DEEP REINFORCEMENT LEARNING FOR AUTONOMOUS NAVIGATION OF MOBILE ROBOTS IN INDOOR ENVIRONMENTS

A Thesis

by

GARGI YATIN VAIDYA

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Dileep Manisseri Kalathil
Committee Members,	Srinivas Shakkottai
	Srikanth Saripalli
Head of Department,	Miroslav Begovic

December 2021

Major Subject: Computer Engineering

ABSTRACT

Conventional autonomous navigation framework for mobile robots is highly modularized with various subsystems such as localization, perception, mapping, planning and control. Although these provide easy interpretation, they are highly dependent on a known map of the robot's surroundings for navigating in a cluttered environment. Local planners such as DWA require a map with all obstacles in the surroundings to calculate an optimal collision-free trajectory to the goal. Planning and tracking a collision-free path without knowing the obstacle locations is a challenging task.

Since the advent of deep learning techniques, the field of deep reinforcement learning has proven to be a powerful learning framework for robotic tasks. Deep Reinforcement Learning has demonstrated wide success in various complex computer games such as Go and StarCraft which have high dimensional state and action spaces. However, it has rarely been used in real world applications due to the Sim-2-Real challenges in transferring the trained RL policy into the real-world.

In this work, we propose a novel framework for autonomously navigating a mobile robot in a cluttered space without known localisation of the obstacles in its surroundings using deep reinforcement learning techniques. The proposed method is a modular and scalable approach due to a strategic design of the training environment. It uses constrained space and randomization techniques to learn an effective reinforcement learning policy in lesser simulation training time. The state vector consists of the target location in the mobile robot coordinate frame and additionally a 36 dimensional lidar vector for obstacle avoidance task. We demonstrate the optimal discrete action policy on a Turtlebot in the real-world. We have also addressed some key challenges in robot pose estimation for autonomous driving tasks.

DEDICATION

To my mother,

for her constant encouragement & for being my biggest inspiration.

ACKNOWLEDGMENTS

Braving a Texas winter storm and a global pandemic, this research work would not have been possible without the constant guidance by my advisor, Dr. Kalathil. I would like to thank him for giving me this research opportunity and showing confidence in my potential. I am thoroughly grateful to him for making various resources available to me, and guiding me at every step in my research journey at Texas AM University. I would also like to thank Dr. Shakkottai for the weekly insightful discussions with the entire project group. Special thanks to Akshay Sarvesh for introducing me to the world of ROS which opened the door for exciting opportunities in the Robotics domain; and Desik Rengarajan for always being encouraging and clarifying my Reinforcement Learning doubts in the simplest words. In closing, I would like to thank the Texas A&M University Graduate and Professional School to allow me to construct this LaTeX thesis template.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Dileep Kalathil (advisor), Dr. Srinivas Shakkottai of the Department of Electrical & Computer Engineering and Dr. Srikanth Saripalli of the Department of Mechanical Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

RL	Reinforcement Learning
MDP	Markov Decision Process
DR	AWS DeepRacer
TB3	Turtlebot3 Burger
PPO	Poximal Policy Optimization
SAC	Soft Actor Critic
WB	Wheel Base
AWS	Amazon Web Services
IMU	Inertial Measurement Unit
GPS	Global Positioning System
LOGO	Learning Online Guidance Offline
TRPO	Trust Region Policy Optimisation
GPU	Graphical Processing Unit
ROS	Robot Operating System
UDM	Unit Distance Model

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Autonomous navigation of mobile robots in indoor environments is an active area of research with the recent advancements in sensor technology & artificial intelligence. Field applications such as exploration, surveillance, search and rescue operations require robots to efficiently explore the space and strategically avoid obstacles. The challenges for this problem include uncertainties in sensor measurements, fusing data from different sensors, inaccurate maps due to dynamic and uncertain environments & physical limitations due to the dynamics of the robot. Model free deep reinforcement learning algorithms have been successful in various complex tasks requiring sequential decision making. This research work addresses some of the challenges involved in navigating mobile robots in indoor environments without a known map of the environment, using state of the art deep reinforcement learning algorithms and onboard sensors.

Current widely used global planners for autonomous navigation tasks provide a sequence of way-points by optimizing trajectory costs without taking the robot dynamics and physical limitations into consideration. The mobile robot further tracks these way-points to reach the destination. Control algorithms such as Proportional-Integral-Derivative (PID) control or Model Predictive Control (MPC) can be used for tracking these control points. However, there are some challenges involved with these algorithms. PID algorithms require frequent fine-tuning. MPC requires a dynamic model to predict the next states with an immediate action. This research work proposes an efficient, modular and scalable waypoint tracking algorithm using deep reinforcement learning that gives the feasible sequence of linear and angular velocities in order to smoothly track the waypoints. The reinforcement learning model is trained to learn the optimal sequence of actions for optimizing the trajectory errors and distances to obstacles.

1.1 Background

1.1.1 Machine Learning

Machine learning (ML) is a subset of the vast field of artificial intelligence. ML entails the development of algorithms that use statistical techniques to learn from data. It is broadly classified into 3 categories - supervised learning, unsupervised learning and reinforcement learning. Supervised learning methods consist of identifying the structure or pattern in labelled data. Some common supervised learning methods include linear regression, support vector machines or k-nearest neighbours. On the other hand, unsupervised learning algorithms deal with associating patterns in unlabelled data. Clustering using k-means is one of the most widely used unsupervised learning algorithm. Lastly, Reinforcement learning algorithms involve learning from experience to take the correct sequence of decisions in-order to maximize the notion of a cumulative reward.

With the advent of high-end compute technology, deep learning has gained more prominence due to its efficiency with large data-sets and its applications in the research and industry. Deep learning is largely based on the foundation of neural networks to learn more complex function approximations or abstractions than traditional ML algorithms. Due to its ease of use with high dimensional unstructured data it has wide applications in the areas of time-forecasting, autonomous driving systems, natural language processing and robotic process automation.

1.1.2 Reinforcement Learning

In contrast to supervised & unsupervised learning, reinforcement learning (RL) is a framework where a learner agent gains experience by repeatedly interacting with the environment and gets a reward signal as feedback from the environment. The objective of the learner is to maximize its cumulative reward over time.

The RL framework, as per Fig. 1.1 [1] consists of following entities - an $agent$, an $environment$ and the $reward$. The $agent$ interacts with the $environment$ by taking $actions$ and receives the observed $state$ and a feedback $reward$ from the $environment$. As the $agent$ acts in the $environment$, it can be in one of many **states** s ($s \epsilon S$), and choose to take an **action** a ($a \epsilon A$). The

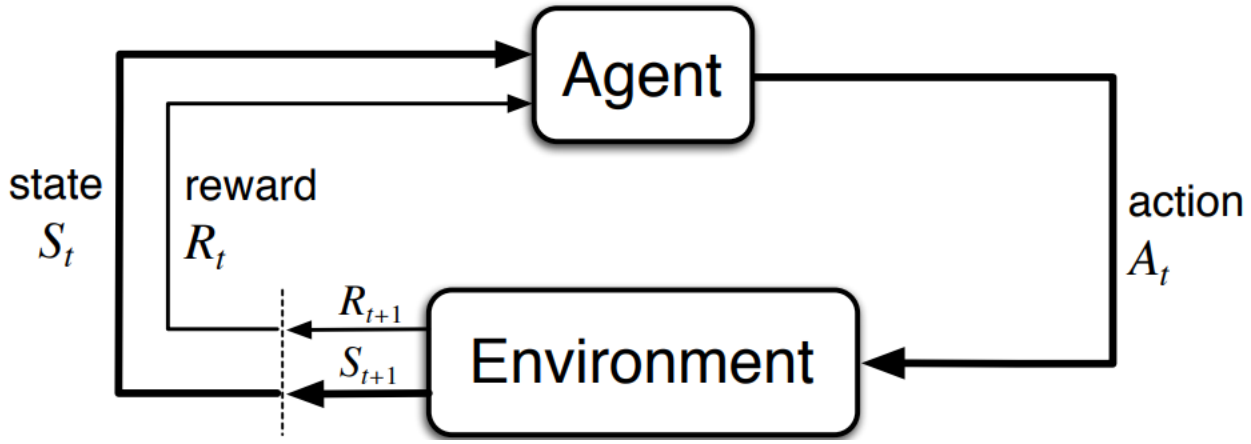


Figure 1.1: Agent-Environment Interaction in an MDP

environment gives a **reward** feedback depending on the action taken and the next observed state, based on the transition probability matrix $P(s'|s,a)$. The **policy** $\pi(s)$ is a behaviour function that defines the optimal action to take in a particular state s to maximize future rewards in an episode. It is a mapping between the states and the actions and can either be deterministic or stochastic. In case of deterministic policy, $\pi(s) = a$, and for stochastic policy, $\pi(a|s) = P_\pi[a|s]$. A training setup is called on-policy if it uses the samples from the target policy to train an RL algorithm. On the other hand, training an RL algorithm on the sample distribution by a behaviour policy other than the target policy is called an off-policy design. The state value function determines how rewarding a state or an action is by calculating a prediction of future reward. The total sum of discounted rewards from current time instant t in an infinite horizon or till the end of the episode is called as the return G_t. The state-value function $(V_\pi(s))$ is the expected return from current state s at time t,

$$V_\pi(s) = \mathbf{E}_\pi[G_t|S_t = s]$$

The action-value function $(Q_\pi(s,a))$ is given as -

$$Q_\pi(s,a) = \mathbf{E}_\pi[G_t|S_t = s, A_t = a]$$

The advantage function given by $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$ is considered a low variance Q-value function, with the state-value taken off as a baseline.

The result of reinforcement learning is the optimal policy learnt by the agent helping it actively adapt to the environment and maximize future rewards. Thus, the optimal value function produces maximum return -

$$V_*(s) = \max_\pi Q_\pi(s,a)$$

and optimal policy is given as -

$$\pi_*(s) = \arg\max_\pi Q_\pi(s,a)$$

1.1.2.1 Proximal Policy Optimisation Algorithm

Proximal Policy Optimisation Algorithm (PPO) is a popular policy gradient algorithm. Policy gradient methods of solving RL problems target modelling and optimizing the policy directly. The policy is modelled with a parametrized function $\pi_\theta(a|s)$. The value function is then expressed as a function of this parameter and various algorithms are applied to optimize θ for the maximum reward. In a generalized form, policy gradient approaches maximize the expected total reward by repeatedly estimating the policy gradient. Two main components of a policy gradient are the policy and the value function. Actor-critic methods consist of these two models where : **Critic** model updates the value function parameters w for $Q_w(a|s)$ or $V_w(s)$ and the **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$ in a direction suggested by the critic. The parameters w and θ have two separate learning rates for the updates.

Let the ratio between the older and newer policies be given by :

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$$

Policy gradient methods are challenging since they are sensitive to the stepsize, the training progress

is slow for a smaller step-size and it is overwhelmed by noisy signals with a larger step size. To avoid parameter updates that change the policy drastically resulting in instability in training, TRPO enforces a KL divergence on the policy update step. PPO simplifies it by using a clipped surrogate objective. It tries to compute an update that minimizes the objective function while ensuring deviation from previous policy is relatively small. The objective function for optimizing policy gradient, as per TRPO, is given by :

$$J^{TRPO}(\theta) = \mathbf{E}[r(\theta)A_\theta(s,a)]$$

The ratio $r(\theta)$ is then constrained to stay within a small interval precisely $[1-\epsilon, 1+\epsilon]$, ϵ being a hyperparameter.

$$J^{PPO}(\theta) = \mathbf{E}[\min(r(\theta)A_\theta(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)A_\theta(s,a))]$$

This algorithm was observed to have the following failure modes -

1. PPO gets unstable with continuous action spaces, when rewards vanish outside bounded support
2. PPO gets stuck at suboptimal actions for discrete action spaces with sparse high rewards
3. PPO is sensitive to initialisation when there are locally optimal actions close to its initialisation

Discretizing the action space, designing a good reward function and using KL regularization are some proposed solutions over above challenges.

1.1.3 Kinematic Modelling

It is crucial to understand the underlying physics of the robots in order to design appropriate control strategies. Considering the robot as a rigid body in motion, operating in a horizontal plane, the total dimensions of the robot are three. The degrees of freedom internal to the joints are ignored to have a simple kinematic model. Thus the pose of the robot at time instant t can be described in a

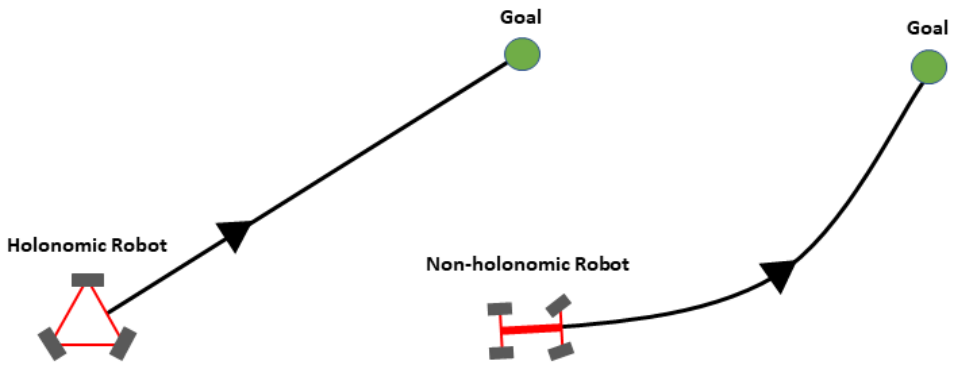


Figure 1.2: Holonomic vs Non-holonomic system path

global reference frame with three elements, x and y coordinate of the actual position of the robot in the global frame and the angular difference between the global and robot local reference frame given by θ.

$$\text{pose} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix}$$

Most widely used mobile robots require the wheels to point in the direction of motion and are not designed to slide sideways as shown in Fig. 1.2, called as the non-holonomic constraints.

1.1.3.1 Dubin's Car Model

This is one of the easiest vehicle model. Although it has three degrees of freedom, the action space is two-dimensional. Consider a vehicle configuration at position and orientation of (x_t, y_t, θ_t) at a time instant t, as shown in Fig. 1.3 with v_t as the linear velocity command and ϕ_t as the steering angle. The car moves in a circular motion, with radius determined by the wheel base (*WB* - distance between front and rear axle) and ϕ. The transition dynamics for this configuration are as follows -

$$x_{t+1} = x_t + v_t \cos(\theta_t)$$

$$y_{t+1} = y_t + v_t \sin(\theta_t)$$

$$\theta_{t+1} = \theta_t + \frac{v_t}{WB} \tan(\phi_t)$$

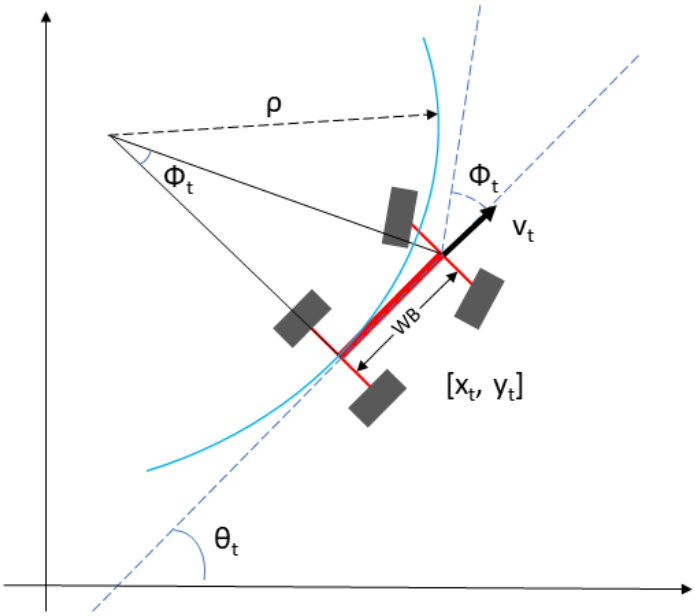


Figure 1.3: Dubin's Car Model Configuration

Here, the steering angle is constrained by the physical limitations between $[-\phi_{max}, \phi_{max}]$, resulting in a minimum turning radius of $\frac{WB}{\tan\phi_{max}}$. In this work, the AWS DeepRacer is the real world robot demonstrating a close-to Dubin's car model.

1.1.3.2 Differential Drive Model

Typical indoor mobile robots are designed differently then a Dubin's car. The configuration consists of two primary motored wheels, a third wheel (caster wheel) helps the robot balance and rolls passively in the direction of the robot's motion. Again, the action space is two dimensional compared to the three degrees of freedom of the robot. Consider a vehicle configuration at position and orientation of (x_t, y_t, θ_t) at a time instant t, as shown in Fig. 1.4 with v_t as the linear velocity command and ω_t as the angular velocity command. Due to the differential drive, the angular velocity is a resultant of the difference in velocities of the left (v_t^l) and right (v_t^r) linear velocities. Thus, with (v_t^r = v_t^l), the resultant velocity of the robot is (v_t^r + v_t^l)/2 in the forward direction of the wheels. In case ($v_t^l \neq v_t^r \neq 0$), the resulting angular velocity is (v_t^r - v_t^l)/WB. The transition dynamics for this

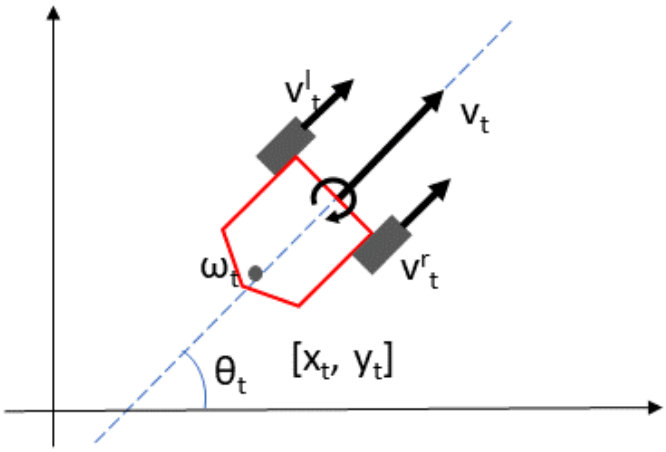


Figure 1.4: Differential Drive Model Configuration

configuration are as follows -

$$x_{t+1} = x_t + v_t \cos(\theta_t)$$

$$y_{t+1} = y_t + v_t \sin(\theta_t)$$

$$\theta_{t+1} = \omega_t$$

In this work, Turtlebot3 Burger is a real world robot demonstrating a close-to Differential drive model.

1.1.4 Mapping & Localization

Simultaneous localization and mapping (SLAM) refers to the problem of generating a map of the unknown environment surrounding the robot along with knowing the location of the robot within that map. It is a challenging research area for indoor mobile robots where there is no sensor accurate enough and GPS is not readily available. There is no direct way to measure a robot's precise position instantaneously. With a known up-to-date map and self-localization, a robot has the necessary information to navigate its surroundings. Foundational approaches to SLAM include

formulation of two probabilistic estimation problems solved by the application of extended Kalman Filter or Rao-Blackwellized particle filters. Most widely used sensors in this process are IMUs, wheel encoders, Lidar, RGB/stereo vision cameras or a sensor fusion of any of these sensors. SLAM algorithms face practical challenges in dynamic environments combined with lower accuracy or uncertainties in sensors.

1.1.5 Path-Planning

In order to reach the goal, a robot typically uses a global and a local planner. The global planner uses the static map to generate a path from the start to goal point. The local planner further use the global plan, in addition to the robot kinematics, sensor estimation to calculate the a local path-planning. The objective of path-planning is to obtain a near-optimal collision-free path from starting to goal position within some constraints such as time, proximity to obstacles, robot kinematics or path-length using a map of the environment. A widely used global path-planning algorithm is A^* designed for weighted graph based situations. The weighted graph is similar to an occupancy map, where weights are the cost of traversing that space. A^* seeks to find the minimum cost path between start and goal node of the graph, where cost is the sum of the transition between the nodes and a heuristic that estimates the cost of cheapest path to the goal node. Since A^* does not take into account the non-holonomic constraints, the calculated path needs to be smoothened to satisfy these constraints. A most commonly used local planner for collision avoidance is the Dynamic Window Approach (DWA). Besides a distance to the goal point the objective function also takes into account the distances to the obstacles and the robot kinematics. Thus, DWA produces linear and angular velocity commands optimal as per the local surroundings of the robot, called the dynamic window. The search space is restricted to this dynamic window of possible next states of the robot given its current velocities. The DWA planner performs a simulation and evaluates the different velocity pair commands using the cost function and chooses the one with the lowest score. The algorithm repeats these steps in an iterative procedure.

1.1.6 Control

Autonomous navigation requires a control system in-place for path following consisting of longitudinal and lateral control. The longitudinal control regulates the cruise velocity while the lateral control steers the vehicle's wheels for accurate path tracking. Pure-pursuit is the most common lateral control strategy. It is a geometric path tracking controller that tracks a reference path using the vehicle kinematics and reference path. It uses a look-ahead point a fixed distance ahead on the reference path. Considering a kinematic bicycle model, the instantaneous radius of curvature R associated with the correct steering is given by -

$$R = \frac{WB}{\tan \delta}$$

where, δ is the desired steering angle. Thus, δ can be shown as -

$$\delta = \arctan(\frac{2WB \sin \alpha}{l_d})$$

where, α is the angle between the vehicle heading and the look-ahead reference line, and l_d is the distance between the vehicle and the target. Pure-pursuit controller works proportionately against the cross-track error. The other path-tracking errors include the heading error and the along-track error as shown in Fig. 3.2.

1.2 Contribution

This work focuses on implementing reinforcement learning to the problem of autonomous navigation of mobile robots. The primary contributions are as follows -

1. Developed a modular & scalable framework for autonomous waypoint tracking by mobile robots

2. Achieves a mapless waypoint tracking scheme

3. Analyses the application to a Dubin's car and a Differential Drive robotic model

4. Demonstrates the evaluation of a custom algorithm using the proposed framework

The framework presented in this work reduces the exploration space and hence lowers the training time for the reinforcement learning model. The trained policy maps the target coordinate in the robot frame to produce the velocity commands resulting in efficient trajectory toward the target waypoint. The designed model is trained in a low fidelity Python simulator as well as Gazebo before transferring to the real-world robot.

1.3 Thesis Outline

Chapter 1 Introduces the area of reinforcement learning and autonomous navigation for robots, gives an overview of this thesis and the major contributions achieved with this research work and the motivation to do the same.

Chapter 2 Reviews the relevant research work done till date in the areas of robot path-planning and obstacle avoidance in cluttered spaces using reinforcement learning.

Chapter 3 Outlines the design of the proposed reinforcement learning strategy for autonomous navigation of mobile robots

Chapter 4 Describes the experimental setup of the research. It includes the hardware as well as the software description of the various sensors, controllers, robots, compute technology and software packages used for this work.

Chapter 5 Presents a detailed analysis and results from the various experiments and design architectures evaluated during the course of this research study.

Chapter 6 Demonstrates one application of the designed framework for autonomous navigation for evaluating a custom Deep RL algorithm.

Chapter 7 Derives conclusions from the results and analysis of the experiments conducted as a part of this research and suggests future work in this area.

2. LITERATURE REVIEW

Traditional approaches to autonomous navigation include recent optimized path-planning methods such as A* search, Rapidly-exploring Random Trees(RRT), D* and D* Lite algorithms. Cheng et al. [2] demonstrate the use of Lidar for Mapping an unknown environment with SLAM techniques (Hector SLAM & GMapping) and implement global path-planning (A*) and a Dynamic Window Approach to navigate the mobile robot in an indoor environment. However, these methods are not suited for dynamic environments where fast response to change is essential. The evolutionary and hybrid approaches adapt neural networks, genetic algorithm, fuzzy systems and reinforced learning techniques. Based on a comparison between the two approach styles by Khaksar et al. [3], learning based approaches seem to have a slight advantage in handling dynamic environments.

Since the inception of Double Q-network (DDQN) by DeepMind [4] in 2016, the use of RL has gained momentum for autonomous navigation . Lei et al. [5] present a novel path-planning algorithm based on DDQN. They use CNNs to solve the generalization problem by effectively extracting the information from Lidar data and reducing network parameters. The work also found that the use of Lidar information over camera frames allowed for better generalization performance. Similar work done by Surmann et al. [6] successfully implements deep RL algorithms on the TurtleBot with fused data from the 3D RGB-D camera and the lidar scanner. They use a custom-built simulation environment in C++ to have a 1000 times faster, memory efficient and several instances in parallel compared to traditional simulators like Unity, Gazebo. The parallel learning is achieved by a Asynchronous Advantage A2C algorithm. Tai et al. [7] presented a learning-based mapless motion planner by taking the sparse 10-dimensional range findings and the target position with respect to the mobile robot coordinate frame as input and the continuous steering commands as output.

Besides path-planning, another challenging problem in autonomous navigation of mobile robots is avoiding stationary or moving objects in its path. Modern approaches implement a sensor fusion of data received from on-board Lidar sensors and stereo cameras. Cimrus et al. [8] proposes a

Convolutional Deep Deterministic Policy Gradient (CDDPG), network of depth-wise separable convolutional layers, to tackle depth images efficiently for a goal-oriented collision/obstacle avoidance by using an actor-critic network and finally transfer it to a real environment for map-less navigation. With a combination of stack of depth images and estimated position from the goal, the actor network produces angular and linear velocity as output and the critic network generates an action based on the actor and estimates Q-value of current state. Two of the most common methods for dynamic programming in RL are SARSA (on-policy learning) and Q-learning (off-policy learning). Manuelli et al. [9] demonstrates that both these methods work evidently similar in the discretized domain. Many of the modern approaches utilize Q-learning based deep neural network to deal with discrete-time decision making.

For the case of dynamic obstacles, [10] explores Q-Learning by perceiving 3 measurements with respect to the obstacle: velocity of obstacle, distance to obstacle, and direction with respect to obstacle. They then propose a reward framework which positively rewards moving away from the static/dynamic obstacle, and moving towards the target. Another work [11] approaches dynamic obstacle avoidance by having two planners, a higher level ("long term") planner which assumes no dynamic obstacles, and a lower level ("short term") local planner for dealing with nearby dynamic obstacles. The former planner gives the agent a general idea of what direction to move in, while the latter deals with providing more specifics on navigating the immediate locale.

The Sim-To-Real gap is commonly observed to degrade the performance of the trained RL policies when transferred onto the real robots. One commonly used technique to close Sim-To-Real transfer gap is domain randomization. In domain randomization, the simulation parameters are perturbed during training, and have been used for successful sim2real transfer for various robotic tasks. Methods include adding noise in dynamics [12] and imagery [13], learning model ensembles [14], adding adversarial noise [15], and assessing simulation bias [16]. Domain adaptation [17] has also been used for sim2real, particularly to address the visual reality gap.

3. DESIGN

This chapter outlines the design of the proposed waypoint tracking method using reinforcement learning and delves deeper into the design choices made and the motivation behind them. A high level overview of the proposed method is presented in Fig. 3.1. The proposed method is a modular & scalable approach to solve the problem of tracking a series of waypoints as provided by a global planner.

3.1 State Space

At every time step the RL agent receives the observed state from the environment and decides the next optimal action to maximize its future rewards. In order to make it a comprehensive state representation for the simple task of navigating from an initial fixed point to a randomly assigned goal for the episode, the state is a relative coordinate of the desired goal coordinates in the robot's frame of reference.

$$state = [x_t - x_o, y_t - y_o, \theta_t - \theta_o] \tag{3.1}$$

In above equation, (x_t, y_t) are the absolute coordinates of the vehicle at time instance t, (x_o, y_o) are the absolute target coordinates of the waypoint, θ_t is the absolute yaw of the vehicle at time instance t as calculated from the quaternion angles and θ_o is the absolute target heading towards the closest waypoint. Thus, the state is parametrized by the absolute relative distance and relative orientation from its current position to the closest waypoint giving us a generalized unit-distance model. In case of the obstacle avoidance task, the state vector is appended with lidar data.

$$state = [x_t - x_o, y_t - y_o, \theta_t - \theta_o, [l_d(k)]] \tag{3.2}$$

In above equation, l_d is a 36-dimensional sectorized vector of the raw lidar scan. A raw lidar scan contains dense information about the proximity of obstacles in two dimensions. Since the tasks in this research work are carried out in two-dimension settings, a lidar scan conveys same information

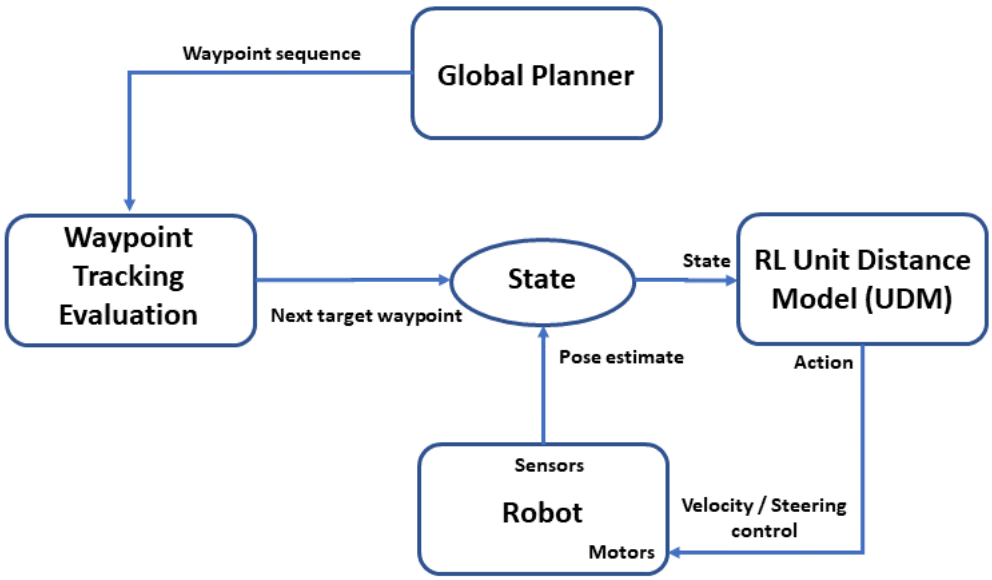


Figure 3.1: System Overview

density as a depth camera but in a lower dimension state space. Although depth images are also common for obstacle avoidance tasks, the higher dimension state vector can slow down training time and lead to sample inefficiency. Similarly, RGB images do not directly provide distances to obstacles and are variant to lighting conditions.

In this way, the relative distance and orientation to the desired goal along with the lidar data does not have any dependence on the mapping or localization. In common words, a robot navigating in a mapless environment only needs an estimate of its relative heading and distance to the desired goal for controlling the angular and linear velocities of the robot. This is directly motivated by the natural navigation sense of humans. The state vector provides all the critical information that the RL agent needs to decide what actions to take to reach the goal.

3.2 Action Space

The action space depends on the robot (or kinematic model) used and the motor's physical limits. This research work uses two widely known robots in the robotics research community - AWS DeepRacer and Turtlebot3 Burger. The AWS DeepRacer resembles a Dubin's car model wheres the Burger bot is close-to a differential drive model. Thus, the AWS DeepRacer uses an action pair

of linear velocity and steering angle $action = [v, \phi]$ commands whereas the Burger robot uses a linear and angular velocity pair $action = [v, \omega]$, ϕ is the steering angle command to the Ackermann driving mechanism, ω is the angular velocity.

Because of the slower speeds of indoor mobile robots, a discrete action space is preferred, allowing the action space to be small as possible while still allowing sufficient resolution of control. The linear velocities that the agent can decide to take ranges from $\{0.0, 0.1, 0.2\}m/s$ and for angular velocity ranges from $\{-0.5, -0.25, 0.0, 0.25, 0.5\}rad/s$. Keeping the action space of linear and angular velocity ensures seamless motion between time-steps leading to a 15-dimension action space.

3.3 Reward Function

The objective of the RL agent is the driving force in designing a proper reward function. For a task of reaching a desired goal point, the environment should reward the agent for motion directed towards the goal and penalize it for any motion directing away from the goal or towards the obstacles. Thus, the following reward function was carefully designed.

$$reward = \begin{cases} +10, & \text{if } |x_t - x_0| \leq 0.2 \text{ and } |y_t - y_0| \leq 0.2 \\ -1, & \text{if it crosses the boundary grid} \\ -100 & \text{if it collides} \\ -(\text{c.t.e}^2 + a.t.e + h.e. & \text{otherwise} \end{cases}$$

where $c.t.e$ is the cross track error given by $c.t.e = l_d \sin \alpha$, l_d is the distance to the target, α is the difference between the heading to the target and the vehicle yaw at time t, $a.t.e$ is the along-track error $a.t.e = |x_t - x_0| + |y_t - y_0|$, $h.e.$ is teh heading error $h.e. = \theta_t - \theta_o)))$, $[x_t, y_t]$ are the current coordinates of the robot while $[x_0, y_0]$ are the target coordinates of the desired goal point. Refer Fig. 3.3. A collision is detected when the least distance to the obstacle is lower than threshold.

The reward function consists of the error terms derived from conventional path tracking algorithm for vehicle control, as per Figure 3.2, mainly the cross track error, along track error and the heading

error. The $reward$ strictly penalizes the cross track error with a quadratic term which demonstrates tighter tracking trajectory in reaching the target point than a linear term.

Figure 3.2: The three types of path following errors **(a)** cross track error (δn) **(b)** along track error (δs) **(c)** heading error $(\delta\theta)$

The above reward function is a result of careful fine-tuning in a low-fidelity simulator to inspect the combination of coefficients resulting in an optimal behaviour of trained policy.

3.4 RL Algorithm

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

State-of-the-art SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way. The entropy regularization coefficient α explicitly controls the explore-exploit tradeoff, with higher α corresponding to more exploration, and lower α corresponding to more exploitation. The right coefficient (the one which leads to the stablest / highest-reward learning) may vary from environment to environment, and could require careful tuning.

PPO works in both discrete and continuous action space, while SAC works only in continuous action space. PPO being an on-policy algorithm tends to be more stable but data hungry whereas off-policy algorithms like SAC tend to be more data-efficient. PPO is also very easy to implement and shows faster convergence and is used for all the training experiments in this work.

The code-base used for this work is StableBaselines3.

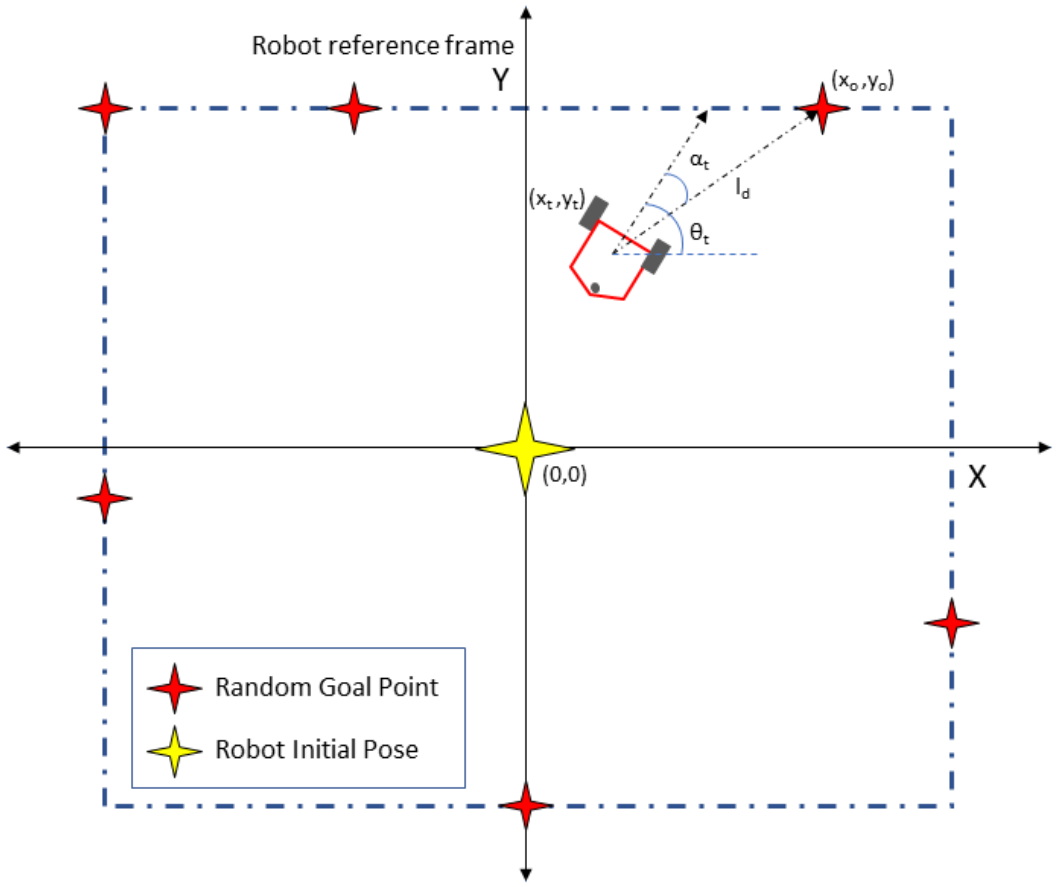


Figure 3.3: Unit Distance Model Training Scheme

3.5 Training Scheme

The RL training setup shown in Fig. 3.3 consists of teaching the robot to reach a random target goal point roughly a unit-distance away from its reset position and orientation. For every training episode, the robot is reset to the origin and a goal point is randomly chosen from a unit square grid boundary. At every time-step in a training episode, the reward is calculated as given above. The episode ends if the robot reaches the goal point for that episode, or if it exceeds the training boundary, or if it collides with an obstacle (in case of the obstacle avoidance task in Gazebo simulation). The training is complete once the reward converges with 100% episode completion.

3.6 Evaluation for Waypoint Tracking

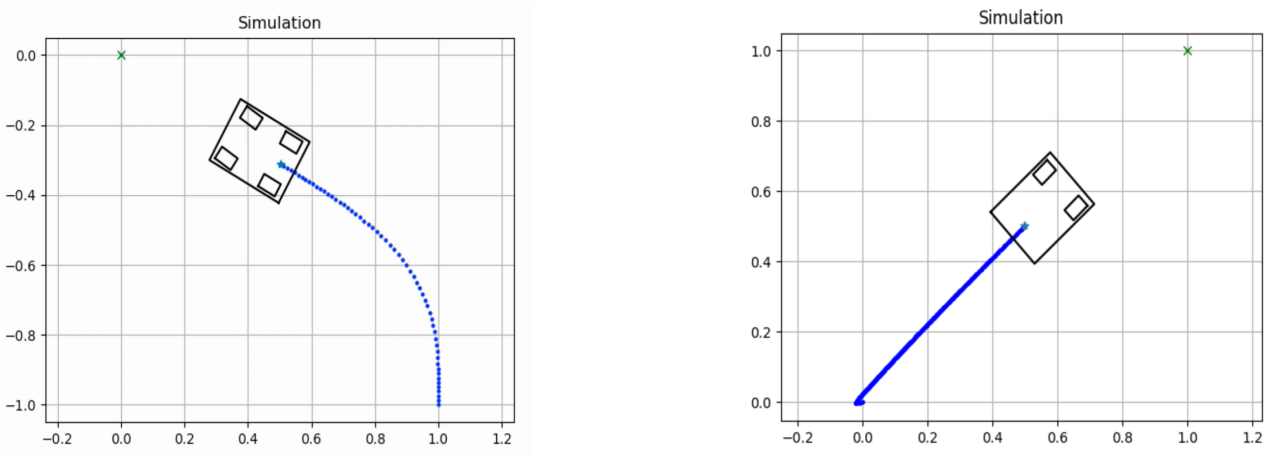
The trained unit-distance RL policy (either from the low fidelity simulator or from Gazebo) is further used for evaluating with a given series of waypoints provided by the global planner, shown in Fig. **??**. The global planner uses a high level map of the robot's surroundings (walls and corridors) for planning a trajectory using waypoints a unit-meter distance away. The robot checks for the two nearest waypoints on the planned trajectory by calculating the distances to the waypoints at every timestep and sets the farther one as the temporary goal-point for its next episode. It uses the trained RL optimal policy to navigate to this goal-point and resets the episode once the goal is reached. Due to the relative distance and orientation in the state vector of the trained policy, it can be generalized to navigate from a point in the grid-space to any point a unit-meter distance away from it. With this fast and scalable training and evaluation strategy, a good RL model with effective results is achieved in lesser time without the use of heavy compute clusters such as multiple GPUs for parallelized computations or simulation instances.

4. IMPLEMENTATION

This chapter outlines the details of implementing the design presented in the previous chapter, along with how the agent was trained and evaluated. Due to larger training time, safety concerns, and cost effectiveness, it is impractical to fully train the agent on a real robot. Additionally, it requires constant human supervision to reset the robot after each episode. Thus, training in a simulator is preferred. The major benefits of training in a simulator include being able to speed up simulation multiple factors of real-time, and being able to script training such that training is consistent and can automatically reset when the agent reaches terminal states. For these reasons, the agent proposed in this thesis is trained entirely in simulation and only evaluated in the real-world.

4.1 Kinematics Simulator

We designed a low-fidelity kinematics simulator using the Open AI Gym framework. It provided faster training to evaluate different design choices of algorithm, state space, action space and reward function. A custom environment (a Python Class) is setup, as in Fig. 4.1 using the OpenAI Gym API for interfacing the RL algorithm with the simulator kinematics equations as follows -



(a) Dubin's Model

(b) Differential Drive Model

Figure 4.1: Low-fidelity Kinematics Simulator

For Dubin's Car Model,

$$x_{t+1} = x_t + v_t \cos(\theta_t)\Delta, \;\; y_{t+1} = y_t + v_t \sin(\theta_t)\Delta, \;\; \theta_{t+1} = \theta_t + \frac{v_t}{WB} \tan(\phi_t)\Delta$$

For Differential Drive Model,

$$x_{t+1} = x_t + v_t \cos(\theta_t)\Delta, \;\; y_{t+1} = y_t + v_t \sin(\theta_t)\Delta, \;\; \theta_{t+1} = \theta_t + \omega_t\Delta$$

where x_t, y_t are the current pose coordinates of the bot, v_t is the linear velocity, ω_t is the angular velocity, and θ_t is its yaw calculated from the quaternion angles, all calculated at time instant t. Δ is the time discretization factor. We define the state s_t to be the normalized relative position w.r.t. the waypoint, i.e., $s_t = ((x_t - x_0)/G, (y_t - y_0)/G, \theta_t - \theta_0)$, where x_0, y_0 are the target coordinates of the waypoint, and θ_t^0 is the target heading to the waypoint at time t. We define the action a_t to be the pair of linear and angular velocities, i.e., $a_t = [v_t, \omega_t]$ for the differential drive model and a pair of linear velocity and steering angle i.e. $a_t = [v_t, \phi_t]$ for the Dubin's car model. We discretize the action space into 15 actions. Thus the environment class description for waypoint-tracking is described in Table. 4.1.

This simulator demonstrates faster execution and thus lower training times for the RL algorithm. Without any complex graphics rendering, this simulator trains an RL algorithm using the Nvidia GeForce RTX 2080 Ti GPU and converges to an optimal policy within a few minutes. Although the dynamics equations and constants can be configured to match the real-world robot behavior, this simulator does not account for any of the real-world uncertainties, latencies and sensor inaccuracies. Also, it is difficult to model complex sensor models such as lidars or vision cameras. However, it can be used as a quick prototype of the reward structure and RL algorithm design for a waypoint navigation task without any obstacles in the surrounding.

Python Simulation : Environment Class Description (Without Obstacle Avoidance)	
Entity	**Description**
State Space	Box(low = [-1.0,-1.0,-4.0], high = [1.0,1.0,4.0])
Action Space $[v, \omega]$	Discrete(15) - 0: [0., -2.5], 1: [0., -1.25], 2: [0., 0.], 3: [0., 1.25], 4: [0., 2.5], 5: [1.0, -2.5], 6: [1., -1.25], 7: [1., 0.], 8: [1., 1.25], 9: [1., 2.5], 10: [2., -2.5], 11: [2., -1.25], 12: [2., 0.], 13: [2., 1.25], 14: [2., 2.5]
Reward	$-cte^2 - \lvert x_1 - x_o \rvert - \lvert y_1 - y_o \rvert - \lvert \theta_t - \theta_o \rvert$
Terminal Condition	Goal Reached, Training Boundary Exceeded Maximum Episode Length Exceeded

Table 4.1: Python Simulation - Environment Class Description

4.2 Gazebo Simulator with ROS

Gazebo is a physics engine simulator with rendering capabilities and can be used for realistic simulation of the environment. It is configurable and can be used to model robots with multiple joint constraints, actuator physics, gravity and frictional forces and a wide range of sensors in indoor as well as outdoor settings. Instead of real-world data for RL training, Gazebo facilitates close-to-real-world data collection from the robot & sensor models. Since it runs in real-time, it takes millions of simulation frames for an RL agent to learn simple tasks such as navigating from a fixed point to a goal point a unit-distance away. With respect to this project, the task of navigating from a starting point to a target point 8m away took 12 to 14 hrs training. Gazebo can also speed up simulation by increasing step size. This can however lead to loss of precision. For this project, we ran the training scheme on Gazebo in almost real time with a simulation step-size of $\Delta T = 0.001$.

The Gazebo simulation setup consists of the differential drive robot (Turtlebot3 Burger) model spawned in either an empty space or a custom space with static obstacles at predefined locations (depending on the evaluation task). A default coordinate grid gets setup with respect to the base-link

Gazebo Simulation - Environment Class Description (With Obstacle Avoidance)	
Entity	**Description**
State Space	Box(low = [-1.0,-1.0,-4.0], high = [1.0,1.0,4.0])
Action Space $[v, \omega]$	Discrete(15) - 0: [0., -2.5], 1: [0., -1.25], 2: [0., 0.], 3: [0., 1.25], 4: [0., 2.5], 5: [1.0, -2.5], 6: [1., -1.25], 7: [1., 0.], 8: [1., 1.25], 9: [1., 2.5], 10: [2., -2.5], 11: [2., -1.25], 12: [2., 0.], 13: [2., 1.25], 14: [2., 2.5]
Reward	+100 (goal reached), -100 (collision detected), -10 (out of boundary)
Terminal Condition	Goal Reached, Training Boundary Exceeded Episode Length Exceeded, Collision Detected

Table 4.2: Gazebo Simulation - Environment Class Description

of the robot model which is used by the odometer sensor model. An environment class, similar to the python simulator, for obstacle avoidance is described in Table. 4.2.A ROS framework instantiated by a custom-built OpenAI Gym environment is used to provide interface between the proposed RL algorithm and the simulation model via ROS topics.

The ROS topics used are as follows -

$/odom$ (for x_t, y_t, v_t) - Contains the Turtlebot3 odometry information based on the encoder

$/cmd\_vel$ (for ω_t, θ_t) - Controls the translational and rotational speed of the robot unit in m/s, rad/s

$/scan$ (for obstacle avoidance) - Reads the scan values of the LiDAR mounted on the Turtlebot3

The topics mentioned were used to capture the state update information of the robot asynchronously in a callback driven mechanism. The $/cmd\_vel$ topic was used as the topic to output the relevant action needed for the robot simulator. The captured state information is used in the step() function of the OpenAI gym interface to take a relevant action. The reset() function of the OpenAI gym interface was used to reset the robot to it's initial configuration.

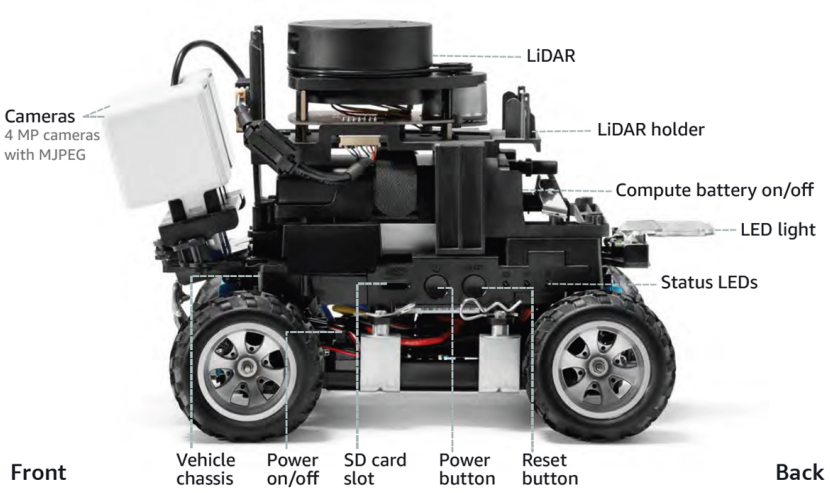


Figure 4.2: AWS DeepRacer

4.3 AWS DeepRacer Evo

AWS DeepRacer, Fig. 4.2 is a robotic vehicle that is a 1/18th scale model car developed for development of autonomous racing algorithms using deep reinforcement learning.It has the DeepLens camera mounted in the front either as monocular vision or two for stereoscopic vision. Additionally, it has an on-board Intel Atom processor compute module. The compute module runs inference in order to drive itself along a racing track. Due to the Ackermann steering mechanism it has non-holonomic constraints in motion and closely resembles the Dubin's model of kinematics. The maximum linear velocity of the AWS DR is 5m/s, with higher velocities for the purpose of racing. Thus, the action space of the reinforcement learning agent is targeted at maximum of 1m/s such that it has controlled motion for navigational purpose. The AWS-DR has a Gazebo simulator model that accurately resembles the kinematics of the real vehicle.

4.4 Turtlebot3 Burger

Turtlebot, Fig. 4.3 is robot commonly used in robotics research. It features a two-wheeled differential drive train allowing the robot to turn about its center point, which is very useful for changing directions without needing to move forward or backward. It is supported by casters to prevent the robot from tipping due to having only two wheels. The maximum linear velocity of the

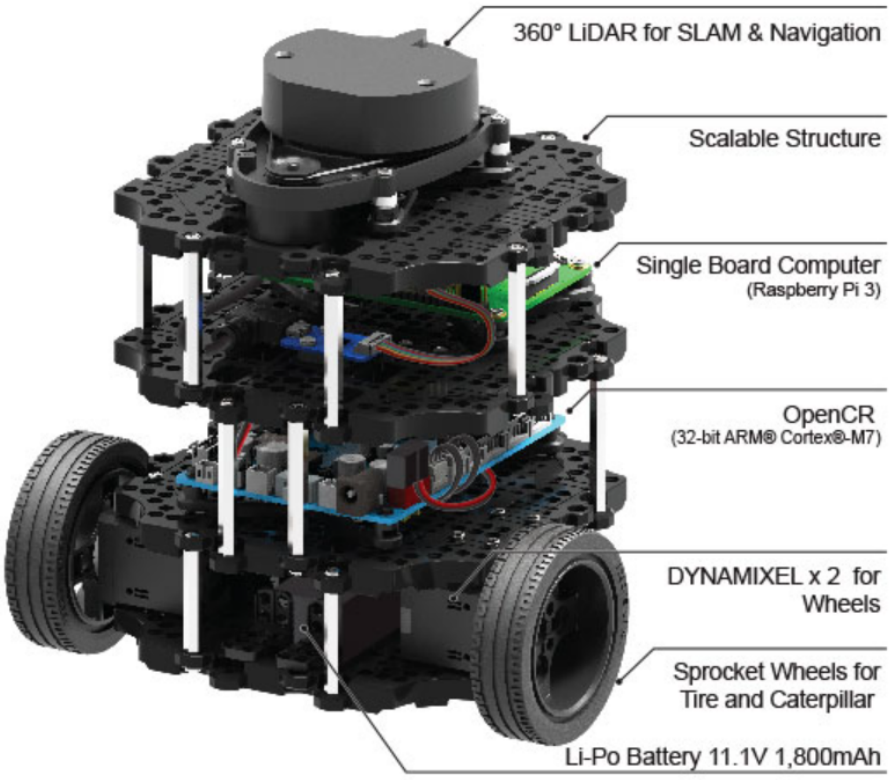


Figure 4.3: Turtlebot3 Burger

Turtlebot is 0.22m/s, so the action space of the reinforcement learning agent is designed such that the maximum velocity the agent can decide to go is 0.2m/s , safely within the Turtlebot's ability.

Turtlebot is open source and a model is publicly available for the Gazebo simulator. This model behaves in simulation just as a Turtlebot in the real world would, with accurate physics and dynamics simulation.

4.5 Training Environment

The RL models are trained in the pure kinematics simulator as well as the Gazebo simulator as per the design framework mentioned in Chapter 3. The low-fidelity simulator is an empty world space without any walls/ corridors/ obstacles. It is designed purely to prototype the design choices and test the proposed method in a simple environment. The model trained here solves the navigation task from a point to any goal point a unit distance away. The environment resembles a plain grid space in the robot's coordinate frame of reference.

Gazebo, being a more complex physics engine can include some advanced environment design and addition of walls/ corridors/ obstacles. The model is trained to navigate from a point to any goal point a unit distance away in the presence of obstacles. The PP algorithm from the Stable Baselines3 codebase uses PyTorch with a CUDA support that utilizes the host Nvidia GPU RTX 2080 Ti to accelerate the training process. For the task of waypoint tracking in an empty world, the Gazebo training environment consists of the TB3 Burger robot model spawned in an empty grid space. In order to limit exploration during the RL training, the Gym environment setup confines the robot within boundaries and ends the episode in case the robot exceeds the boundary grid. For the obstacle avoidance task, an obstacle is placed at a fixed location throughout the training (as in Fig. 4.4) and the desired goal point is randomized to ensure a rich exploration of states.

4.6 Evaluation Environment

The trained models from simulation are evaluated with the real robots in an experimental lab setup. The Unit Distance Model trained for navigating to goal locations is used to track down way-points as planned by a global planner. The experimental setup consists of a host PC (an Intel

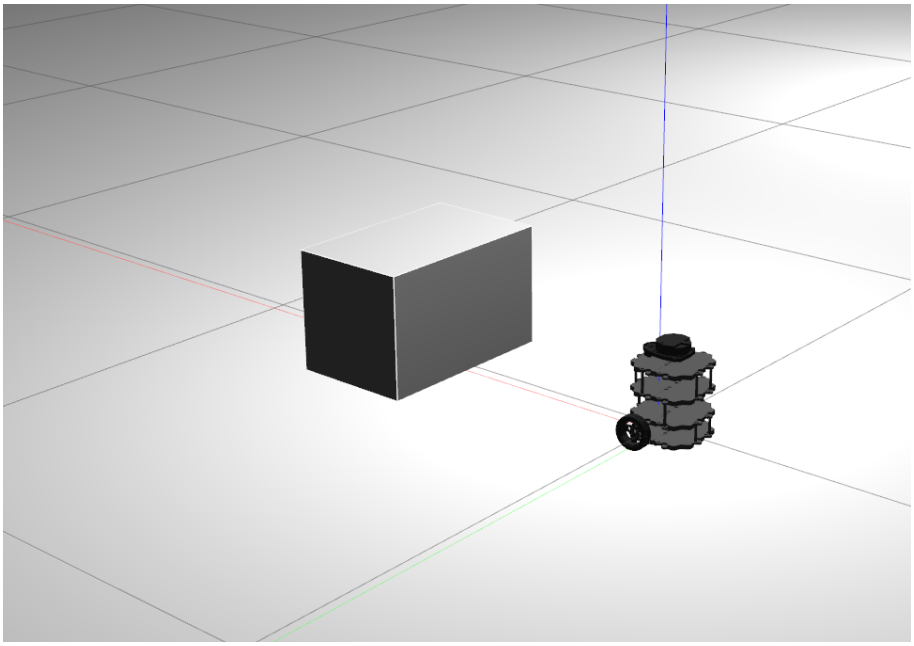


Figure 4.4: Gazebo Simulator Training Environment

NUC) running the ROS Master core. The turtlebot consists of an onboard Raspberry Pi to run the inference. The ROS aboard the Raspberry Pi subscribes to the same roscore running on the host PC. The Raspberry Pi runs the nodes for communicating with various sensors and actuators on the turtlebot and publishes the data onto topics. An evaluation script running on the host PC subscribes to these topics via ROS messages. The optimal policy then generates the best sequence of actions to take based on the observation state vector and publishes that onto the respective ROS topics. The Raspberry Pi then passes these ROS messages to the motor drives.

5.1 AWS DeepRacer

5.1.1 Pose Estimation Challenges

Successful implementation of the proposed methodology highly depends on the pose estimation performance by the robot sensors. This challenge was addressed for the real-world robots. The AWS Deepracer has a fleet of sensors such as the RP Lidar, monocular & stereo cameras. However, neither of these sensors can provide absolute pose estimation data. We have experimented with using an external IMU sensor, wheel encoders as well as the on-board Lidar sensor.

The Turtlebot-3 Burger on the other hand has wheel encoders attached on the Dynamixel servo motors. The Dynamixel servo motors are ultra-precise electric motors equipped with the magnetic encoders for precise pose estimation of the robot based on wheel rotations. Since the experiments in this work have been strictly on hard surfaces, the pose estimate has been precise. The wheel encoders can induce incorrect readings on slippery surfaces. In that case, a possible fusion of the wheel encoder estimate with an additional sensor such as the Lidar or visual camera can provide a correction.

Using IMU : We have integrated a 9DoF Razor IMU (as in Fig. 5.1) which houses an ITG 3200 gyro, ADXL345 accelerometer, & HMC5883L magnetometer; to make for the pose estimation required for our experiments. Prior experimentation, the IMU was calibrated by exposing it to different static poses and offsetting the measurements. Further, we implemented a Kalman Filter to process the raw data measured from the sensor (Refer Fig.5.3) and the acceleration measurement was double integrated to calculate the pose, as shown in Fig.5.2. However, the pose estimate is observed to accumulate significant drift over time (2 - 3 degrees per minute), even in static conditions, and requires a fusion with another sensor was rectification. An external wheel-encoder attachment was considered, but due to physical constraints on the free motion of the robot's wheels, other approaches were considered.

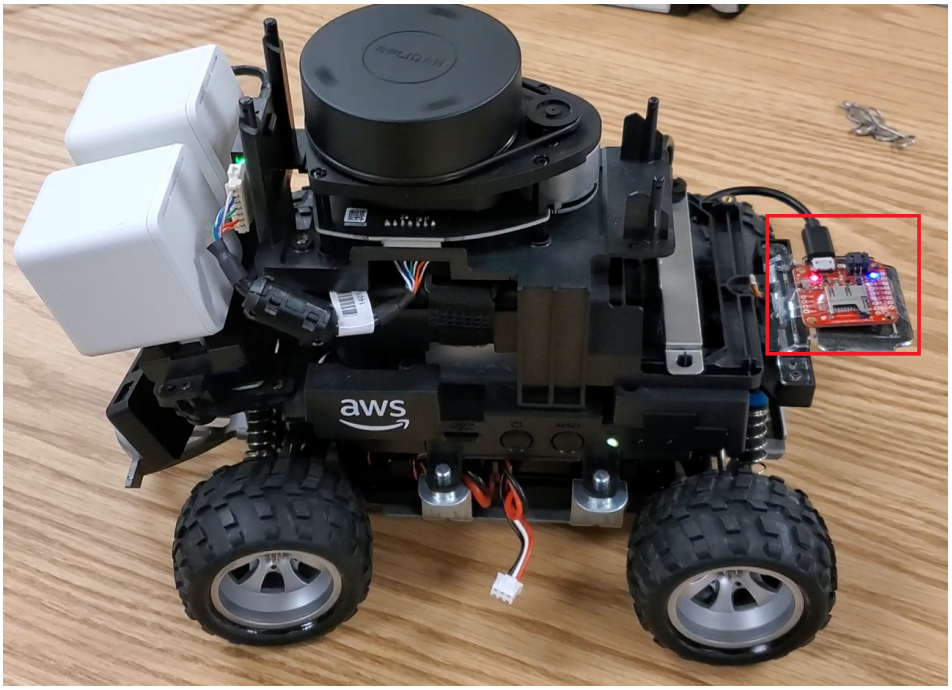
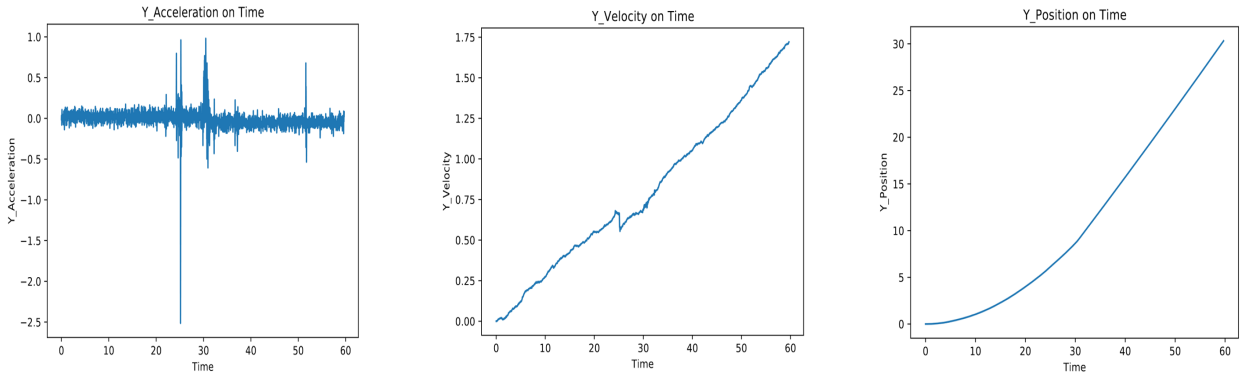


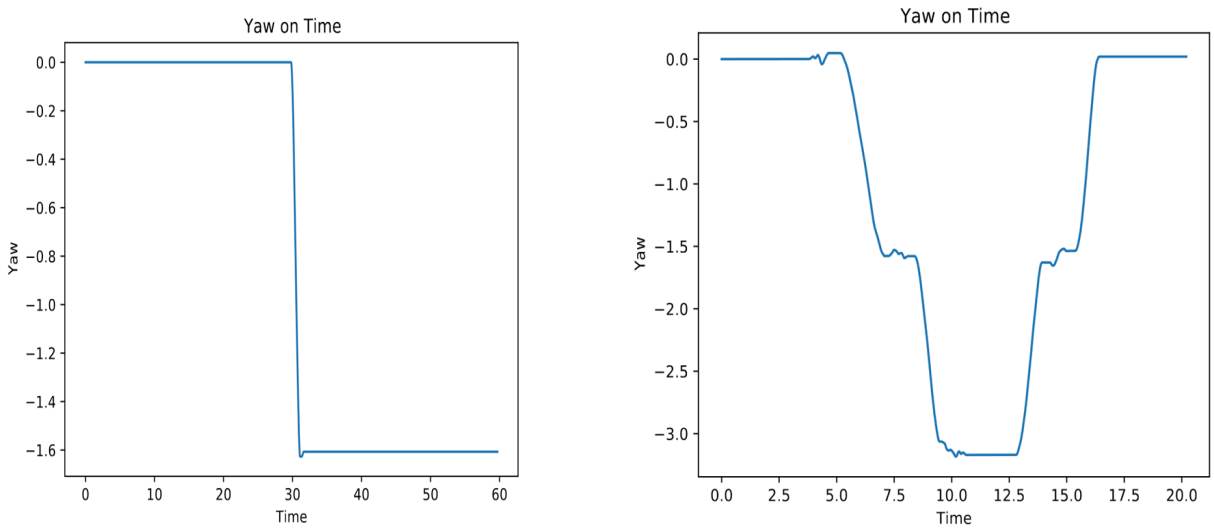
Figure 5.1: DR with IMU Integration



(a) (b) (c)

Figure 5.2: IMU Integration data for robot position - **(a)** is raw acceleration, **(b)** is velocity after integration, **(c)** is pose after double integration

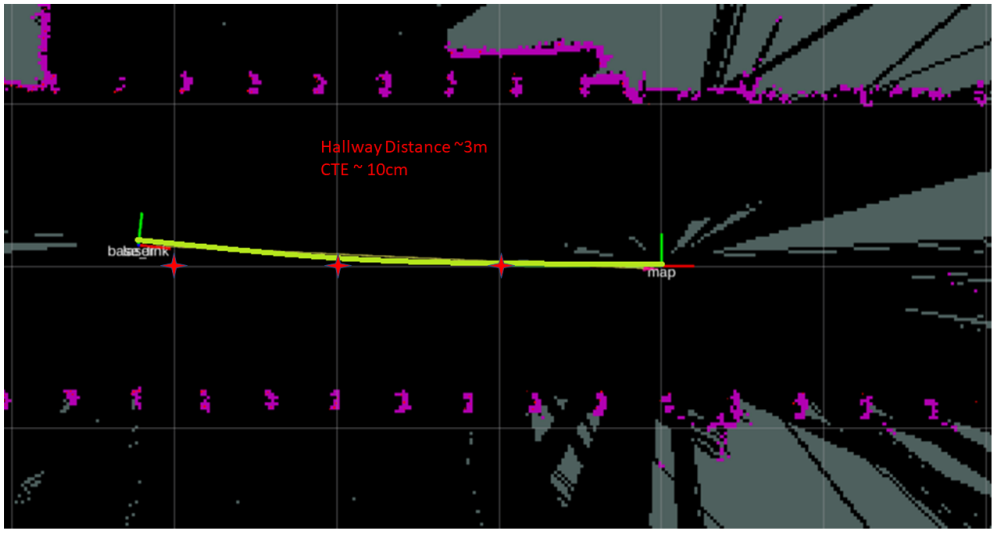
Using Laser Scan Matching : The DR has an onboard 360^o Lidar sensor which was used as a combined state estimation and localization sensor. We implemented the laser scan matching algorithm that incrementally matches consecutive lidar scans to estimate a pose of the lidar sensor without any other odometry sensors in place. This algorithm was tested for loop-closure and



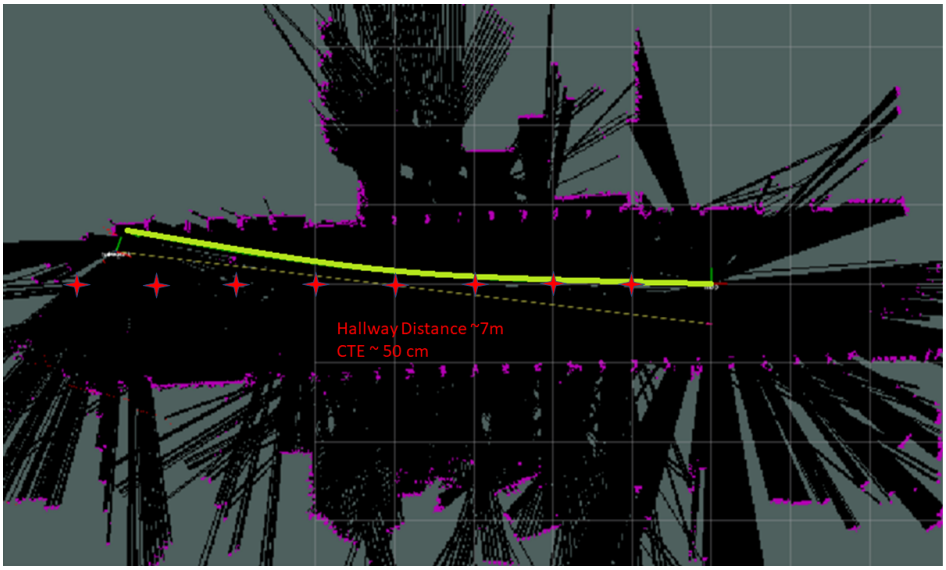
(a) (b)

Figure 5.3: IMU Integration data for robot orientation - **(a)** is yaw after a 90 degree turn & **(b)** is yaw after a 360 degree turn

localization performance showing upto $0.1m$ deviation at slower speeds (upto 1m/s) and $\geq 0.5m$ deviation for faster speeds. We further tested this pose estimation for a straight path along a hallway as in Fig. 5.4 for performance evaluation using PID waypoint tracking.



(a)



(b)

Figure 5.4: PID Waypoint Test with Lidar Scan Matcher Pose in a long hallway - **(a)** over 4m distance and **(b)** over 8m distance.

Since the AWS-DR follows Dubin's path, it was trained with Dubin's Model Kinematic model in the simulators before transferring to the real world.

5.1.2 Unit Distance Model

In Low Fidelity Simulator : Due to longer training times in Gazebo simulation for the DR physics model, the low fidelity simulator as discussed in section (4.1). was used. Following the training scheme from section (3.5), a unit-distance RL model for navigating from a fixed point in space (origin) to a randomly assigned goal point over a square boundary was trained using the PPO algorithm from Stable Baselines3 (PyTorch). The training reward curve in Fig. 5.5 converged within 800 episodes taking upto 2hrs on an Intel i7-8550U CPU machine without parallelization.

The Unit-Distance Model was evaluated with different target points in Fig. 5.6. The optimal policy learns to navigate with the shortest feasible path. The robot evaluates successfully for each of the runs with different target points. A continuous action space was used to profile the action space as in Fig. 5.7 for linear velocity and steering angle respectively.

In Gazebo : The above trained Unit Distance Model is evaluated in Gazebo for the same target points. Note that the inbuilt \odom topic from Gazebo is used for the pose estimate required for the proposed method. The robot ends its path 10cm before reaching the goal due to the stopping conditions in-place during model training. The robot is observed to replicate the same feasible paths, in Fig. 5.8 as observed in the kinematics simulator.

With AWS-DR : The trained Unit Distance Model is further evaluated with the AWS-DR in a real-world experimental setup. Due to absence of any absolute pose estimation sensors as addressed in previous Chapter 5. the laser scan matching algorithm is used to provide a pose estimate during this evaluation task. Although the AWS-DR model in Gazebo closely models the real-vehicle, the observed evaluation performance as in Fig. 5.9 deviates from the expected trajectory path. Additionally, the left-turn trajectory Fig. 5.9 (a) is observed to be asymmetric to the right-turn path Fig. 5.9 (c). Moreover, the straight path is observed towards the right nearing the end of the trajectory. This deviation could be caused by two natural possibilities -

1. The laser scan matching pose estimate is inaccurate and hence produces incorrect sequence of actions

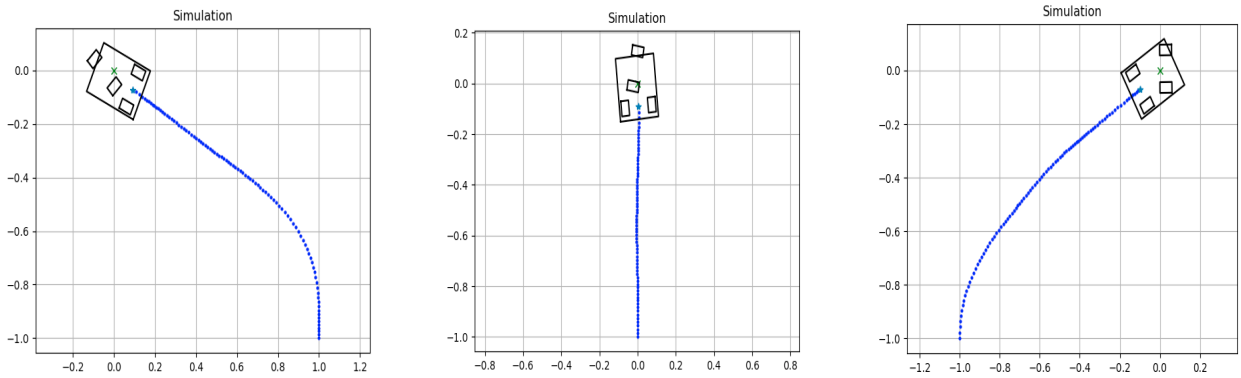
Figure 5.5: Reward curve during training in low-fidelity simulator

2. There are steering asymmetries in the vehicle physics which causes the vehicle to steer right without any steering command.

However, the trajectory paths in Fig. 5.9 are in the robot frame of reference and the laser scan matching tool is evaluated in separate experiments in Fig. 5.4. Fig. 5.4 (b) shows a slight deviation to the right. Observed through multiple experiments, it is evident that the AWS-DR has a steering asymmetry which cannot be modelled in a Physics engine unless an RL model is trained on real-world data.

5.1.3 Waypoint Tracking Evaluation

In Low Fidelity Simulator : The trained Unit Distance Model is used for a waypoint navigation task as per the evaluation scheme mentioned in Section 3.6. The model is evaluated for different trajectories with a series of waypoints either a unit-metre distance away as in Fig. 5.10 (a) & (b) or more than a unit-metre distance away as well as seen in Fig. 5.10 (c) & (d). The planned trajectories (a) & (b) are withing feasible region for the robot to track, and are tracked with the



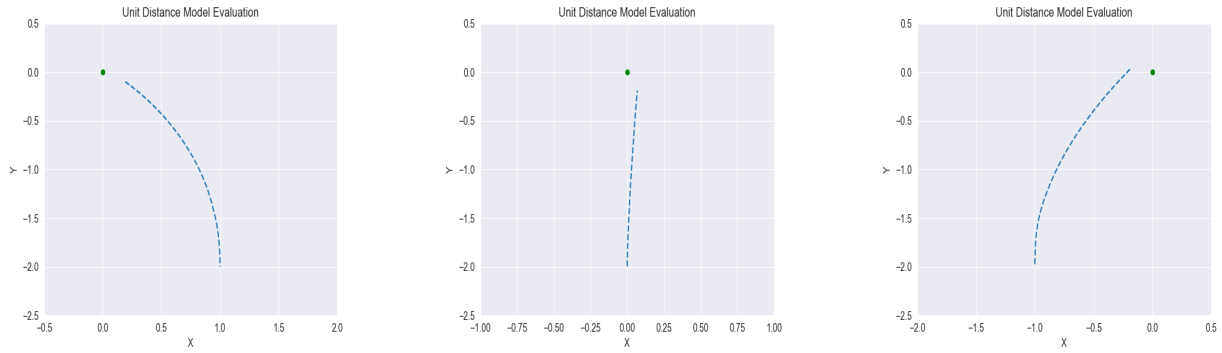
(a) (b) (c)

Figure 5.6: Unit Distance Model Evaluation on Dubin's model in low-fidelity simulator

(a) (b)

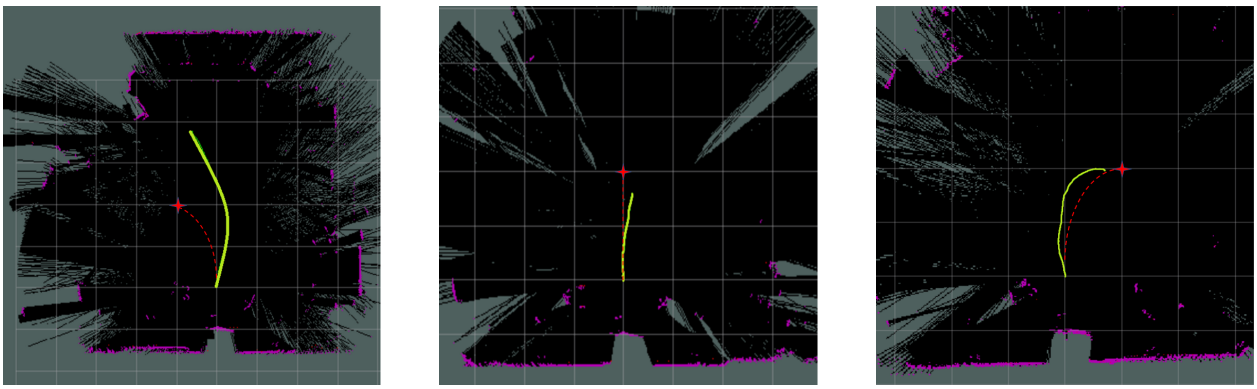
Figure 5.7: Dubin's Model Action Space **(a)** Velocity and **(b)** Steering Angle Profiling

most-optimal path by the robot reaching the final target point in both cases. On the other hand, the trajectories (c) & (d) test the performance of the model for almost-infeasible paths. The robot tries to track all the waypoints but is constrained by its maximum radius of curvature with the ackermann steering.



(a) (b) (c)

Figure 5.8: Unit Distance Model Evaluation in Gazebo



(a) (b) (c)

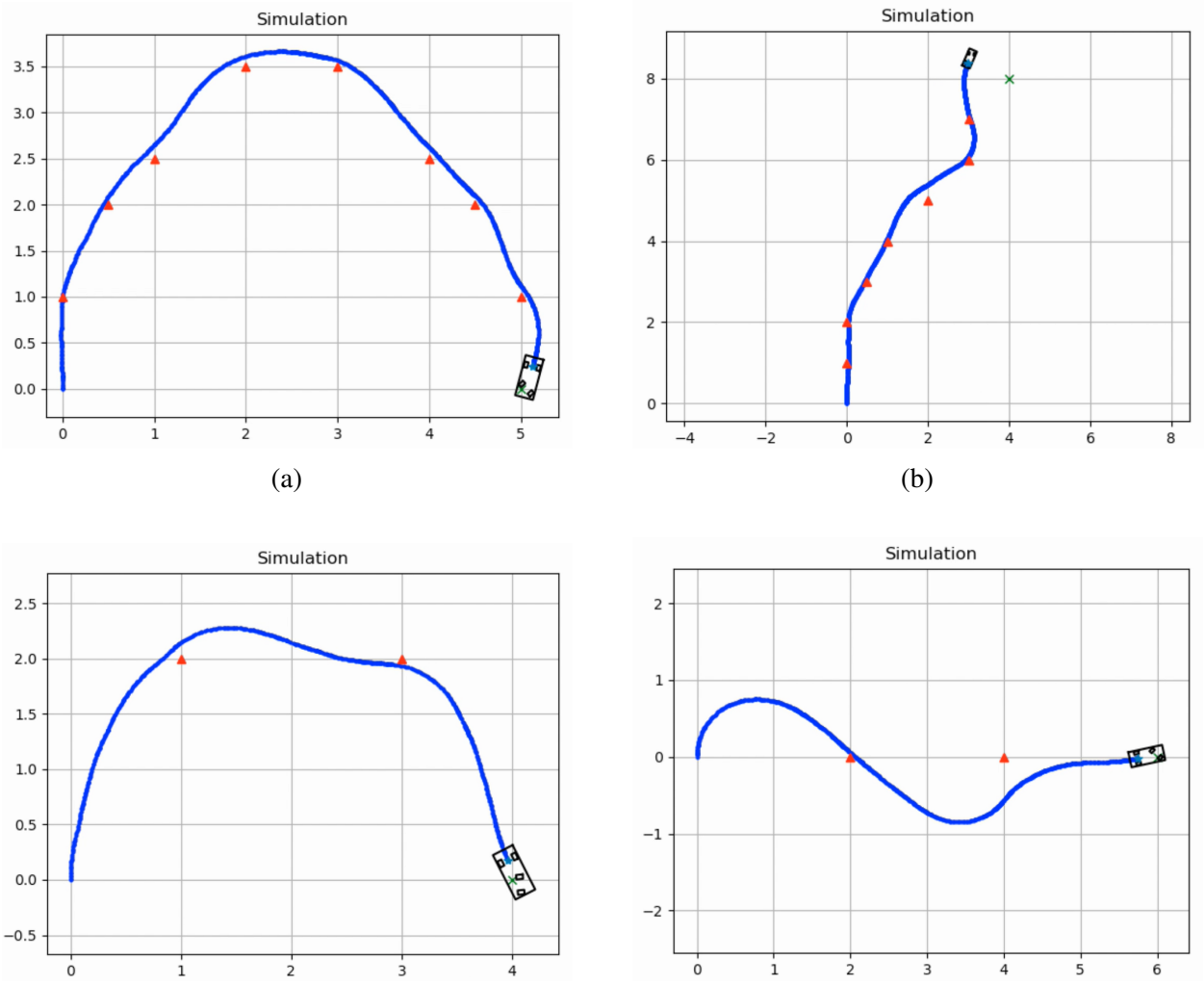
Figure 5.9: Unit Distance Model Evaluation on DR with optimal trajectory in red.

5.2 Turtlebot3 Burger

Since the TB3-Burger follows Differential Drive constraints, it was trained with a Differential Drive Kinematic model for different tasks in the simulators before transferring to the real world.

5.2.1 Unit Distance Model

In Low Fidelity Simulator : Following the proposed training scheme from section (3.5), a unit-distance RL model for navigating from a fixed point in space (origin) to a randomly assigned goal point over a square boundary was trained using the PPO algorithm from Stable Baselines3



(a) (b)

(c) (d)

Figure 5.10: Evaluation of Waypoint Tracking in Low-Fidelity Simulator

(PyTorch), this time with the differential drive kinematics. The training reward curve in Fig.5.11 converged within 200 episodes taking just 45 minutes on an Intel i7-8550U CPU machine without parallelization. It is observed that the differential drive model is easier to train from the lesser number of training iterations required. This is justified as the differential drive kinematics can cause the axles to follow any continuous path in a 2D plane by first rotating itself to point in the direction of the desired goal point (without translational motion) and then translating linearly towards it. The most optimal tracking involves only the Euclidean distance travelled by the central axle of the robot. The Unit Distance Model was evaluated with different target points same as the Dubin's Model. Since the optimal policy learns to navigate with the shortest feasible path (learnt as per reward function design), the navigation paths as seen in Fig. 5.12 are almost a straight path after a turn-in-place rotation by the robot in the goal direction. The robot evaluates successfully for each of the different target points with a similar observed behaviour.

In Gazebo : The above trained UDM is evaluated in Gazebo for the same target points. A built-in \odom topic from Gazebo is used again for the pose estimate required for the proposed method. The robot is observed to replicate the same optimal paths, per Fig. 5.13, as observed in the low-fidelity simulator.

With TB3 : The trained Unit Distance Model is further evaluated with the AWS-DR in a real-world experimental setup. Since the available pose estimate from the wheel encoders onboard the TB3, and no motion asymmetries, the robot replicates the same behaviour as in Gazebo. Thus, an RL model trained in a low fidelity simulator was easily transferable to the real-world in the absence of physical asymmetries and sensor uncertainties.

5.2.2 Waypoint Tracking Evaluation

In TB3 Due to the observed easy transferability from the low fidelity simulator to the real-world, the UDM trained is evaluated for waypoint navigation in an experimental setup. The model was evaluated for a single trajectory with a sequence of waypoints. The robot precisely follows the planned trajectory with an observed deviation of upto 15cm from the optimal path.

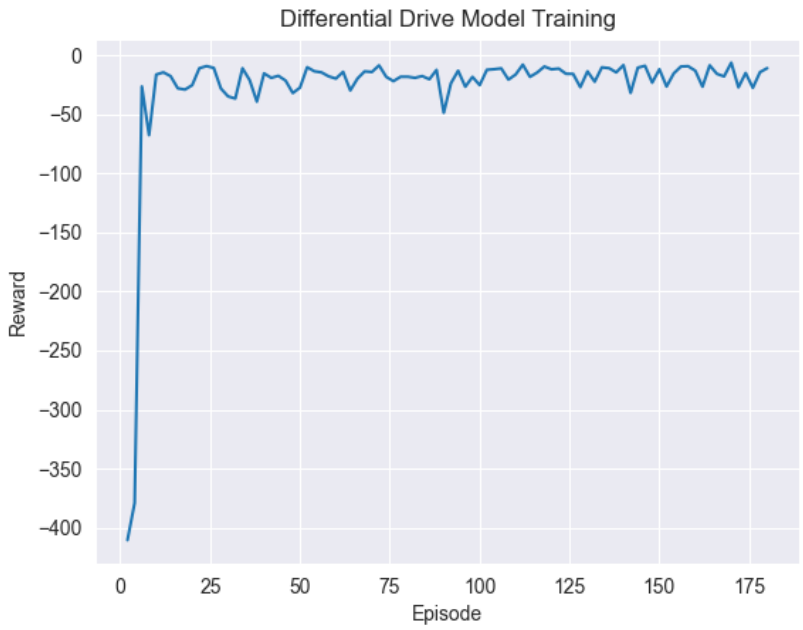
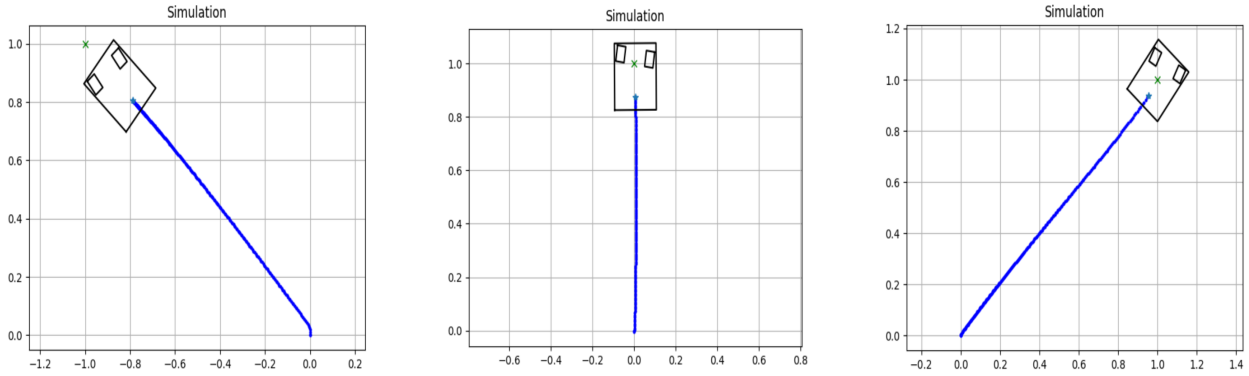


Figure 5.11: Reward curve during training in low-fidelity simulator

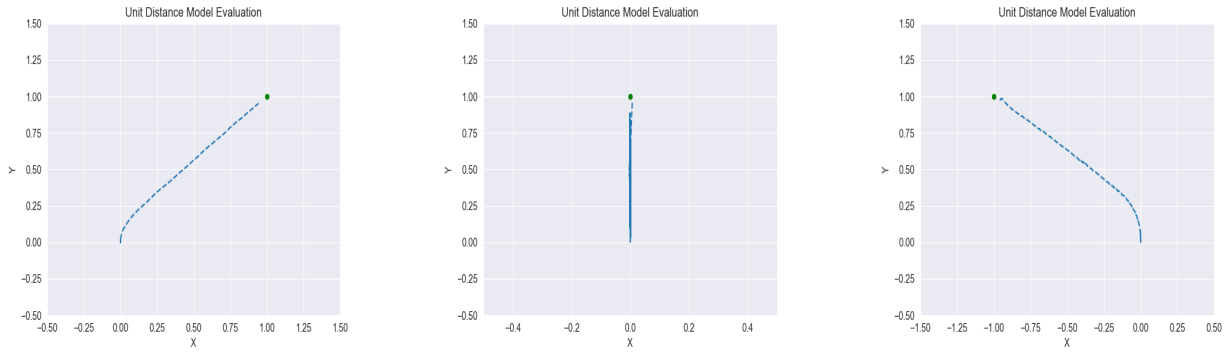


(a) (b) (c)

Figure 5.12: Unit Distance Model Evaluation on Differential Drive Model in low-fidelity simulator

5.2.3 Obstacle Avoidance

The differential drive is further trained for an obstacle avoidance task. Since the obstacles or lidar sensors required for an obstacle avoidance training setup are not easily modelled in the low-fidelity simulator, the UDM is trained in Gazebo. This UDM was trained with a similar training scheme



(a) (b) (c)

Figure 5.13: Unit Distance Model Evaluation on Differential Drive Model in Gazebo simulator

as in previous sections using the PPO algorithm from Stable Baselines3 (PyTorch). Although the reward curve, as seen in Fig. 5.14, converges in about 1000 episodes, the model is trained further for smoother trajectories and a higher success rate during evaluation. Being trained in Gazebo, the simulation frames take close-to real-time for execution and cannot be accelerated. This model achieved desirable performance after upto 8 hours of training on an Nvidia GeForce 2080 Ti GPU.

During evaluation, the robot is tasked with navigating to the same points it experience during training. As seen in Fig. 5.15, it achieves the desired goal it experienced during training. The trajectory by nature is a reactive path as the robot gradually deviates away from the obstacle and one it out of its path, it directs itself towards the desired goal. In trajectories (a) & (b), the optimal route (shortest / fastest path to goal) is from the left of the obstacle. However, it is observed to navigate from the right side.

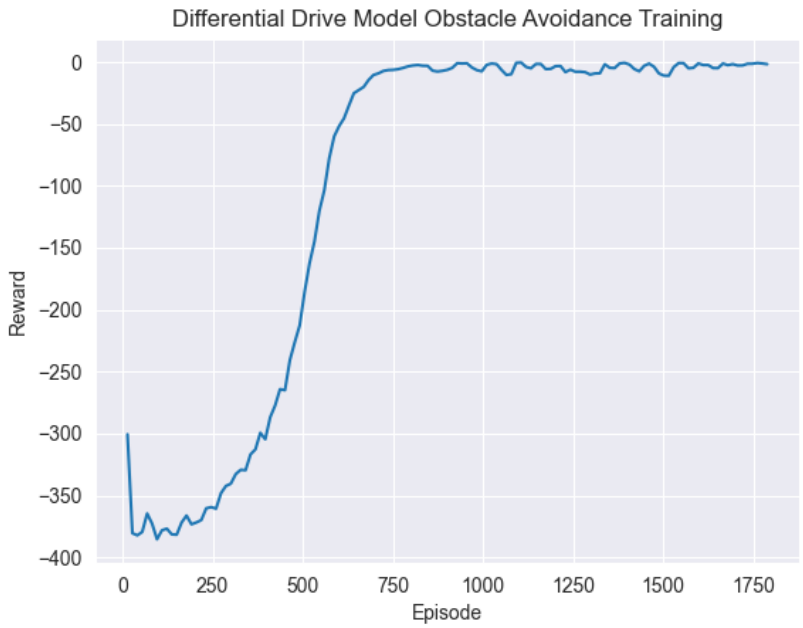


Figure 5.14: Reward curve during training in Gazebo for obstacle avoidance

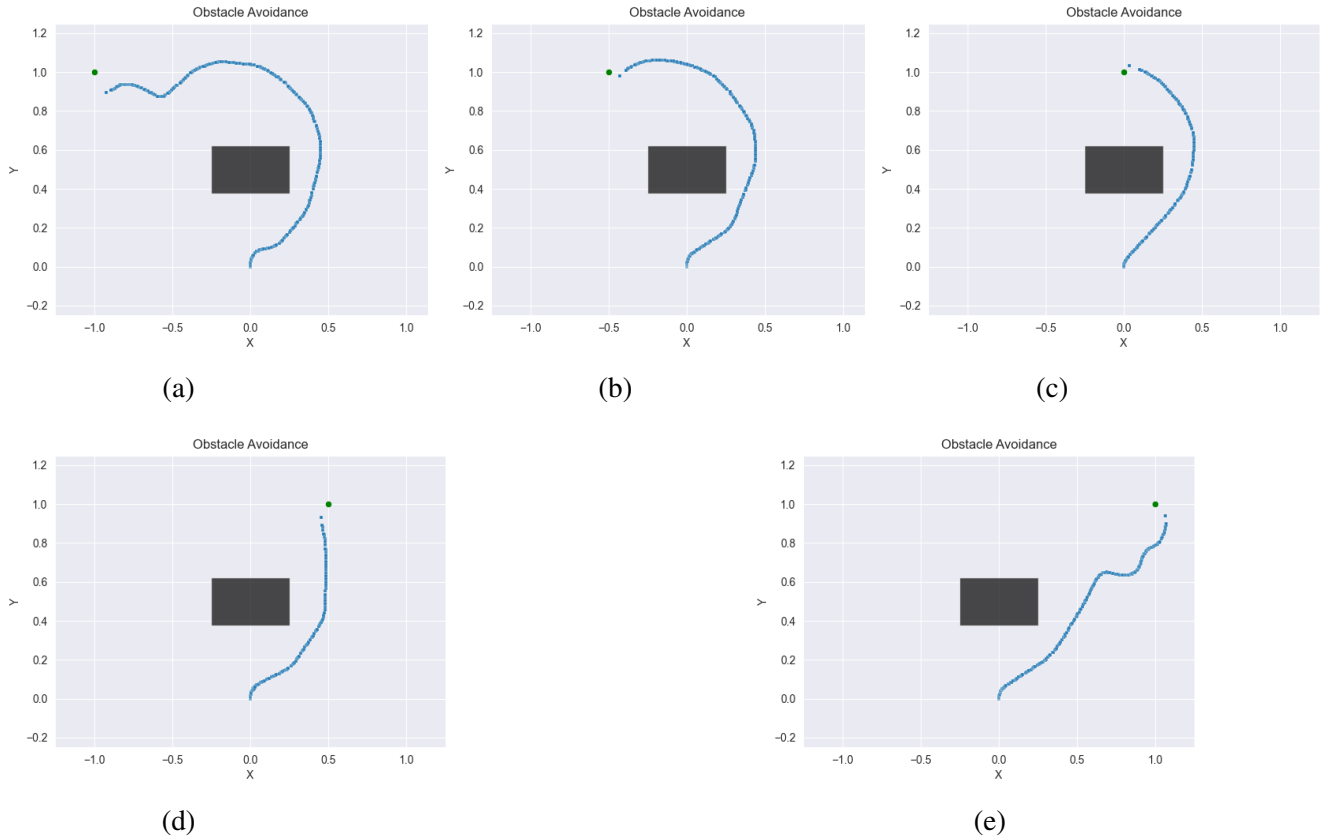


Figure 5.15: Obstacle Avoidance Evaluation

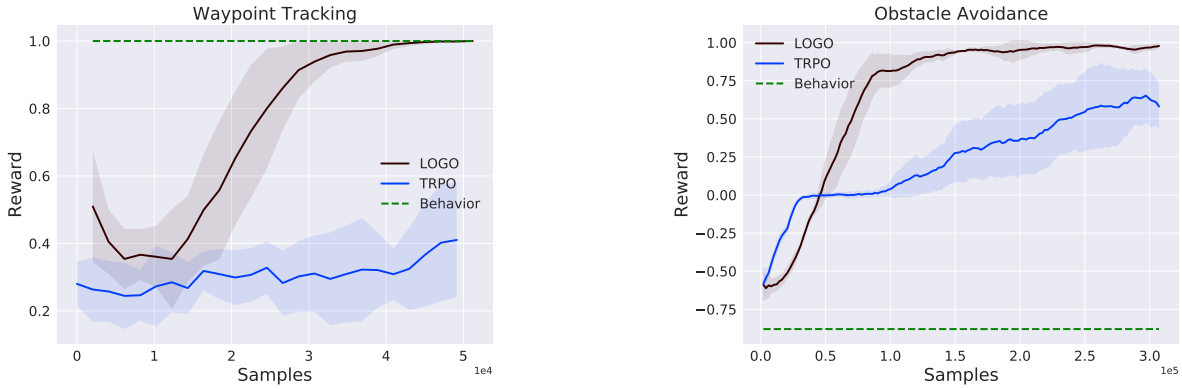
6. APPLICATION

The proposed methodology for autonomous waypoint tracking is used to evaluate a custom-built RL algorithm LOGO in a real-world setting. The algorithm addresses the challenging problem of reward sparsity in complex reinforcement learning environments such as navigating autonomously in unknown environments. As seen in this research thesis, a simple low-fidelity simulator was used to design an effective reward function for every step taken by the robot to achieve the task at hand. However, the most intuitive reward function is of a very sparse nature, which goes as - a reward of $+10$ for reaching the desired goal point, -1 penalty for every collision detected and 0 reward otherwise. Thus, current RL approaches depend on a carefully crafted reward function (possibly from domain experts) as a feedback to the agent. Similar to the strategy of Trust Region Policy Optimisation (TRPO) where the target policy is within a trust region of the current policy and is proven to perform well in a dense reward setting; the test algorithm here uses a behaviour policy trained in a dense reward setting with partially observed data to guide exploration in the sparse reward setting with full observed state.

6.1 Results

The training performance extraordinary improvement compared to a standard TRPO reward curve as seen in Fid. 6.1. The algorithm is evaluated on the waypoint tracking & obstacle avoidance task for the differential drive robot, Turtlebot-3 Burger.

Waypoint Tracking: The goal is to train a policy that takes the robot to an arbitrary waypoint within 1 meter of its current position in an episode of 20 seconds. The episode concludes when the robot either reaches the waypont or the episode timer expires. The state space of the agent are its $x, y,$ coordinates and orientation ϕ to the waypoint. The actions are its linear and angular velocities. The agent receives a sparse reward of $+1$ if it reaches the waypoint, and 0 otherwise. We created a sub-optimal Behavior policy by training TRPO with dense rewards on our own low fidelity kinematic model Python-based simulator with the same state and action space. While it shows



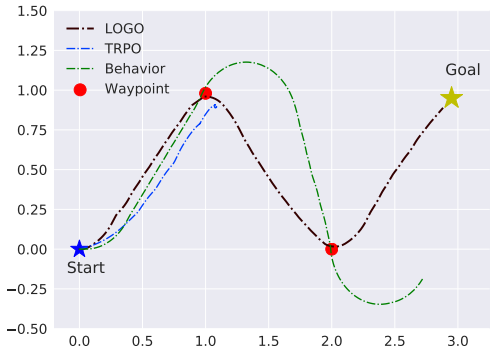
(a) (b)

Figure 6.1: Training Reward Convergence

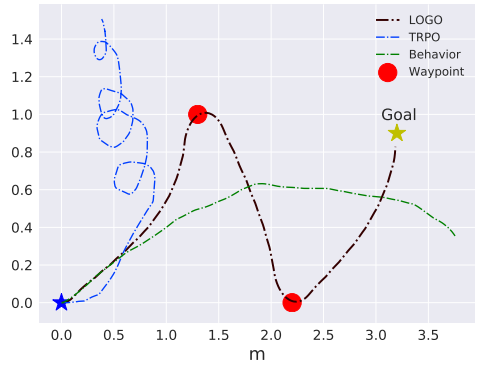
reasonable waypoint tracking, the trajectories that it generates in Gazebo are inefficient, and its real-world waypoint tracking is poor Fig. 6.2 . As expected, TRPO shows poor performance in this sparse reward setting and often does not attain the desired waypoint before the episode concludes, giving the impression of aimless circling seen in Fig. 6.2**(b)**. LOGO is able to effectively utilize the Behavior policy and shows excellent waypoint tracking seen in Fig. 6.2

Obstacle Avoidance: The goal and rewards are the same as in Task 1, with the addition of an obstacle that must be avoided to attain an arbitrary waypoint, shown in Fig. 6.3. The complete state space is now augmented by a 2D Lidar scan in addition to coordinates and orientation described in Task 1. However, the Behavior policy is still generated via the low fidelity kinematic simulator *without* the obstacle, i.e., it is created on a lower dimensional state space. As seen in Fig. 6.3 **(a)** , this renders the Behavior policy impractical for Task 2, since it almost always hits the obstacle in both Gazebo and the real-world. However, it does possess information on how to track a waypoint, and when combined with the full state information, this nugget is utilized very effectively by LOGO to learn a viable policy as seen in Fig. 6.3. Further, TRPO in this sparse reward setting does poorly and often collides with the obstacle in real-world experiments as seen in Fig. 6.3 **(b)** .

Training: Waypoint Tracking : In navigation problems, we have a global planner that uses a high level map of the bot's surroundings for planning a trajectory using waypoints a unit-meter

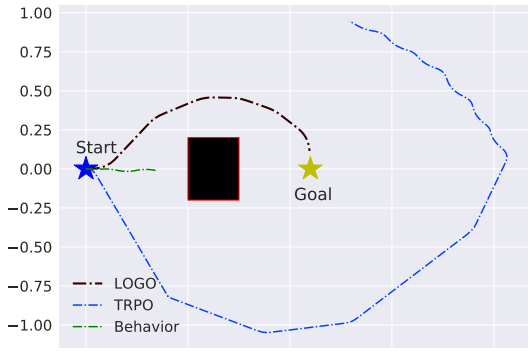


(a) Gazebo Simulator

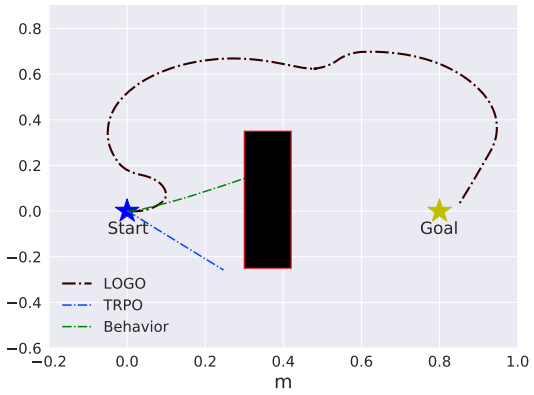


(b) TB3 Burger

Figure 6.2: Waypoint Tracking Evaluation



(a) Gazebo Simulator



(b) TB3 Burger

Figure 6.3: Obstacle Avoidance Evaluation

distance from each other, while the goal of the bot is to achieve these waypoints. To obtain a waypoint tracking scheme, we first train a behavior policy as per proposed framework in this research thesis. In each training episode, the bot is reset to the origin and a waypoint is randomly chosen as $x_w \sim \text{uniform}([-1, 1]), y_w = 1$. The episode is terminated if the robot reaches the waypoint, or if it crosses the training boundary or exceeds the maximum episode length. The behavior policy obtained after training in the low fidelity simulator is then used in the LOGO algorithm training on Gazebo. LOGO is trained in Gazebo with sparse rewards, where a reward of $+1$ is provided if the bot reaches the waypoint, and 0 otherwise. We evaluate the trained policy in Gazebo and the real world as per Fig. 6.2, by providing it a series of waypoints to track in order to reach its final goal.

Training: Obstacle Avoidance : We train our bot for obstacle avoidance in Gazebo using the behavior policy described in the section above. The goal is to use the skills of waypoint navigation from the behavior policy to guide and learn the skills of obstacle avoidance. The state space includes the Lidar scan values in addition to the relative state space reprsentation described previously. The `/scan` provides 360 values, each of these indicate the distance to the nearest object in a 1 sector. For the purpose of our experiments, we use the minimum distance in each 60 sector. This reduces the Lidar data to 6 values. We train our algorithm on Gazebo with a fixed obstacle for random waypoints. In each training episode, the bot is reset to the origin and a waypoint is generated similar to the previous section. The episode is terminated if same conditions in the previous section are satisfied or if a collision with the obstacle occurs. We demonstrate the performance of our algorithms both in Gazebo as well as the real-world as shown in Fig. 6.3.

6.2 Implementation Details

We implement all the algorithms in this evaluation task using PyTorch. For all our experiments, we have a two layered (128×128) fully connected neural network with $tanh$ activation functions to parameterize our policy and value functions. We use a learning rate of 3×10^{-4}, a discount factor $\gamma = 0.99$, and TRPO parameter $\delta = 0.01$. We decay the influence of the behavior policy by decaying δ_k. We start with δ_0, and we do not decay δ_k for the first K_δ iterations. For $k > K_\delta$, we

geometrically decay δ_k as $\delta_k \leftarrow \alpha\delta_k$, whenever the average return in the current iteration is greater than the average return in the past 10 iterations. In table 6.1 we provide details on the demonstration data collected using the behavior policy.

Environment	Offline $\mathcal{S}$	Online $\mathcal{S}$	$\mathcal{A}$	Samples	Average Episodic Reward
Waypoint traking	$\mathbb{R}^3$	$\mathbb{R}^3$	15	Policy	1
Obstacle avoidance	$\mathbb{R}^3$	$\mathbb{R}^9$	15	Policy	-0.88

Table 6.1: Demonstration data details

7. FUTURE SCOPE AND CONCLUSION

7.1 Conclusion

This work was motivated by the current challenges in dynamic path planning of mobile robots in unknown environments.

In this thesis work, a mapless autonomous waypoint tracker is dynamically able to navigate in the real-world with the use of onboard sensor data. We trained the agent first in a low-fidelity simulator to fine-tune the reward design, algorithm selection and the state & action space. We then train a transferable model replicating the same RL design in Gazebo, which is a more complex Physics engine and then evaluate the trained optimal policy in the real world. We have used a relative state vector and trained in a randomized environment to make it a generalized model for any pair of initial and target points. A 36-dimensional sectorized lidar scan is included in the state vector for the obstacle avoidance task. Compared to conventional move-base planners or DWA local planners, our approach is more effective in complicated environments. The model evaluates well on Turtlebot3 in the real world equipped with the servo motors with Dynamixel wheel encoders and the 2D RP Lidar.

We also addressed some of the challenges associated with implementing the same approach for a high speed Dubin's car model. Although the low-fidelity simulator and Gazebo simulation did not show any difference in performance for the same waypoint tracking algorithm, there were significant challenges in transferring the same on a real-robot (AWS DeepRacer). A canonical laser scan matcher was used to incrementally localize and map the environment from the previous and current laser scan readings. However, the brushless DC motors are difficult to control with a pure RL algorithm. Since the DeepRacer has a servo steering mechanism, it is easier to control the steering for lateral control compared to the longitudinal control. Thus, it is more suited for an application such as a lane following task where primarily lateral control is expected.

7.2 Future Scope

From the detailed analysis and conclusions of this thesis work, we can suggest future directions to this research and potential improvements in the design.

Dynamic Obstacle Avoidance This thesis work can be extended to a dynamic obstacle avoidance task by augmenting the state vector with a history of lidar scan data, or with depth vision to estimate the relative velocity of the dynamic obstacles.

Real World Data Challenges with sensor uncertainties/inaccuracies and robot asymmetries as seen in Section 5.1.1 can be overcome with gathering real-world data from the robot from demonstration runs along a path. This data can be used to aid the policy training in Gazebo.

Domain Randomization Challenges with sensor/environment uncertainties can be overcome with domain randomization of state vector. Also, domain randomization in the low fidelity simulator can make the optimal policy directly transferable to the real-world robot as it can handle minor noise/deviations in sensor data.

Semantic segmentation for autonomous navigation Semantic segmentation of the robot's surrounding can help in training the robot to autonomously navigate to certain objects in its local neighbourhood. This thesis work can be extended by including a monocular camera on the robot and using semantic targets for waypoint navigation.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Y. Cheng and G. Y. Wang, "Mobile robot navigation based on lidar," in *2018 Chinese Control And Decision Conference (CCDC)*, pp. 1243–1246, 2018.

[3] W. Khaksar, S. Vivekananthen, K. S. M. Saharia, M. Yousefi, and F. B. Ismail, "A review on mobile robots motion path planning in unknown environments," in *2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*, pp. 295–300, 2015.

[4] 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*, AAAI'16, p. 2094–2100, AAAI Press, 2016.

[5] X. Lei, Z. Zhang, and P. Dong, "Dynamic path planning of unknown environment based on deep reinforcement learning," *J. Robotics*, vol. 2018, pp. 5781591:1–5781591:10, 2018.

[6] H. Surmann, C. Jestel, R. Marchel, F. Musberg, H. Elhadj, and M. Ardani, "Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments," 2020.

[7] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," *CoRR*, vol. abs/1703.00420, 2017.

[8] R. Cimurs, J. H. Lee, and I. H. Suh, "Goal-oriented obstacle avoidance with deep reinforcement learning in continuous action space," *Electronics*, vol. 9, p. 411, 2020.

[9] L. Manuelli, "Reinforcement learning for autonomous driving obstacle avoidance using lidar," 2017.

[10] J. Qiao, Z. Hou, and X. Ruan, "Application of reinforcement learning based on neural network to dynamic obstacle avoidance," in *2008 International Conference on Information and Automation*, pp. 784–788, 2008.

[11] M. Etemad, N. Zare, M. Sarvmaili, A. Soares, and B. Brandoli, "Using deep reinforcement learning methods for autonomous vessels in 2d environments," 03 2020.

[12] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018.

[13] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," *CoRR*, vol. abs/1703.06907, 2017.

[14] A. Rajeswaran, S. Ghotra, S. Levine, and B. Ravindran, "Epopt: Learning robust neural network policies using model ensembles," *CoRR*, vol. abs/1610.01283, 2016.

[15] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," *CoRR*, vol. abs/1703.02702, 2017.

[16] F. Muratore, F. Treede, M. Gienger, and J. Peters, "Domain randomization for simulation-based policy optimization with transferability assessment," in *Proceedings of The 2nd Conference on Robot Learning* (A. Billard, A. Dragan, J. Peters, and J. Morimoto, eds.), vol. 87 of *Proceedings of Machine Learning Research*, pp. 700–713, PMLR, 29–31 Oct 2018.

[17] V. M. Patel, R. Gopalan, R. Li, and R. Chellappa, "Visual domain adaptation: A survey of recent advances," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 53–69, 2015.