EXPLORING DEEP REINFORCEMENT LEARNING TECHNIQUES FOR AUTONOMOUS NAVIGATION

An Undergraduate Research Scholars Thesis

by

VINCENT T. POTTER

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dileep Kalathil

May 2022

Major: Computer Engineering

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Vincent Potter, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Exploring Deep Reinforcement Learning Techniques for Autonomous Navigation

Vincent Potter
Department of Electrical and Computer Engineering
Texas A&M University

Research Faculty Advisor: Dr. Dileep Kalathil
Department of Electrical and Computer Engineering
Texas A&M University

This paper contains research into efficient autonomous navigation algorithms powered by deep reinforcement learning. These algorithms enable a mobile robot to perform waypoint tracking in an indoor environment. The robot does not contain a map of the environment nor does it know the location of the waypoints. A reward function is used to encourage behaviors in the robot that lead it closer to the goal. This is an active area of research encouraged by recent advancements in neural networks applied to sequential decision making. The reinforcement learning algorithms utilize LiDAR and IMU sensors in order to navigate the unknown environment by calculating the robot's current state and what its next action should be. At each step the action that is most likely to yield the maximum reward is sent to the robot in order to follow the sequential targets along the path to the final goal location. I use a low-fidelity custom simulator based on the Dubins Path along with a high-fidelity 3D simulator, Gazebo, to train various policies. The Dubins simulator is constructed from Python and executes very fast, while Gazebo requires more resources but is very advanced. After training is complete, ROS is used to deploy the RL policy onto the physical robot and convert the action commands into linear and angular velocities that can be understood by the robot’s hardware/motors. The TurtleBot3 Burger is the robot being used for evaluation in the real world. Often times, there is a severe drop in performance between the simulator and the real world so this is also monitored and factored into performance. Dense and sparse reward functions are explored in order to mimic various real-world scenarios where the reward is not always known at every step. Finally, Deep Q-Learning, Trust Region Policy Optimization, and a new RL algorithm called Learning Online with Guidance Offline are implemented and tested throughout the course of the research.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Kalathil for his support throughout the course of this research. Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience. Finally, thanks to graduate students Desik Rengarajan, Jaewon Kim, and Akshay Sarvesh for their encouragement and guidance.

Some materials used for the Exploring Deep RL Techniques for Autonomous Navigation Project were provided by Dr. Kalathil and Desik Rengarajan. The analyses depicted in the Deep RL for Autonomous Navigation Project were conducted in part by Desik Rengarajan and were published in 2022.

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

Funding Sources

This project received no funding.

NOMENCLATURE

RL	Reinforcement Learning
ROS	Robot Operating System
LiDAR	Light Detection and Ranging
SLAM	Simultaneous Localization and Mapping
DQN	Deep Q-Network
TRPO	Trust Region Policy Optimization
LOGO	Learning Online with Guidance Offline
OpenCR	Open Control Board for ROS

1. INTRODUCTION

1.1 Background

Many current path planning algorithms for autonomous navigation take advantage of GPS and known environments for determining an optimal route from point A to point B. Additionally, front and rear cameras incorporate object detection systems for obeying traffic laws and avoiding pedestrians. This includes many popular approaches such as A* search and SLAM techniques. However, these setups can struggle under highly dynamic situations or unknown environments [1].

Khaksar et al. [2] concluded in a study that hybrid approaches using neural networks and learning models provide advantages in these scenarios. Training a reinforcement learning policy without a map has applications in exploration, surveillance, search and rescue operations, and autonomous vehicular navigation since the environment in these scenarios is often dynamic or completely unknown. This project explores opportunities to improve upon recent success reinforcement learning has shown in mobile robot autonomous navigation within indoor environments.

The three RL algorithms tested are Deep Q-Learning (DQN), Trust Region Policy Optimization (TRPO), and Learning Online with Guidance Offline (LOGO). DQN is an off-policy method for discrete action spaces using Q-values. TRPO is an on-policy method for discrete or continuous action spaces using an Advantage metric. LOGO is an expansion upon TRPO that incorporates offline learning which proves to be especially useful for sparse environments [3].

The idea for this research comparing RL algorithms and reward structures was inspired from AWS' DeepRacer program. Individuals compete across a variety of virtual 'race tracks' to demonstrate the speed and intelligence of robots trained with reinforcement learning algorithms. However, this project also addresses the inconsistencies that are experienced when a policy is transplanted from a virtual simulator to physical hardware. That is why extensive work is done physically testing the robot's accuracy throughout different tasks.

1.2 Overview

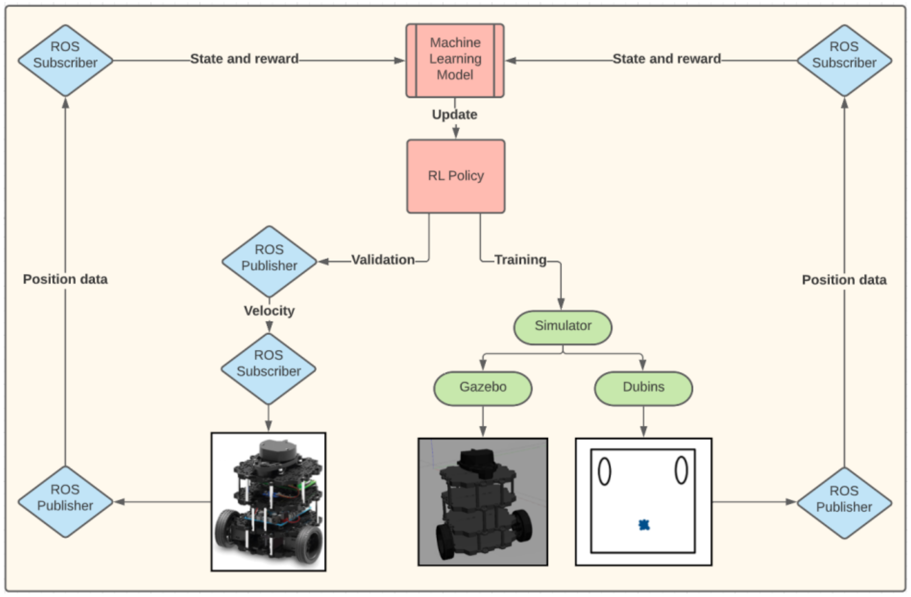


Figure 1.1: Subsystem Diagram.

The structure for this project consists of three main subsystems. The ROS subsystem seen in blue, the reinforcement learning subsystem seen in red, and the simulator and physical robot subsystem seen in green above in Figure 1.1. Each of these subsystems are developed to function independently for ease with debugging and unit testing. Then, they are combined together for integration and end-to-end testing. I explain below the function and implementation of each subsystem.

1.2.1 ROS Subsystem

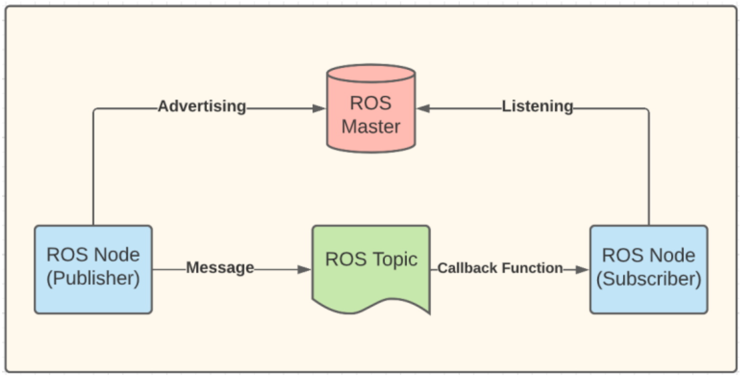


Figure 1.2: ROS Message Structure.

All communication between components is completed through ROS messages, visualized in Figure 1.2. ROS uses messages containing data that is then published to topics that other ROS nodes can subscribe to. Multiple ROS nodes can connect to a single ROS master that allows these topics to become accessible. I use between 4-6 ROS publishers/subscribers for this project depending on whether obstacle avoidance is desired. Linear and angular velocity for the robot to execute is published to the ‘command_velocity’ topic using the ‘Twist’ data structure which consist of two 3D arrays. These arrays correspond to the x, y, and z components of linear and angular velocity. A subscriber located onboard of the robot listens for this data. The current state of the robot is published from the robot to a subscriber in the reinforcement learning subsystem through the ‘odometry’ topic which contains the pose (x, y, and z coordinate) and orientation. Finally, if the LiDAR sensor is used for obstacle avoidance, then there is an additional publisher/subscriber pair communicating over the ‘scan’ topic containing the range and angle of the LiDAR scan.

Since these subsystems are built independently, the instrument used to control the robot and send velocity commands can vary. For this project, I used the following instruments increasing in difficulty as I got more familiar. To begin, the teleop package is used to control the physical robot or in the simulator with the arrow keys on a keyboard. Twist messages can also be manually constructed from the command line and published directly to the 'command_velocity' topic. Finally, control is given to the RL policy which produces the ROS messages without requiring input from the user for autonomous navigation. However, the manual control options are very useful for performing tests of the subsystem.

1.2.2 Reinforcement Learning Subsystem

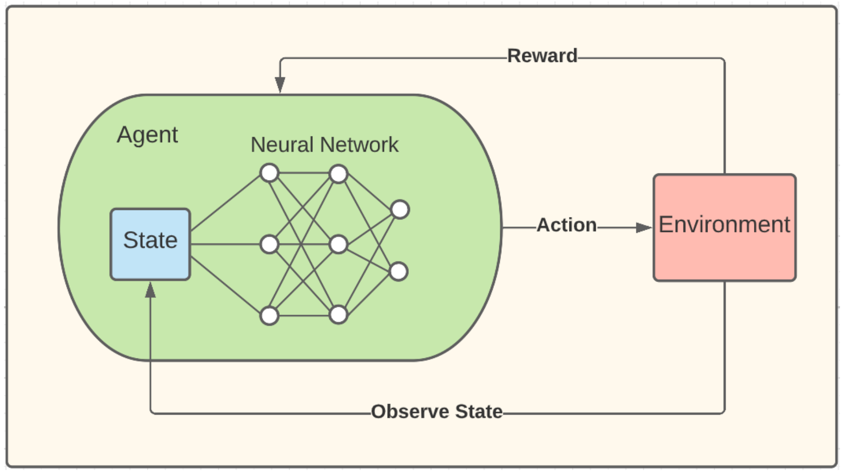


Figure 1.3: Deep Reinforcement Learning Process.

An overview of the general flow this subsystem takes is shown in Figure 1.3. All components of the reinforcement learning subsystem are constructed with Python. This includes the implementation of the agent, environment, actor-critic networks, DQN, TRPO, LOGO, and training process. The rospy package is used for the subscriber to the robot's odometry and publisher to its velocity. Using command line arguments, I can swap between combinations of continuous or discrete and sparse or dense environments. This alters the reward function used and the size of the action space. Through the environment, I can initialize the starting location and direction of the robot along with the location of the waypoints.

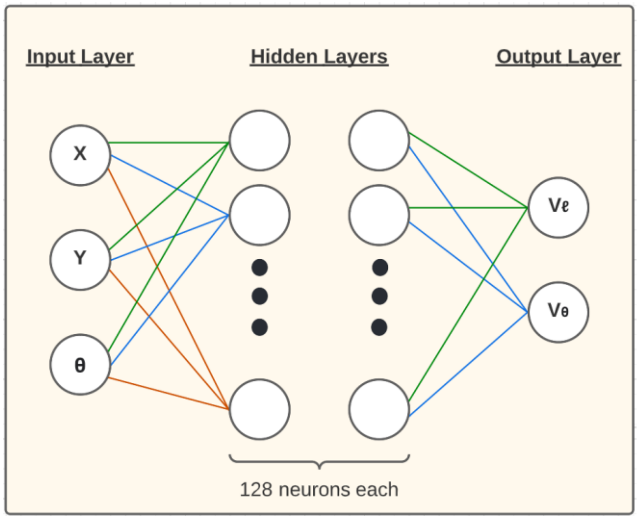


Figure 1.4: Neural Network Visualization.

Three reinforcement learning algorithms are used. DQN is used for the obstacle detection task while TRPO and LOGO are used for pure waypoint tracking tasks involving the dense and sparse reward structures. To draw a clearer comparison between the performance of TRPO and LOGO under dense and sparse rewards obstacle avoidance tasks were not tested but it should be noted that both of these algorithms can also handle that problem domain. All of these algorithms use some variation of the neural network seen above in Figure 1.4. Obstacle detection requires additional neurons at the input layer representing the distance between the robot and nearest object determined by the LiDAR sensor and uses four more fully connected layers. To simplify the obstacle avoidance, the network also uses one less output. The linear velocity was held constant to allow plenty of time to react to objects as they appear in LiDAR while only the angular velocity was controlled using the action from the output layer of the network.

$$Dense\ Reward = \begin{cases} +10 & within\ goal \\ -1 & out\ of\ bounds \\ -(\lambda_1(l_d sin(\theta_1 - \theta_0)^2 + \lambda_2|x_1 - x_0| + \lambda_3|y_1 - y_0| + \lambda_4(\theta_1 - \theta_0)) & otherwise \end{cases}$$

where,
$\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are hyperparameters
l_d is the distance to the target
x_1, y_1, θ_1 are the current coordinates of the robot
x_0, y_0, θ_0 are the coordinates of the goal

$$Sparse\ Reward = \begin{cases} +1 & within\ goal \\ 0 & otherwise \end{cases}$$

Figure 1.5: Reward Functions.

Figure 1.5 shows both the dense and sparse structures used to calculate the reward for the robot at each step of the training or evaluation process for the pure waypoint tracking tasks. The dense reward function penalizes behavior based on three types of path-tracking error (cross-track, along-track, and heading error) to produce more optimal routes. A penalty is also given for exceeding certain boundaries. Once the robot reaches within a desired threshold of the waypoint it is given +10. For the sparse reward structure, the robot is only rewarded upon reaching the goal with +1. During all other steps, the robot does not receive a reward (0). This significantly increases the difficulty of training since it can take a very long time for the robot to learn a proper policy to follow. This scenario mimics some real-world environments where it is not always clear if you are getting closer to the goal until actually reaching it.

1.2.3 Simulator and Robot Subsystem

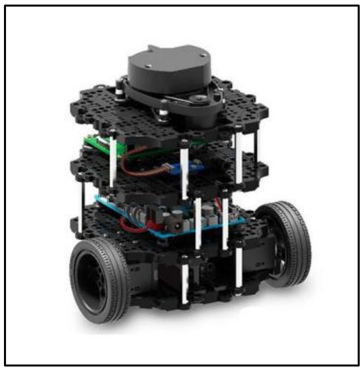


Figure 1.6: TurtleBot3 Burger Robot.

This subsystem contains all the hardware of the physical robot, the software running onboard, and the simulator software. The TurtleBot3 Burger is a small and open-air robot as seen in Figure 1.6. It is 138mm x 178mm x 192mm (L x W x H) with an outer radius of 105mm. It contains 2 Dynamixel servo motors driven by an 11.1V battery, a 360-degree LiDAR sensor, a 3-axis gyroscope, and 3-axis accelerometer. Additionally, there is a Raspberry Pi 4 for communicating via the ROS master with a remote PC executing the RL program and an OpenCR board equipped with a 32-bit ARM Cortex-M7 for communicating with the wheel motors.

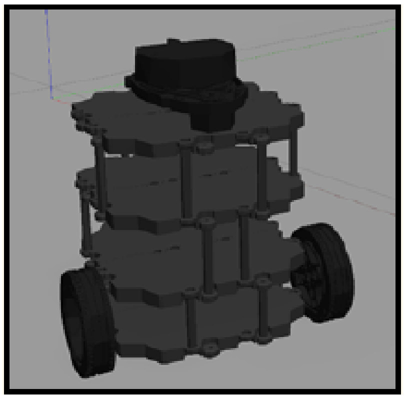


Figure 1.7: Gazebo TurtleBot3 Burger Model.

The Gazebo simulator has advanced physics principles that govern the vehicular dynamics. It also contains a 3D rendered environment where a constructed model of the robot, seen in Figure 1.7, can be interacted with through a ROS framework. Therefore, state information and actions are communicated via actual ROS nodes like with the physical bot. The design of the model can be fully customized to match the real-world version through a spatial data file containing max speed, weight, dimensions, coefficient of friction, number of LiDAR samples, etc. for the robot. Gazebo is used for training the DQN for obstacle avoidance because objects and whole rooms can easily be created and manipulated throughout the rendered world. These features allow for closely mimicking the real world to produce a robust model even when deployed. However, the GUI takes a considerable amount of computing resources to render. Additionally, due to the more advanced vehicular kinematics, the training process is slowed considerably.

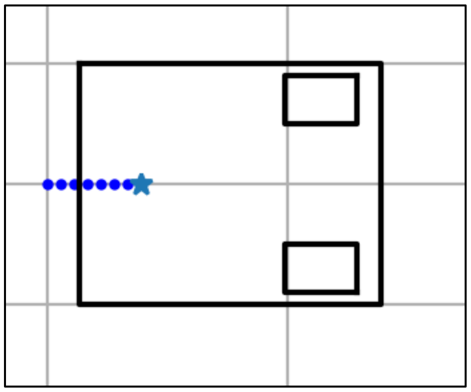


Figure 1.8: Dubins TurtleBot3 Burger Model.

The custom Dubins simulator is much simpler and thus can be executed much faster. It is built using a 2D top-down model of the vehicle, seen in Figure 1.8, moving through an environment rendered using Matplotlib. The state and action are gathered/sent using functions that interact with the Matplotlib figure rather than with ROS. This allows for the training process to be accelerated rapidly. While some of the more realistic physics and the ability to spawn objects is sacrificed, the model can reach an optimal policy for simpler tasks much quicker than with Gazebo. Therefore, for waypoint tracking without obstacles this is the more efficient manner for training. Models trained in this simulator are also deployed into the real world to ensure the robot is still reaching within the desired threshold of all waypoints.

1.3 Operating Concept

1.3.1 Scope

This project only validates autonomous navigation on ~1/18th scale vehicles. Additionally, the target use is for indoor environments only. It is used to explore theoretical reinforcement learning techniques and provide proof of concepts that can then be scaled up to larger self-driving robots and vehicles. Obstacle detection is tested only for stationary objects. The obstacles can be of any size as long as it is viewable by the onboard LiDAR sensor.

1.3.2 Operational Description and Constraints

The project's main objective is to research various applications of deep reinforcement learning in autonomous navigation and sequential waypoint tracking accuracy. Therefore, the algorithms produced are for academic use and study. They are evaluated in comparison with each other in various task domains. They also prove that deep learning techniques can succeed in dynamic situations involving autonomous navigation. Therefore, the following constraints exist for this project:

- The robot must be equipped with an onboard 360^{o} LiDAR sensor for obstacle avoidance model.
- The robot is operated indoors within a confined space.
- The robot exhibits symmetrical steering.

1.3.3 Users

For the model and scale being worked on in this project the users will be myself and other graduate students. Since this project is being conducted for research purposes there are many trials with the results being recorded and documented for others. The robot navigates autonomously and requires extensive training to be done beforehand in complex simulator environments. Therefore, it is difficult to provide enough education for other users to operate the system in its current state. Initially the main benefactors will be the academic community. Discovering the performance of various algorithms for path planning and obstacle avoidance is an ongoing area of deep reinforcement learning as the use of neural networks is growing in popularity.

However, once this technology is proven, its applications and users are endless. For example, it can be applied to exploration and surveillance scenarios where the environment may be hostile to humans or simply unknown. Another popular user would be manned or unmanned commercial vehicles. Even outside of autonomous navigation, many problems can be mapped to sequential decision making where deep reinforcement learning techniques can be applied.

1.4 Sample Scenarios

1.4.1 Competitions

This first scenario involves racing conditions similar to the AWS DeepRacer program. The goal is speed and efficiency navigating around a defined track. There are two types of competitions: head-to-head and obstacle avoidance. While the goal is to win with the fastest time in both, head-to-head races involve racing against multiple other cars to post the best lap time. Obstacle avoidance races add the additional difficulty of dodging items placed on the track and other vehicles.

1.4.2 Environment Exploration

Exploring an unknown environment requires the use of a deployed version of the reinforcement learning algorithm. The robot could be used to develop a 3D scan of the surroundings through the use of the LiDAR sensor as it is navigated between waypoints set throughout the environment. Therefore, more information can be gathered in conditions where humans cannot reach the environment or as a precaution before humans are sent.

2. METHODS

2.1 Training

To begin the training process in the simulator, I activate a Conda environment with Gym, Pytorch, Numpy, Matplotlib, and Tensorboard installed. Then, in a python file I import the files containing the environment, actor-critic network, the chosen RL algorithm, and the agent. Using command line arguments, I select the type of environment I'd like to train on and which algorithm I'd like to use. I can also edit the hyperparameters and if/when to save the trained model.

Next, the environment is initialized defining the model of the vehicle, its spawn location, the waypoint location, the action space, the reward function, the episode timer, and maximum boundaries the robot should not exceed. Once the robot and waypoint are spawned, the current state of the agent is observed. The current state consists of the robot's x-coordinate, y-coordinate, and heading given by an angle. Then, the actor-critic networks estimate the advantage from the trajectories. The networks are updated and an action is output for the robot to take. Next, we perform a policy optimization step. The state of the robot is updated to its next state based on the action that was output. The new state is checked if it is within the desired threshold of the goal and the reward is determined based on its position relative to the goal. Negative rewards are used to deter the robot from navigating outside of bounds or farther from the waypoint. Positive rewards are given for achieving the goal. Finally, this movement is modeled in the rendered environment. This process of updating the policy is repeated until optimal actions that maximize the reward function are output based on the current state of the robot at each step.

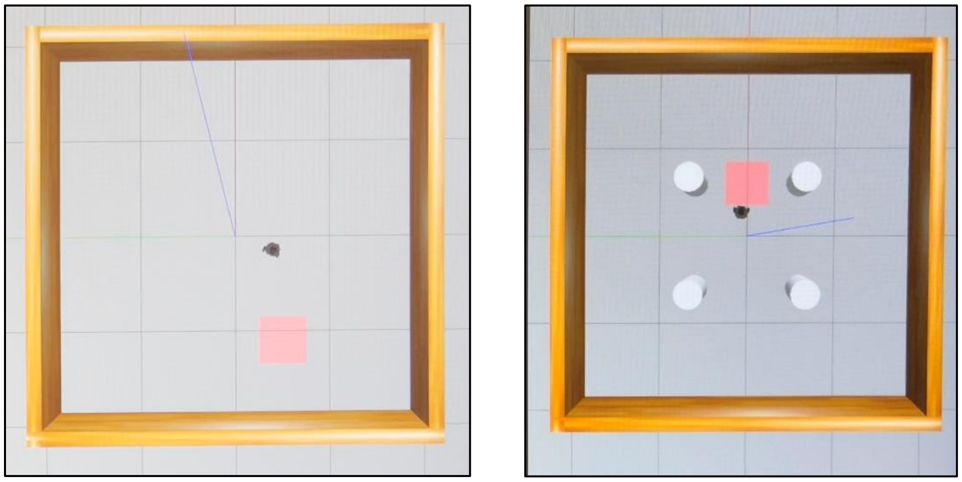


Figure 2.1: Gazebo Waypoint Tracking (left) and Obstacle Avoidance (right).

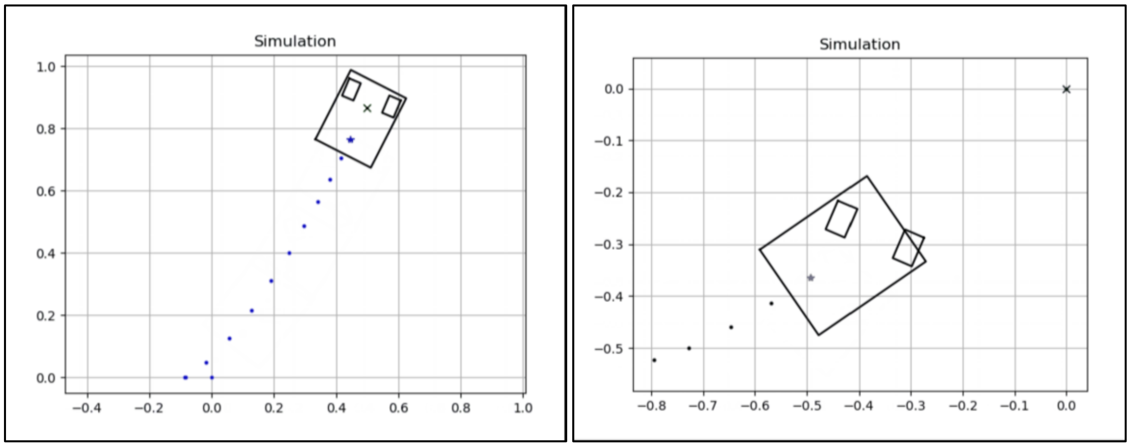


Figure 2.2: Dubins Random Waypoint Model (left) and Random Spawn Point Model (right).

Models can be trained for various tasks that do or do not involve avoiding obstacles shown in Figure 2.1. One task involves models training to navigate from a fixed starting location to a fixed end location. Additionally, models can be trained in environments where the starting location is randomized but the goal remains fixed, or vice versa where the spawn point is fixed and the goal is randomly placed. Both of these tasks can be seen above in Figure 2.2.

2.2 Evaluation

The evaluation process is much simpler since no policy updates are performed. While each training episode consists of the robot navigating to a single waypoint, evaluation can be performed with a series of waypoints. This is because a set of waypoints forming a path can be broken down where the robot thinks of each as an individual episode. Therefore, after the environment is initialized, Pickle is used to load the trained policy network data structure. Then, an array of waypoints are spawned. The robot continually takes actions until reaching all of the waypoints.

To determine its performance, I manually publish actions to the robot that steer it in straight lines to each waypoint. The reward received in this scenario is the maximum possible since no path will be shorter than this. Then, the reward received from the autonomous navigation can be compared to see if it is taking optimal routes. For simple routes, this can also be confirmed through visual inspection of the rendered environment. For models trained in the sparse environment, the dense reward function is used for evaluation. This is because the sparse reward function cannot effectively be used to measure performance. For example, a policy could be reaching the goals but in a very inefficient manner. When compared with an optimal policy, both would have sparse rewards of +1.

One thing to note is that for the discrete environment, sometimes there is no available action in the discretized action space that lead the robot in a straight line directly to the goal. Therefore, in these cases the robot is observed to oscillate between two actions that keep it near the straight line connecting the starting point and waypoint. The result is a slightly lower reward but still a near optimal route.

2.3 Deployment

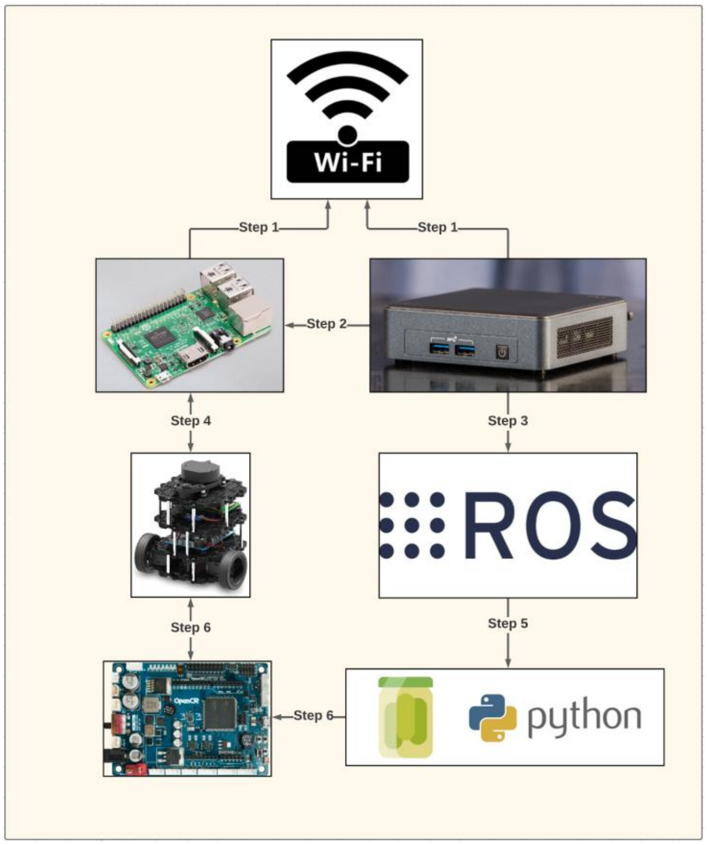


Figure 2.3: Deployment Process.

Once the RL policy has matured sufficiently during training, it is deployed onto the physical hardware of the robot and evaluated on similar tasks. The process is similar to evaluation in the simulator with a few extra steps. The full process is shown in Figure 2.3. In step 1, the remote PC and onboard Raspberry Pi is connected to the same WIFI network. In step 2, the ROS master URI is set on the remote PC and Raspberry Pi. Then, in step 3 a roscore instance is spun up on the remote PC so all ROS nodes can communicate with the master. Step 4 involves launching the turtlebot3_bringup package on the Raspberry Pi to initialize all ROS nodes to begin publishing and subscribing. Next, in step 5 a python file is run that is similar to the evaluation file but uses rospy to publish and subscribe to the ROS master interfacing with the physical robot. Finally, in step 6 actions are sent to the OpenCR board to control the motors. Using physical markings on the floor and observing the output of onboard sensors, the time and proximity to the waypoints achieved by the physical robot is compared to the simulated model for analysis.

3. RESULTS

3.1 Simulator

3.1.1 Gazebo and Obstacle Avoidance

I began by spawning the TurtleBot3 Burger model into a blank Gazebo world to test the communication between all of the ROS nodes. A script was written to steer the robot in a circle while its current state was output. The robot successfully completed multiple circles while returning to its exact starting location. Then, more complex environments were used to test the simulated LiDAR sensor. World files provided in ROS and Gazebo tutorials were used that mimicked a room with evenly spaced columns and an office setting with walls, desks, and doors. Figure 3.1 shows that the simulated LiDAR was measuring up to 3m away which is what the physical sensor is rated for.

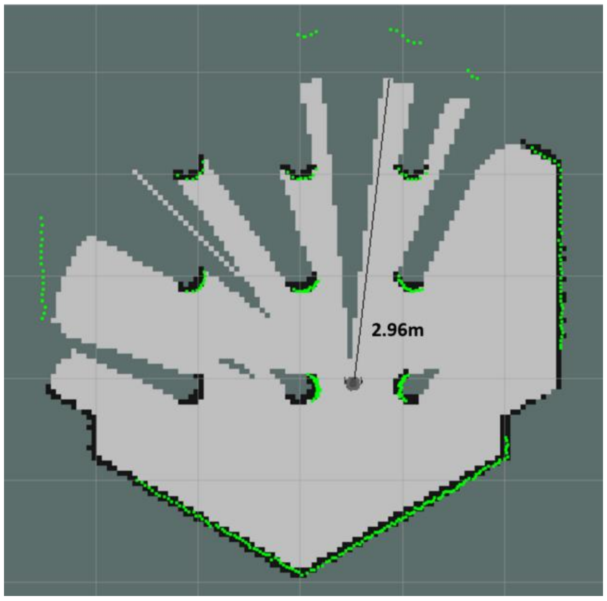


Figure 3.1: Validation of Simulated LiDAR.

Then, the ability for ROS nodes to send and save the LiDAR scans while navigating was tested. The robot was manually controlled throughout an office setting until the environment had been fully scanned. Then, the saved results of the scan were compared to the physical environment for confirmation that the robot had detected the location of each obstruction. As is shown in Figure 3.2, the scan very closely matches the physical environment from the robot's perspective.

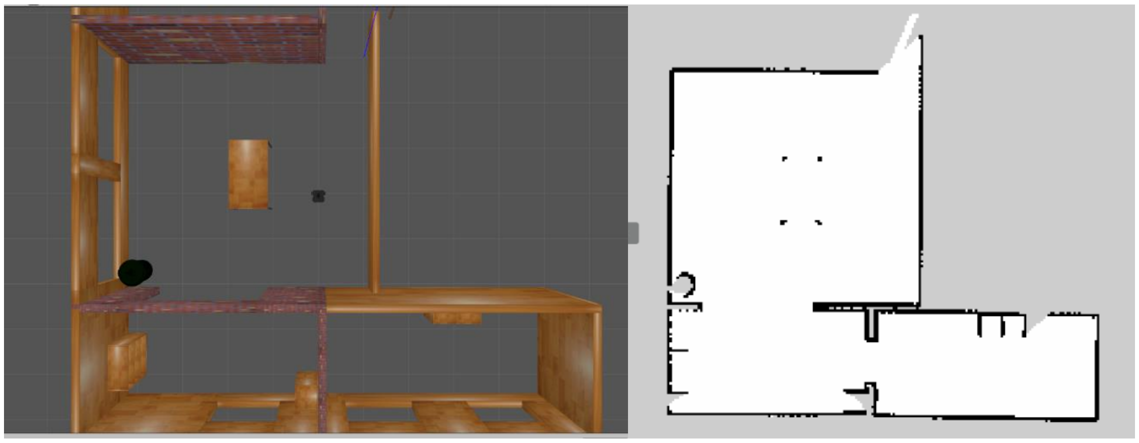


Figure 3.2: Physical Environment vs LiDAR Scan.

Next, a SLAM node was run to test the robot autonomously navigating a known environment with a pre-scanned map of the obstacles. Therefore, the LiDAR sensor data is overlayed on top of the saved map of the environment for localization of the robot. Then, waypoints were set for the robot to navigate to while remaining a certain distance away from all obstacles. An inflation radius was tuned to ensure the robot avoided all objects by a large margin. Below is a visual representation of what was observed through the simulator in Figure 3.3. While, the robot was able to reach most of the waypoints without a collision, this technique required traversing the environment prior to acquire a map and didn't handle dynamic conditions.

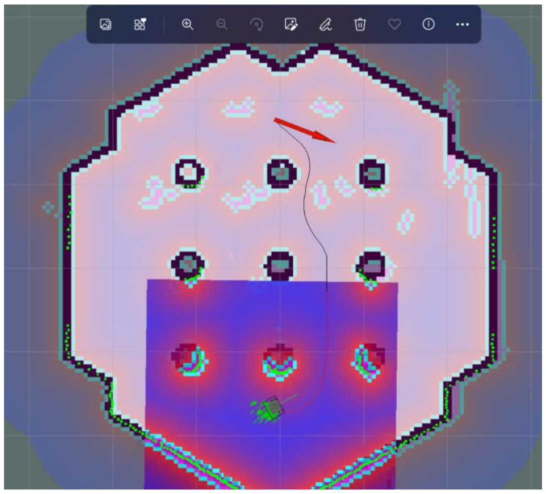


Figure 3.3: SLAM Navigation.

Therefore, DQN was implemented to learn how to traverse unknown environments that may contain obstacles. The number of LiDAR samples was reduced from 360 to 24 and the linear velocity was held constant at 0.15 m/s to promote exploration. A simpler reward function was used that gave +200 for reaching the goal, -200 for crashing, and penalized based on the L2 distance from current state to goal for other steps. It eventually converged to an optimal policy in around 250-300 iterations after 5 hours of training which can be seen by Figure 3.4.

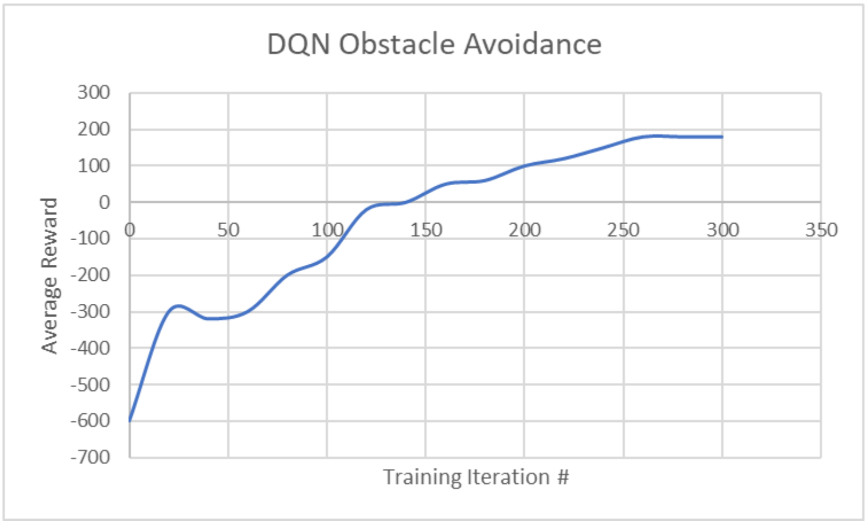


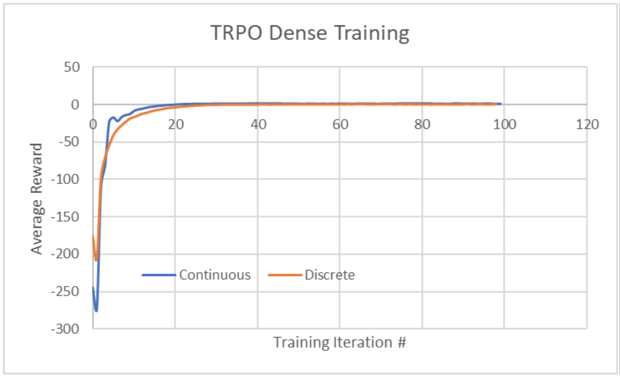
Figure 3.4: Average Reward of DQN while Training.

3.1.2 Custom Dubins

3.1.2.1 Trust Region Policy Optimization

Since the Gazebo simulator was taking extremely long times to train, I wanted to compare the performance of a lower fidelity simulator and a different algorithm. Therefore, I began training in the custom built Dubins simulator. TRPO provides guarantees for monotonic improvement and doesn't require tuning hyperparameters. This algorithm was implemented with control over both the linear and angular velocities, the neural network architecture from Figure 1.4, and the dense and sparse reward functions from Figure 1.5.

I conducted training with the fixed starting and ending point model. The waypoint was spawned at a fixed location 1m from the starting location of the robot. I began by comparing the performance of a continuous action space from 0 up to the robot's maximum linear and angular velocities and a discretized action space with 20 combinations to choose from. The maximum possible reward found by manually steering the robot in a perfect line for this environment is 2.53. As seen below in Figure 3.5, there are no major differences in performance or speed of convergence between the two action spaces during both training and evaluation. This was observed for all tasks, therefore, for the remaining analysis I only focus on the discretized version. Both models converge to an average reward of 2.1 which is considered optimal.



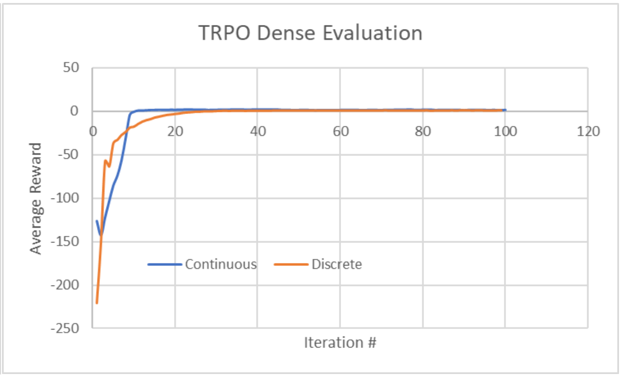


Figure 3.5: Average Reward of TRPO Fixed Start/End with Continuous and Discrete Actions.

Next, testing was done with the same fixed point environment but with the sparse reward function. During evaluation, the reward was computed with the dense reward function to gain insight into its efficiency in reaching goals. The results are shown below in Figure 3.6.

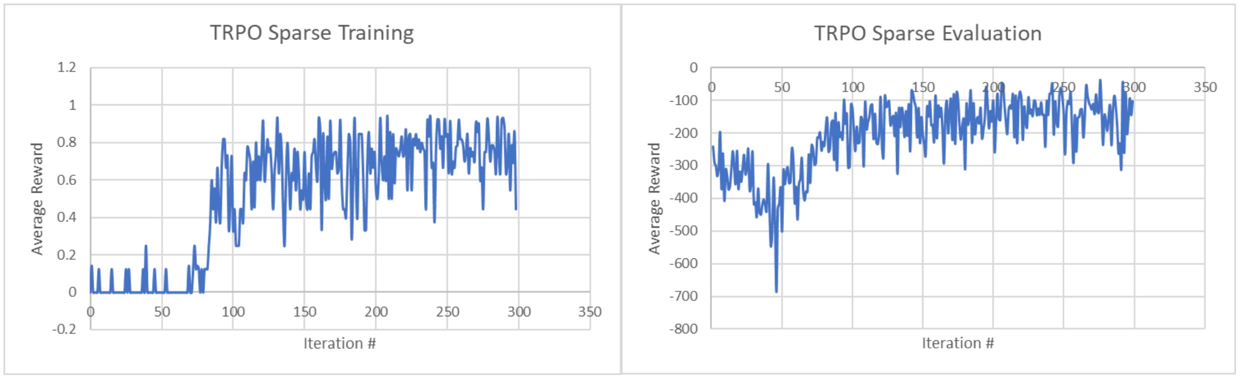


Figure 3.6: Average Reward of TRPO with Sparse Reward.

After 100 iterations it began reaching the goal more than half of the time. However, even after 300 training iterations the goal was only reached 80% of the time. Furthermore, by looking at the evaluation, which converges to -130, it is obvious that the robot did not take optimal paths to reach the goals. By viewing the rendered environment from the custom Dubins simulator, the robot was aimlessly wandering in circles for a large majority of the time in the first 60 iterations. Even beyond that iteration, there was a lot of circling in the direction of the goal exhibited by the robot.

To help TRPO learn more information, I changed the environment to spawn the robot directly facing the goal only for this trial. Previously, the robot was being spawned facing 90 degrees away from the goal. This made a significant impact on the training performance of TRPO since it was able to reach more goals and learn information quicker. The new results are shown below in Figure 3.7.

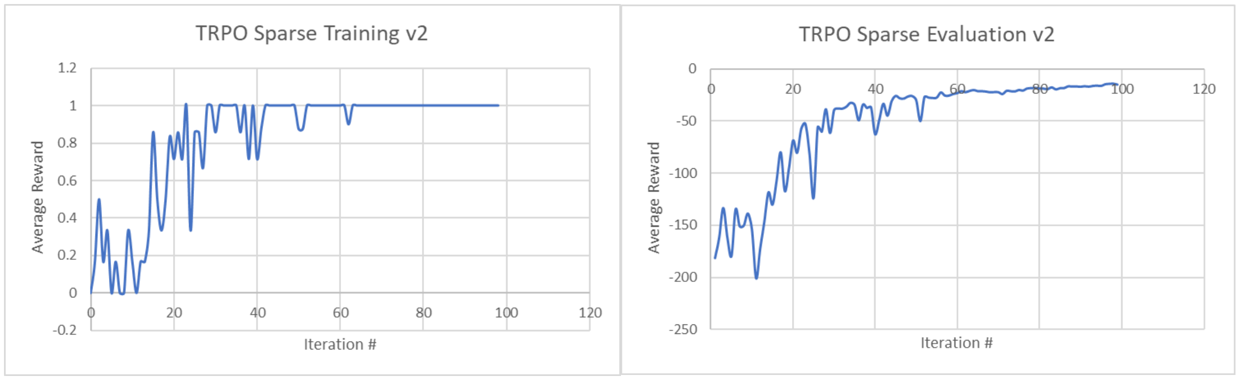


Figure 3.7: Average Reward of TRPO with Sparse Reward and New Spawn.

With this help, TRPO is able to reach the goal 100% of the time after just 62 iterations. Additionally, the evaluation converges to an average reward of -15. This is still not a perfect path but is considerably better than the previous model. However, it is important to note that it is not desired for such little changes to make such a drastic impact.

This is a common problem with TRPO and numerous other reinforcement learning algorithms in sparse environments. Since no reward is given at each time step, it is difficult to estimate the advantage any action has over the other. Therefore, the policy can spend large amounts of time not learning anything meaningful leading to poor behavior. This led to the implementation of LOGO.

3.1.2.2 Learning Online with Guidance Offline

LOGO uses two steps. The first is a policy improvement step that is identical to TRPO in order to generate candidate policies that lie within the trust region. The second step is a policy guidance step. In this step the policy that is closest to the guiding behavior is chosen. The trust region can be dynamically adjusted so that the newly trained policy does not exactly imitate the guidance policy, but rather, improves upon it. Thus, the guidance policy can be sub-optimal.

While the main focus of LOGO is to improve performance in sparse environments it can also be used in dense environments. For the first experiment, the optimal policy from the TRPO dense environment was used as guidance. Then, a sub-optimal policy was acquired by running TRPO for only five iterations on the same environment and also used as guidance in a second experiment. The results are shown below in Figure 3.8.

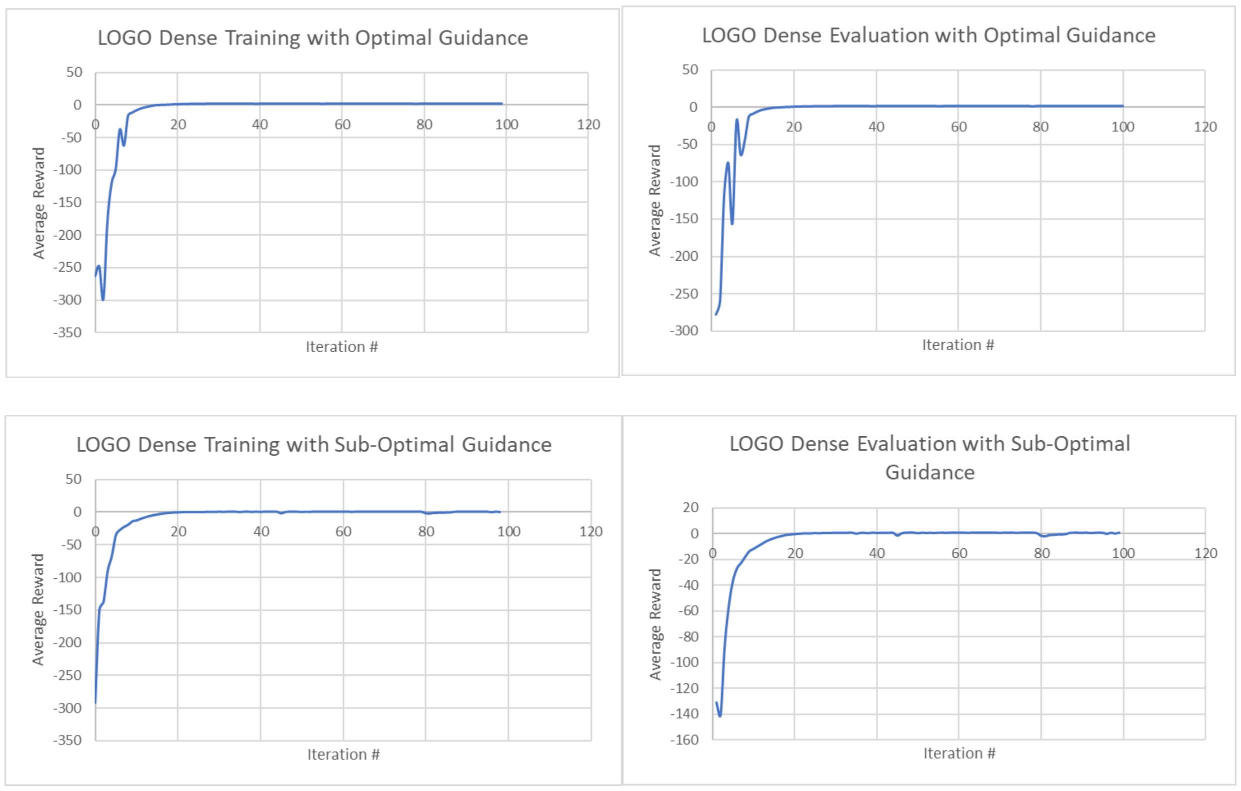
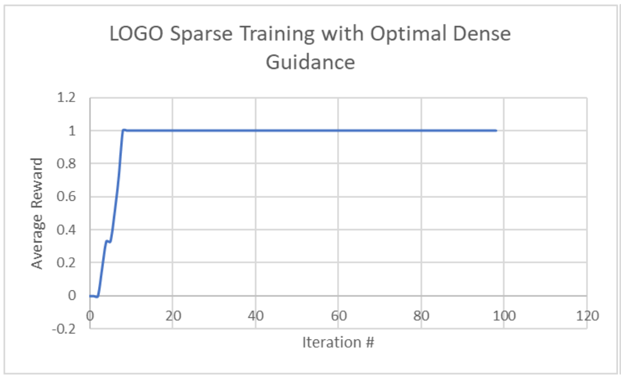


Figure 3.8: Average Reward of LOGO with Dense Reward with Optimal (top) and Sub-Optimal (bottom) Guidance.

In both scenarios an optimal policy was generated. While this occurs a few iterations sooner than just with TRPO, this was expected behavior since TRPO was able to reach an optimal policy in the same environment, the second step of LOGO wasn't offering too much of an advantage.

For the following experiment, I trained LOGO in a sparse environment but with the optimal policy generated from TRPO in the dense environment for guidance. The results are shown below in Figure 3.9.



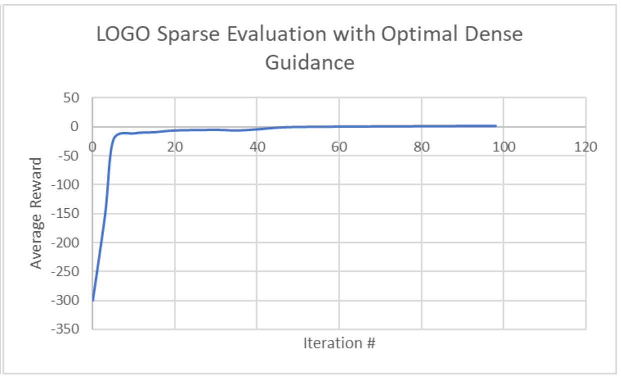


Figure 3.9: Average Reward of LOGO with Sparse Reward with Optimal Dense Guidance.

This generated a policy that reached 100% of the goals in just 10 iterations. During evaluation, it performed optimally similar to the dense environments. Even with the modified spawn location to help TRPO learn, it took 60 iterations to converge in the sparse structure and produced a smaller reward. Therefore, LOGO outperformed TRPO in this scenario in speed and quality of actions produced.

However, in sparse environments, it may not always be possible to attain an optimal policy or one generated using a dense reward function. Therefore, a final test was performed comparing three scenarios of increasing difficulty. First, LOGO was trained in a sparse environment with the sub-optimal policy obtained from training TRPO in a dense environment for just five iterations. This mimics scenarios where it might be possible to collect a few samples of data using a dense reward structure, or some sort of desired behavior or state-action pairs may be known ahead of time that can be used to guide the model. Additionally, LOGO was trained in a sparse environment with guidance from the modified TRPO sparse environment. This models scenarios where the problem may be simplified or made easier to gain some initial knowledge on the direction to head in. Finally, LOGO was trained in a sparse environment with the unmodified TRPO sparse environment policy used as guidance. This is for scenarios where the problem or environment cannot be edited to gain any new information. The results are shown below in Figure 3.10.

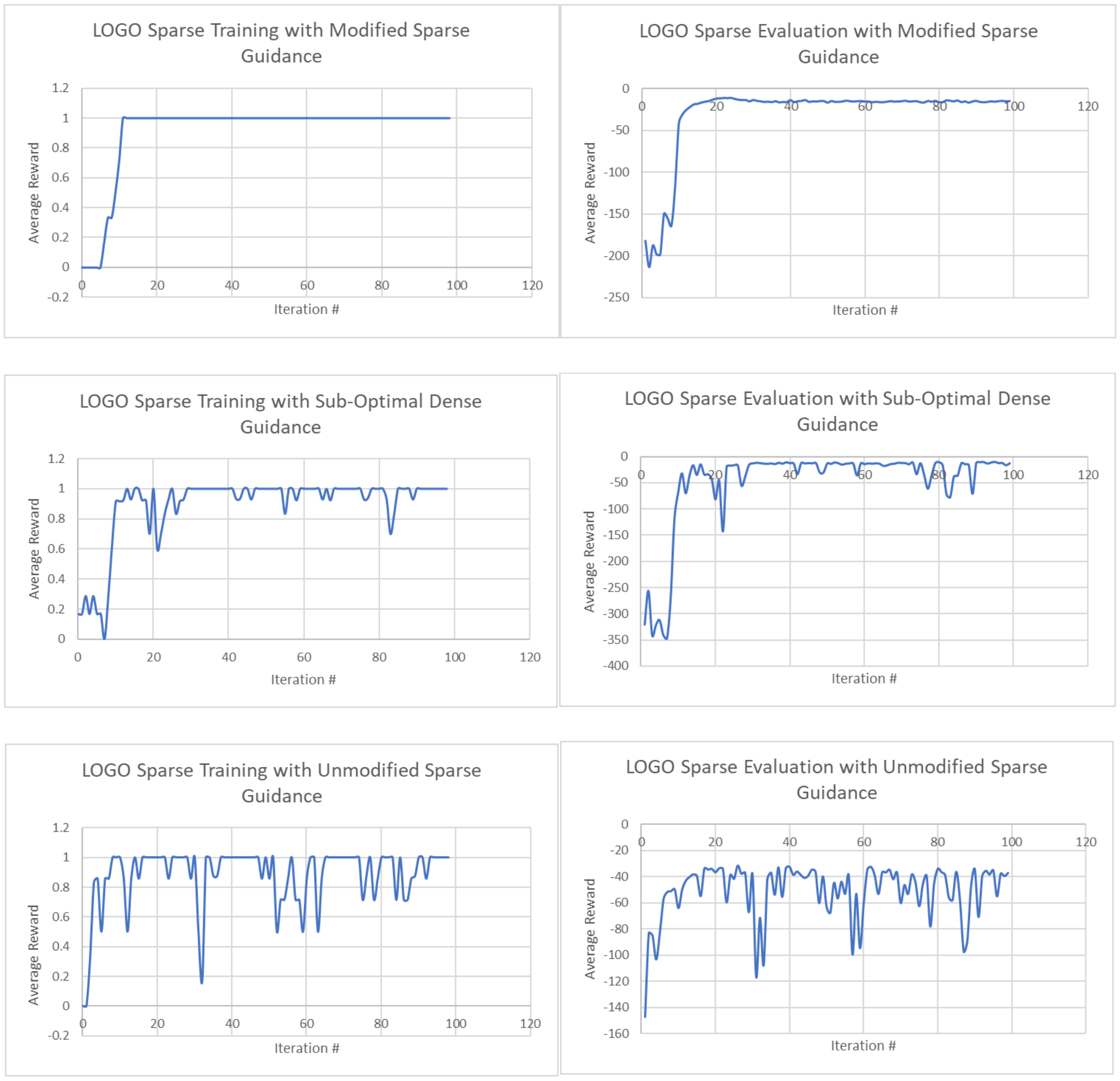


Figure 3.10: Average Reward of LOGO with Sparse Reward Increasing in Difficulty from Top to Bottom.

As was expected, the performance degraded as the problem became increasingly difficult. However, it is important to note that LOGO still outperformed its TRPO counterpart in every scenario. Even for the sparse environment where nothing can be altered to gain new information, LOGO is able to reach to goal more frequently and with a final reward of about -40 compared to -130 using TRPO.

3.2 Real World

3.2.1 DQN and Obstacle Avoidance

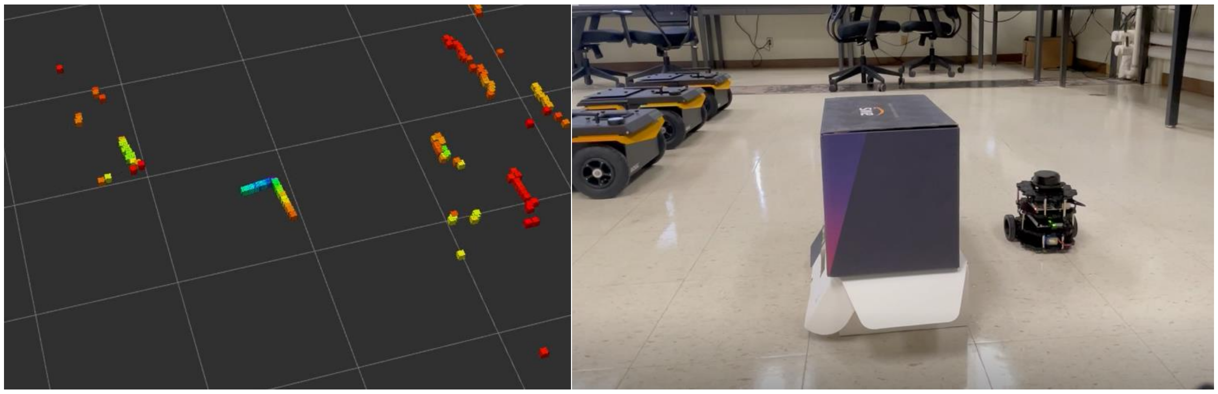


Figure 3.11: Physical LiDAR Scan vs Actual Environment.

The DQN policy was deployed onto the physical robot and instructed to reach a waypoint approximately 1m away located on the opposite side of a stationary box. The physical setup and LiDAR view is shown in Figure 3.11. The process was repeated ten times and every time the robot reached the correct destination. It took 14.3s on average and reached within .189m of the goal. It is important to note that in the simulator, all goals are given a threshold of 0.05m to train very accurate models. In the real world, the goal is to reach within a 0.2m threshold of every goal so that the performance drop is not too large in the real world.

3.2.2 Dense Environment

Once deploying the TRPO and LOGO models where actions controlled the linear and angular velocities the robot began exhibiting erratic and incorrect behavior. It was discovered that the TurtleBot3 Burger being used could not reach its maximum linear and angular velocities simultaneously. In fact, it was noticed that there were many combinations of linear angular velocities that when sent to the robot would cause incorrect behavior. This went unnoticed with the DQN model since the linear velocity was held constant and all turns were gradual. Therefore, the set of discrete actions had to be modified in order for the robot to be able to execute each one. The TurtleBot is reported to be able to reach 0.22 m/s linearly and 2.84 rad/s angularly. However, what was observed, shown in Figure 3.12, was 0.20 m/s and 2.30 rad/s. It has not been determined whether this limitation is present in all TurtleBot3 Burger robots.

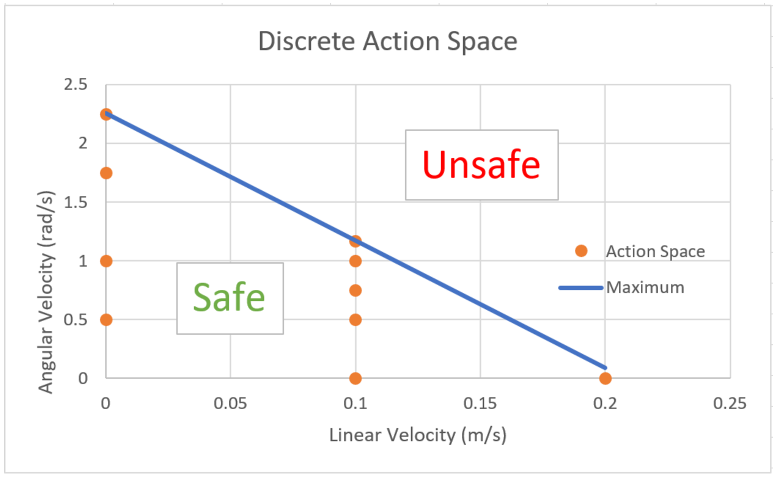


Figure 3.12: Modified Safe Discrete Action Space.

There was an additional change due to operating in the real world. In the simulator, the robot is not held to the same linear velocity limitation in order to speed up training. During training, the robot was able to reach 1.5 m/s. However, as was discussed, the physical robot has a much slower maximum speed. Since the robot reaches a negative reward at every step where it has not reached the goal, the slower it moves, the smaller the reward will be. Therefore, the maximum possible reward achievable is smaller in the real world and also depends largely on the distance between each sequential waypoint.

First, the dense TRPO policy from Figure 3.5 was deployed. The path it took relative to the actual waypoints is shown below in Figure 3.13. The path length is 5.325m and it took an average of 54s to complete. It reached to within an average threshold of 0.077m of all goals.

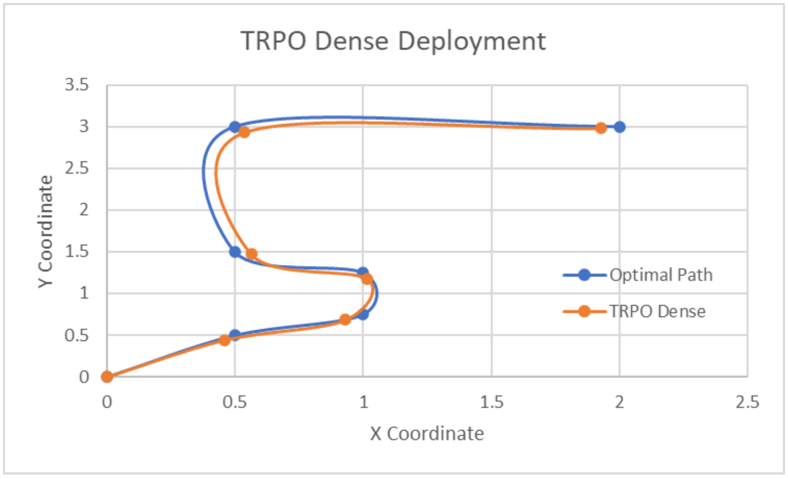


Figure 3.13: Dense TRPO Policy Deployment.

Next, the modified sparse TRPO policy from Figure 3.7 was deployed. Its path is shown below in Figure 3.14. It took just 36.5s to complete on average but only reached within 0.166m of each goal on average. Additionally, it only reached within 0.215m of one goal which does not meet the desired threshold for deployment.

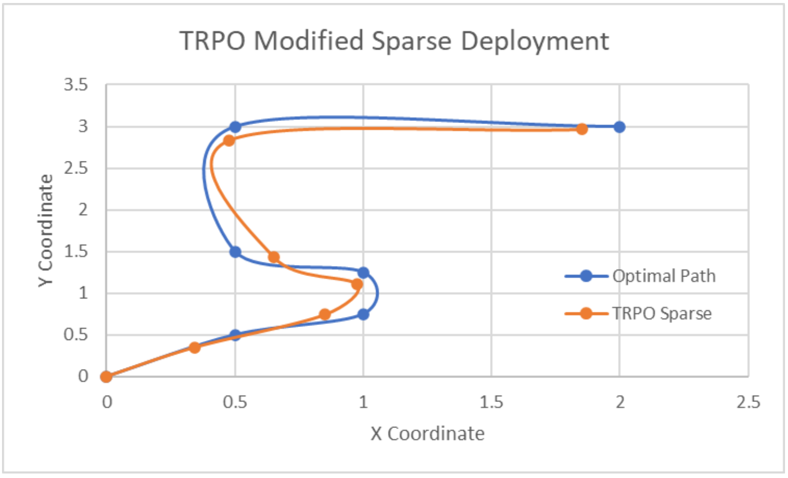


Figure 3.14: Dense TRPO Policy Deployment.

Then, the sparse LOGO policy with optimal dense guidance from Figure 3.9 was deployed. Its path is shown below in Figure 3.15. It took an average time of 57.5s but its performance closely matched the optimal performance from the dense environment. It had an average threshold of 0.091m.

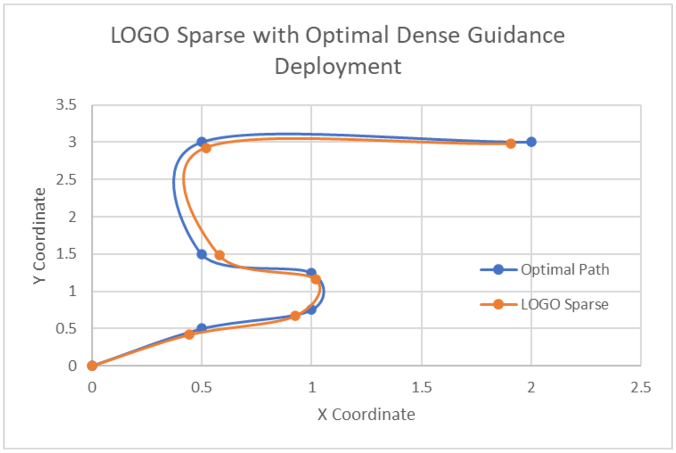


Figure 3.15: Sparse LOGO with Optimal Dense Guidance Deployment.

Finally, the final two polices were taken from the first and third scenario described in Figure 3.10. The sparse LOGO policy with modified sparse guidance is shown in Figure 3.16. It had an average time of 35s and threshold of 0.135m. The sparse LOGO policy with unmodified sparse guidance is shown in Figure 3.17. It had an average time of 62.5s and threshold of 0.182m. Therefore, it barely met the desired performance during deployment. A summary of the real-world results is shown in Table 3.1.

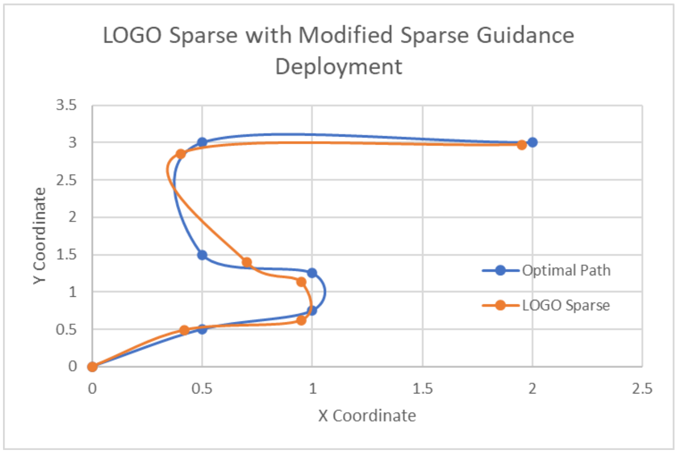


Figure 3.16: Sparse LOGO with Modified Sparse Guidance Deployment.

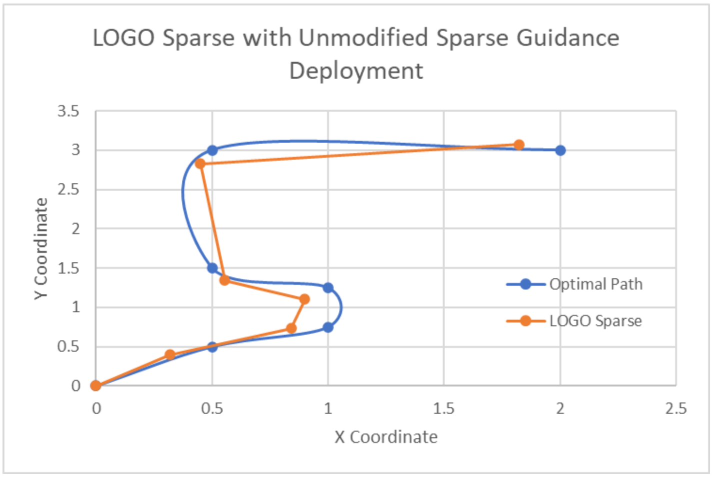


Figure 3.17: Sparse LOGO with Unmodified Sparse Guidance Deployment.

Table 3.1: Summary of Real-World Results.

	TRPO Dense	TRPO Modified Sparse	LOGO Sparse with Optimal Dense Guidance	LOGO Sparse with Modified Sparse Guidance	LOGO Sparse with Unmodified Sparse Guidance
Time (s)	54	36.5	57.5	35	62.5
Average Threshold (m)	0.07709	0.16629	0.09090	0.13535	0.18166

4. CONCLUSION

Overall, I can draw many conclusions from the results of this project. First of all, deep reinforcement learning can definitely be applied to autonomous navigation. Incorporating multiple algorithms allows for all types of problems and environments to be solved. Next, it was determined that low fidelity simulators can suffice for certain navigation tasks and even perform quite well. The dense model only experienced a sim2real drop in performance from a 0.05m threshold to 0.077m. Therefore, a low fidelity simulator can save hours of time even when the goal is real-world deployment. Finally, it can be concluded that LOGO is far superior to TRPO in sparse environments and performs similarly or better in dense environments as well. This is promising since it expands the types of problems these techniques can be applied to.

In the future I would like to explore unknown environments that also contain moving obstacles rather than static. Additionally, I would like to test LOGO with problems where censored state information is used for guidance and observe what kind of policy improvements to expect. This covers additional real-world scenarios where information can be gathered on some of the state variables but not all.

REFERENCES

[1] Cheng Y. & Wang G.Y. (2018) Mobile Robot Navigation Based on Lidar, *The 30th Chinese Control and Decision Conference*.

[2] Khaksar, W., Vivekananthen, S., Saharia K.S.M., Yousefi M. & Ismail F.B. (2015) A Review on Mobile Robots Motion Path Planning in Unknown Environments, *IEEE International Symposium on Robotics and Intelligent Sensors*.

[3] Rengarajan, D., Vaidya, G., Sarvesh, A., Kalathil, D. & Shakkottai, S. (2022) Reinforcement Learning with Sparse Rewards using Guidance from Offline Demonstration, *The 10th International Conference on Learning Representations.*