dynamic task setup should be developed. The following improvements have been identified:

- Especially when pedestrians are crossing the robot's way, it seems that the agent does not consider the walking direction of the robot. Further investigation should be made in the proper learning of time-dependent input data.
- The pedestrians should walk on a realistic walking speed at about 0.8 - 1.5 m/s.
- More complex maps with more clutter and narrow corridors should be provided during training. The RL-agent should be able to learn social-relevant behavior like driving on the right side in narrow corridors.
- Only single pedestrians moving and standing around are considered. In further work, various pedestrian behaviors should be modeled, especially socially relevant behaviors like e.g. group walking and group gathering.

In conclusion, the obtained results are very convincing and the proof-of-concept for applying Deep Reinforcement Learning to the navigation of the MiR100 robot is confirmed. A high potential for improvement can be expected if more work and further investments are made in this field. Furthermore, hybrid variants – a combination of the traditional local planner and an RL-agent – should be taken into consideration. As a consequence, the RL-agent could specialize on tasks, that are unsolvable for the traditional planner but does not need to take care of simple tasks, that are fulfilled very well by the traditional local planner like pure path following and avoidance of static objects.

A. Parameters of the static training

Parameter Name	Parameter	Value
look ahead distance	$d_{lookahead}$	1.5 m
image size		$[(80 + 40) \times 100]$
laser scan vector length	l_{laser_vec}	360
number of waypoints	n_{wp}	4
resolution	res	0.05 m/px
max translational velocity	v_max	0.5 m/s
max rotational velocity	ω_{max}	0.5 1/s

Table A.1.: Parameters of the static training setup.

Parameter description	Parameter	Value
constant for reaching the goal	R_g	10
constant for colliding with an obstacle	R_o	15
distance threshold to the objects	D_o	0.96
weight for approaching a waypoint	w_1	2.5
weight for departing from a waypoint	w_2	3.5
constant for reaching a waypoint	R_{wp}	1.0
distance threshold to the wayoints	D_{wp}	0.2

Table A.2.: Parameter set for reward function 1.

Parameter name	Parameter	Value
discount factor	γ	0.99
optimal batchsize	$\text{optim}_batchsize$	4096
clip parameter	ϵ	0.2
entropy loss weight	entcoeff	0.003
optimizer's number of epochs	optim_epochs	4
optimimzer's stepsize	α	0.001
advandtage estimation	λ_{adv}	0.95
epsilon of adam optmizer	ϵ_{adam}	0.00005

Table A.3.: Used PPO1 parameters in the stable baselines library [2].

B. Parameters of the dynamic training

Parameter Name	Parameter	Value
look ahead distance	$d_{lookahead}$	1.5 m
image size		$[(70+20) \times 70]$
number of waypoints	n_{wp}	8
resolution	res	0.15 m/px
max translational velocity	v_{max}	0.5 m/s
max rotational velocity	ω_{max}	0.7 1/s

Table B.1.: Parameters of the dynamic training setup.

Parameter description	Parameter	Value
distance threshold to pedestrians	D_{ped}	0.85
distance threshold to static objects	D_o	0.66
distance threshold to waypoints	D_{wp}	0.2
constant for reaching the goal	R_g	10
constant for being to close to a pedestrian	R_{ped}	7
constant for colliding with static obstacles	R_{so}	15
constant for turning only	R_{vel1}	0.001
constant for standing still	R_{vel2}	0.01
constant for reaching a waypoint	R_{wp}	0.3
reaction time of pedestrians	$t_{reaction}$	0.8
maximal allowed speed close to pedestrians	$v_{reaction,max}$	0.0
weight for approaching a waypoint	w_1	4.5
weight for departing from a waypoint	w_2	5.5

Table B.2.: Parameter set 1 for reward function 2.

Parameter description	Parameter	Value
distance threshold to pedestrians	D_{ped}	0.85
distance threshold to static objects	D_o	0.66
distance threshold to waypoints	D_{wp}	0.2
constant for reaching the goal	R_g	10
constant for being to close to a pedestrian	R_{ped}	7
constant for colliding with static obstacles	R_{so}	15
constant for turning only	R_{vel1}	0
constant for standing still	R_{vel2}	0
constant for reaching a waypoint	R_{wp}	0.3
reaction time of pedestrians	$t_{reaction}$	0.8
maximal allowed speed close to pedestrians	$v_{reaction,max}$	0.1
weight for approaching a waypoint	w_1	4.5
weight for departing from a waypoint	w_2	5.5

Table B.3.: Parameter set 2 for reward function 2.

Parameter Name	Parameter	Value
discount factor	γ	0.99
number of steps per environment per update	n_steps	840
entropy coefficient	ent_coef	0.003
learning rate	learning_rate	0.0001
value function coefficient for loss	vf_coef	0.5
maximum gradient clipping value	max_grad_norm	0.5
lam	lam	0.95
number of minibatches	nminibatches	5
clipping paramater	cliprange	0.2

Table B.4.: Used PPO2 parameters in the stable baselines library [2].

Bibliography

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition, booktitle = Proceedings of the IEEE," 1998, pp. 2278–2324.

[2] A. Hill, A. Raffin, M. Ernestus, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable Baselines," https://github.com/hill-a/stable-baselines, 2018.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[4] J. N. Tsitsiklis and B. Van Roy, "Analysis of Temporal-difference Learning with Function Approximation," in *Proceedings of the 9th International Conference on Neural Information Processing Systems*, ser. NIPS'96. Cambridge, MA, USA: MIT Press, 1996, pp. 1075–1081. [Online]. Available: http://dl.acm.org/citation.cfm?id=2998981.2999132

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[6] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1995–2003. [Online]. Available: http://proceedings.mlr.press/v48/wangf16.html

[7] H. v. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 2094–2100. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016100.3016191

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *Computing Research Repository (CoRR)*, vol. abs/1511.05952, 2015. [Online]. Available: http://arxiv.org/abs/1511.05952

[9] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy Networks for Exploration," *Computing Research Repository (CoRR)*, vol. abs/1706.10295, 2017.

[10] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," *Computing Research Repository (CoRR)*, vol. abs/1707.06887, 2017.

[11] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *Computing Research Repository (CoRR)*, vol. abs/1710.02298, 2017. [Online]. Available: http://arxiv.org/abs/1710.02298

[12] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Mach. Learn.*, vol. 8, no. 3-4, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[13] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html

[14] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *Computing Research Repository (CoRR)*, vol. abs/1506.02438, 2015.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *Computing Research Repository (CoRR)*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2016.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *Computing Research Repository (CoRR)*, vol. abs/1602.01783, 2016.

[18] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine, "QT-Opt: Scalable

Deep Reinforcement Learning for Vision-Based Robotic Manipulation," *Computing Research Repository (CoRR)*, vol. abs/1806.10293, 2018. [Online]. Available: http://arxiv.org/abs/1806.10293

[19] OpenAI, "Learning Dexterous In-Hand Manipulation," *Computing Research Repository (CoRR)*, vol. abs/1808.00177, 2018. [Online]. Available: http://arxiv.org/abs/1808.00177

[20] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-Real Robot Learning from Pixels with Progressive Nets," *Computing Research Repository (CoRR)*, vol. abs/1610.04286, 2016. [Online]. Available: http://arxiv.org/abs/1610.04286

[21] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," *Computing Research Repository (CoRR)*, vol. abs/1707.02286, 2017. [Online]. Available: http://arxiv.org/abs/1707.02286

[22] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, "DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 41:1–41:13, Jul. 2017. [Online]. Available: http://doi.acm.org/10.1145/3072959.3073602

[23] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving," *Computing Research Repository (CoRR)*, vol. abs/1610.03295, 2016. [Online]. Available: http://arxiv.org/abs/1610.03295

[24] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep Reinforcement Learning framework for Autonomous Driving," *Computing Research Repository (CoRR)*, vol. abs/1704.02532, 2017.

[25] S. Han, H. Choi, P. Benz, and J. Loaiciga, "Sensor-Based Mobile Robot Navigation via Deep Reinforcement Learning," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, Jan 2018, pp. 147–154.

[26] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation," *Computing Research Repository (CoRR)*, vol. abs/1703.00420, 2017. [Online]. Available: http://arxiv.org/abs/1703.00420

[27] P. Long, T. Fan, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning," *Computing Research Repository (CoRR)*, vol. abs/1709.10082, 2017. [Online]. Available: http://arxiv.org/abs/1709.10082

[28] N. V. Dinh, N. H. Viet, L. A. Nguyen, H. T. Dinh, N. T. Hiep, P. T. Dung, T. Ngo, and X. Truong, "An extended navigation framework for autonomous mobile robot in dynamic environments using reinforcement learning algorithm," in *2017 International Conference on System Science and Engineering (ICSSE)*, July 2017, pp. 336–339.

[29] Y. Kato, K. Kamiyama, and K. Morioka, "Autonomous robot navigation system with learning based on deep Q-network and topological maps," in *2017 IEEE/SICE International Symposium on System Integration (SII)*, Dec 2017, pp. 1040–1046.

[30] T. Fan, X. Cheng, J. Pan, P. Long, W. Liu, R. Yang, and D. Manocha, "Getting Robots Unfrozen and Unlost in Dense Pedestrian Crowds," *Computing Research Repository (CoRR)*, vol. abs/1810.00352, 2018. [Online]. Available: http://arxiv.org/abs/1810.00352

[31] L. Xie, S. Wang, A. Markham, and N. Trigoni, "Towards Monocular Vision based Obstacle Avoidance through Deep Reinforcement Learning," *Computing Research Repository (CoRR)*, vol. abs/1706.09829, 2017. [Online]. Available: http://arxiv.org/abs/1706.09829

[32] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, "End-to-End Race Driving with Deep Reinforcement Learning," *CoRR*, vol. abs/1807.02371, 2018. [Online]. Available: http://arxiv.org/abs/1807.02371

[33] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J. Allen, V. Lam, A. Bewley, and A. Shah, "Learning to Drive in a Day," *CoRR*, vol. abs/1807.00412, 2018. [Online]. Available: http://arxiv.org/abs/1807.00412

[34] A. Folkers, "Steuerung eines autonomen Fahrzeugs durch Deep Reinforcement Learning," Master's thesis, University Bremen, Bremen, Germany, 2018.

[35] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-Regret Reductions for Imitation Learning and Structured Prediction," *Computing Research Repository (CoRR)*, vol. abs/1011.0686, 2010.

[36] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning," *Computing Research Repository (CoRR)*, vol. abs/1606.03476, 2016.

[37] A. Y. Ng and S. J. Russell, "Algorithms for Inverse Reinforcement Learning," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 663–670. [Online]. Available: http://dl.acm.org/citation.cfm?id=645529.657801

[38] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard, "Socially Compliant Mobile Robot Navigation via Inverse Reinforcement Learning," *The International Journal of Robotics Research*, 2016.

[39] P. Abbeel, D. Dolgov, A. Y. Ng, and S. Thrun, "Apprenticeship learning for motion planning with application to parking lot navigation," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2008, pp. 1083–1090.

[40] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. A. Riedmiller, "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards," *Computing Research Repository (CoRR)*, vol. abs/1707.08817, 2017.

[41] M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. I. Nieto, "Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Mapless Navigation by Leveraging Prior Demonstrations," *Computing Research Repository (CoRR)*, vol. abs/1805.07095, 2018.

[42] C. Cheng, X. Yan, N. Wagener, and B. Boots, "Fast Policy Learning through Imitation and Reinforcement," *Computing Research Repository (CoRR)*, vol. abs/1805.10413, 2018.

[43] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99).*, July 1999.

[44] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[45] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with Provable Bounds on Sub-Optimality ," in *Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference (NIPS-03)*. MIT Press, 2004.

[46] I. Ulrich and J. Borenstein, "Vfh*: local obstacle avoidance with look-ahead verification," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 3, April 2000, pp. 2505–2511 vol.3.

[47] A. Karpathy, "CS231n Convolutional Neural Networks for Visual Recognition," http://cs231n.github.io/, [Online; accessed 2018-06-14].

[48] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[49] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[50] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–, Oct. 1986. [Online]. Available: http://dx.doi.org/10.1038/323533a0

[51] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[52] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167

[53] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 779–788.

[54] "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, author = Ren, Shaoqing and He, Kaiming and Girshick, Ross and Sun, Jian, booktitle = Advances in Neural Information Processing Systems 28, editor = C. Cortes and N. D. Lawrence and D. D. Lee and M. Sugiyama and R. Garnett, pages = 91–99, year = 2015, publisher = Curran Associates, Inc., url = http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf."

[55] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," *Computing Research Repository (CoRR)*, vol. abs/1512.02325, 2015. [Online]. Available: http://arxiv.org/abs/1512.02325

[56] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017, pp. 2980–2988.

[57] A. W. Harley, K. G. Derpanis, and I. Kokkinos, "Segmentation-Aware Convolutional Networks Using Local Attention Masks," *Computing Research Repository (CoRR)*, vol. abs/1708.04607, 2017. [Online]. Available: http://arxiv.org/abs/1708.04607

[58] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. [Online]. Available: http://incompleteideas.net/book/RLbook2018.pdf

[59] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html

[60] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14.

JMLR.org, 2014, pp. I–387–I–395. [Online]. Available: http://dl.acm.org/citation.cfm?id=3044805.3044850

[61] G. E. Uhlenbeck and L. S. Ornstein, "On the Theory of the Brownian Motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.36.823

[62] Avidbots, "Flatland," https://github.com/avidbots/flatland, 2018.

[63] B. Okal, T. Linder, D. Vasquez, and L. P. Sven Wehner, Omar Islas, "pedsim_ros," https://github.com/srl-freiburg/pedsim_ros, 2018.

[64] E. Rohmer, S. P. N. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov 2013, pp. 1321–1326.

[65] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, Sep. 2004, pp. 2149–2154 vol.3.

[66] P. M. Dirk Helbing, "Social force model for pedestrian dynamics," 1998. [Online]. Available: https://arxiv.org/abs/cond-mat/9805244

[67] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[68] Unknown, "stdr_simulator," https://github.com/stdr-simulator-ros-pkg/stdr_simulator, 2014.

[69] A. Yorozu, T. Moriguchi, and M. Takahashi, "Improved Leg Tracking Considering Gait Phase and Spline-Based Interpolation during Turning Motion in Walk Tests," *Sensors*, vol. 15, no. 9, pp. 22 451–22 472, 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/9/22451

[70] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[71] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Syst. J.*, vol. 4, no. 1, pp. 25–30, Mar. 1965. [Online]. Available: http://dx.doi.org/10.1147/sj.41.0025

[72] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *CoRR*, vol. abs/1312.6120, 2013. [Online]. Available: http://arxiv.org/abs/1312.6120

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Odense, den _____ Unterschrift: _____