UPPSALA UNIVERSITY

PROJECT REPORT

Collaborative Fenceless Robotics Using Turtlebot (TMNT)

Authors:
Alexandros PATSANIS
Arnthór Helgi SVERRISSON
Bengu ERENLER
Ibrahim Abdirahman HERSI
Ina BOFF
Samuel Gebre YOHANNES
Santhosh Kumar NAGARAJAN
Dasiga Venkata SANDEEP

January 23, 2019

Abstract

Fenceless zone, a space in which robots can perform any task without meddling the work of humans. Safety is one of the most critical aspects in human-robot paradigm. Unlike caging the robots behind a protective grill, the new epoch of Fenceless robots are equipped with sensors, computer vision and programmed self-control etiquettes bringing the fences down in production facilities and enabling humans and robots join hands to work safely side-by-side. In this project, we used Deep Q-Learning python implementation for robot's collaborative behavior, ROS (an open-source operating system for Turtlebot 2i) and Gazebo (a robot simulator to rapidly train AI system using realistic scenarios and test algorithms).

Acknowledgements

First of all, we as a team would like to thank Uppsala University and Ericsson AB for providing us with the opportunity to work on an industry grade project.

Since the start of the project, we adopted clear communication with each other which enabled better teamwork. We have used project planning methodologies and tools such as Trello and Scrum. We used Trello to allocate cards for the different tasks and assign the members of the team who is interested in working on that specific task. Along with that, we also had daily Scrum meetings to get an update of each member of the team which enabled us to plan the project accordingly.

Our project course co-ordinator Stefanos Kaxiras and the course assistants Amendra Shrestha and Xiuming Liu have supported in everything from providing the support from the university, allocating the room which represented work like atmosphere to providing coffee.

We would be remiss to not to mention our supervisor of the project and representative from Ericsson, Konstantinos Vandikas, who has given much support regarding the project. He was always ready to take time off to come down to university or chat over Skype. He has guided us since the beginning of the project and even helped us to understand certain topics which we were not familiar with. We would like to thank him for that.

Contents

1 Introduction **5**
- 1.1 Background . . . 5
- 1.2 Project CS and Ericsson . . . 5
- 1.3 Project Goals . . . 5

2 Project Workflow **6**

3 Core Framework **6**
- 3.1 Acronyms . . . 6
- 3.2 ROS . . . 7
  - 3.2.1 Control the Turtlebot . . . 8
  - 3.2.2 Roscore . . . 8
  - 3.2.3 Catkin Package . . . 8
  - 3.2.4 Creating package . . . 8
  - 3.2.5 Roslaunch . . . 9
  - 3.2.6 Rostopic . . . 9
  - 3.2.7 ROS Kinetic Kame . . . 9

4 Product Description **10**
- 4.1 Main components . . . 10
- 4.2 Navigation . . . 10

5 Simulation Environment **11**
- 5.1 Gazebo . . . 11
- 5.2 V-Rep . . . 11

6 Reinforcement Learning **11**
- 6.1 Q-learning . . . 11
- 6.2 Deep Q-learning . . . 13

7 Related Work **13**

8 System Architecture **14**
- 8.1 Hardware architecture . . . 14
- 8.2 Software architecture . . . 15

9 Machine Learning Implementation **15**
- 9.1 Parameters and general information . . . 15
- 9.2 Q-learning Implementation . . . 16
  - 9.2.1 Environment Created . . . 17
  - 9.2.2 States . . . 17
  - 9.2.3 Actions . . . 18
  - 9.2.4 Rewards . . . 19

9.2.5 Training . . . 19
9.3 Deep Q-Learning Implementation . . . 20
9.3.1 Simulation environment . . . 20
9.3.2 State space . . . 21
9.3.3 Action Space . . . 22
9.3.4 Reward Function . . . 23
9.3.5 Activation Function . . . 24
9.3.6 ReLu activation function . . . 25
9.3.7 Advantages of ReLu . . . 26
9.3.8 Disadvantages of ReLu . . . 26
9.3.9 Machine Learning Tools . . . 26

10 Real World Environment **29**
10.1 Environment Mapping . . . 29
10.2 Moving From Gazebo To Real World . . . 30
10.3 Augmented Reality . . . 30

11 Container Based Deployment **31**

12 Conclusions **32**
12.1 Challenges . . . 32
12.1.1 Gazebo Simulation . . . 32
12.1.2 Cloud . . . 33
12.2 Results . . . 33

13 Future Work **35**

1 Introduction

1.1 Background

Robots can take different sizes and depend on the type of collaboration, force, and movement used; they could compromise safety in the environment. Industrial robots generally need to work inside cages so that human workers can be safe. Fenceless robotic systems were invented to circumvent this limitation, enabling the usage of safety protocols and sensors to work alongside humans. One of the benefits of fenceless robotics is that the company using such robots do not need to spend money and time making the environment safe by setting fences or buying expensive equipment. Also without fences, humans have more freedom to interact with robots. Other positive aspects are the reduced space in the working environment or factory, and when production needs change, it is easier to relocate and reconfigure robots.

1.2 Project CS and Ericsson

The project developed and presented in this report is conducted by students taking the course Project CS offered by Uppsala University. The theme of the project and work process was set and conducted in close cooperation with an industry partner, in this case, Ericsson. The course was an also opportunity for students to gain experience in team collaboration and the usage of modern agile project methodologies for software development, such as Scrum.[1].

This year's project was conducted in cooperation with and funded by Ericsson and Uppsala University. Ericsson is one of the leading providers of Information and Communication Technology (ICT) to service providers, and it offers technology ranging across Networks, Digital Services, Managed Services, and Emerging Business.

1.3 Project Goals

The project aimed to "teach" an Interbotix Turtlebot2i [2] running Robot Operating System (ROS) how to behave, while doing some task, when approached by humans or when they near obstacles, by dynamically reacting to the environment thus minimizing risks of collision.

The main approach involves:

- Devising a reinforcement learning model for training a robot on how to behave while it tries to achieve a goal.
- Training the robot in a simulation environment that mimics real-world conditions and objects.
- Transfer the knowledge gained from training in simulation to the physical robot, so it can dynamically react to conditions in the real environment i.e. behave autonomously.

- Develop an augmented reality Android application that visualizes the safety zone around humans interacting with the robot.

2 Project Workflow

During our project, we made use of Agile software development. The workflow of the project was divided into Sprints, with one sprint spanning approximately two weeks. The three main parts of one sprint were development, testing, and documentation.

As new objectives arose, they were added to a project-specific Trello board [3]. There were no expressed limitations on how much work could be in progress at any given time except for the limitations specified for the sprints. As the content of each sprint was decided, the workload of the sprint was implicitly decided. There were no defined roles within the project, but members of the team took turns leading and managing each sprint, i.e. scrum masters.

Tasks were assigned to each member of the team during our daily sprint meetings. During these meetings, each person discussed their progress and challenges from the day before and their plans for that day. It was also during these meetings that tasks were assigned to each individual. Each task and the person/people assigned to it were kept track of using Trello. This helped us keep visualize tasks that were complete, pending or in progress and the person/people handling them. Work that was not complete from the previous sprint was carried over to the next sprint as backlog.

Regarding version control, we made use of Github and Ericsson provided the official repository for this project [4].

3 Core Framework

This section explains the details of the core framework that runs our project on the robot.

3.1 Acronyms

- NUC (Next Unit of Computing): The barebone kits consist of the motherboard in a plastic case with a fan which has a built-in CPU, sold by Intel, an external power supply, and a mounting plate [5].
- OpenAI Gym: Toolkit for developing and comparing reinforcement learning algorithms. [6]
- ROS: Robot Operating System [7].
- Roscore: The main process that manages all of the ROS systems. In order for ROS nodes to communicate, It is a must to have a score running.

- Rostopic: A channel where nodes can either read information and write them. Topics handle information between messages. Different types of messages are available.
- Subscriber: Reads information from a topic.
- Publisher: Publishes information on a topic.
- Ros message: Message files that can be found inside a msg directory of a package.
- Catkin Package: Package contains launch files (launch folder), source files (cpp, python, src folder), list of cmake rules for compilation (CMakeLists.txt) and package information and dependencies(package.xml).
- Roslaunch: Through roslaunch command, it is possible to launch multiple ROS nodes locally and remotely via SSH, and to set parameters.
- Ros Kinetic Kame: A ROS distribution is a versioned set of ROS packages. [8]

3.2 ROS

The robot operating system was originally designed by Stanford artificial intelligence laboratory. ROS provides services designed for the heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message passing between processes and package management. ROS consists of a collection of tools and libraries that were created with the intent of simplifying the task of creating robust robot behavior across many different robotic platforms.

ROS is written in C++ and Python and runs on Linux.

Software in ROS is separated into:

- ROS client library implementation such as roscpp,rospy,roslip
- Language and platform independent tools used for building and distributing ROS based software.
- Packages containing application - related code which uses one or more ROS client libraries.

ROS package application includes:

- Motion tracking
- Control
- Planning
- Face recognition

- Gesture recognition
- Object detection
- Perception
- Mobile robotics

3.2.1 Control the Turtlebot

To move the robot, we have used some commands in the beginning:
q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

3.2.2 Roscore

The roscore is the main process that manages all of the ROS systems. In order for ROS nodes to communicate, it is a must to have roscore running. Roscore automatically starts up a ROS Master, a ROS Parameter Server, and a rosout logging node.

3.2.3 Catkin Package

To organize its programs ROS uses packages. Package contains launch files (launch folder), source files (cpp, python, src folder), list of cmake rules for compilation (CMakeLists.txt) and package information and dependencies(package.xml). Example of the catkin package in the project:

```
~/turtlebot2i/src/turtlebot2i_gazebo$ ls
CMakeLists.txt  launch  package.xml  worlds
```

Every ROS program that created and executed will have to be organized in a package. Packages are the main organization system of ROS programs.

ROS usually works with catkin workspace which, is designed to allow for better distribution of packages and cross-compiling support and better portability. It is the official build system of ROS, rosbuild. Rosbuild were created by early days of ROS, and it has been evolved all the years along to support the needs of the ROS eventually, this led to suboptimal design decisions and complexity. So, a new foundation of catkin created because of these problems.

3.2.4 Creating package

The catkin workspace helps not only building multiple packages at once but also building packages in the correct order. Catkin workspace automatically finds, uses and includes other packages. When creating a new package, firstly making a catkin workspace; below the commands.

```
mkdir -p ~/your_workspace_name/src
cd ~/ your_workspace_name/src
catkin_init_workspace
catkin_make
source devel/setup.bash
```

3.2.5 Roslaunch

Through roslaunch command, it is possible to launch multiple ROS nodes locally and remotely via SSH and to set parameters. Through it, it is also possible to automatically respawn processes that are already dead. Roslaunch is XML configuration files that have the .launch extension which specify the parameters used, the nodes that will be launched, and the machines that they should be run on. One roslaunch file can launch other roslaunch files. From the script below; There are arguments in the launch file, finding and using the world file we have created and getting the scan topic and loading the URDF int to the ROS Parameter Server.

Roscore is a specialization of the roslaunch tool for bringing up the "core" ROS system.

3.2.6 Rostopic

Topics are a channel where nodes can either read information and write them. Topics handle information between messages. Different types of messages are available. Messages are defined in the .msg file; it can be found inside a msg directory of a package. Below provided command is to get information about a message.

```
Rosmsg show <message>
```

Subscriber
Simply a subscriber is a node which, reads information from a topic. The subscriber node each time reads something then it calls a function which, a print of the message.

Publisher
This is a node which writes information to a specified topic.

3.2.7 ROS Kinetic Kame

A ROS distribution is a versioned set of ROS packages. Its purpose is to let developers work against a relatively stable codebase until they are ready to roll everything forward.

The distribution of ROS that we worked on is ROS Kinetic Kame, which is mainly targeted at the Ubuntu 16.04 (Xenial) release.

4 Product Description

The project focus on a mobile robot that looks like Roomba vacuum cleaners [9]. The robot moves autonomously towards one or more destinations and while simultaneously avoiding obstacles so that humans can interact safely with it. The robot can as well have its tasks analyzed by humans through an augmented reality app which shows the safety zones surrounding the humans.

4.1 Main components

- Interbotix Turtlebot 2i, the robot.
- Intel RealSense camera, used mainly for working with the robot's arm. Not used in the scope of this project.
- Lidar sensor, gives a 360° laser scan of environment.
- Orbbec Astra Cam which is used for detecting the environment and mapping. Also used as a laser scanner instead of the Lidar.
- Accelerometer/Gyro/Compass for odometry detection, where odometry is the linear and angular position values of the robot in the environment.
- Edge Detection and Bumper Sensors, used only if the camera doesn't detect the obstacles.
- Computers for training the robot, having Intel Core i7 processors, 16GB Ram, Ubuntu 16.04 Operating systems.

4.2 Navigation

The Turtlebot uses AMCL (Adaptive Monte Carlo localization) algorithm for localization [10]. AMCL is a probabilistic localization system for moving a robot. It implements the adaptive Monte Carlo localization approach which uses a particle filter to track the pose of a robot against a known map.

AMCL takes in a laser-based map, laser scans, and transform messages and output pose estimates on start up, AMCL initializes its particle filter according to the parameters provides.

The turtlebot_navigation package takes an AMCL launch file and a map file as parameters on start up.

5 Simulation Environment

5.1 Gazebo

Gazebo is a 3D dynamic simulator that precisely simulates the behavior of robots in complex indoor and outdoor environments with real physics. Gazebo simulations offer a high degree of accuracy, multiple physics engines, a customizable library of models and environments, a suite of sensors and programmatic and graphical interfaces to design robots, test algorithms and train Artificial Intelligence using realistic scenarios [11].

Gazebo supports best on Linux systems that have:

- A CPU, Intel i5 and later or equivalent
- 500 MB of free disk space
- Ubuntu Trusty or later installed

5.2 V-Rep

V-Rep is another robot simulation that "is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution [12].

At the beginning of the project, we explored using both Gazebo and V-Rep to decide which simulator would suite us best and we decided to use Gazebo since it has more support for ROS integration using Python.

6 Reinforcement Learning

To solve the task in this project we used reinforcement learning techniques.

6.1 Q-learning

Q-Learning algorithm is one of the Reinforcement Learning methods. Q-Learning uses the concepts of agents, states, actions, and rewards. Q-Learning learns from a value of being in a given state and taking a specific action there. As seen in figure 2, Q-Learning has a table which, called Q-Table, it consists of values for every state as a row and every possible action as a column in the environment. First, the table fills up with zero then, updates the table according to rewards that observed from actions. Q-table serves a purpose to find the optimal action for each state. Q-Learning can be summarized in figure 1.

Agent: An Agent takes actions; Turtlebot2i finding the goal, avoiding the obstacles, remains

safety zones. The Agent tries to maximize cumulative reward.

Action: Affects the environment, based on the Q-values the agent chooses an action in the current state.

State: A description of the current situation in the environment.

Reward: A quality measure of the environment's state.

1. Initialize Q-values ($Q(s,a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s,\cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s,a) := Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$

Figure 1: Pseudo code of the Q-Learning

	Action 1	Action 2
State 1	0	10
State 2	10	0
State 3	0	10

Figure 2: Q table

Like in figure 3, NewQ(s, a) which, is an expected future reward of that action a, at the state s, can be estimated with the Bellman equation updates to Q(s, a).

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward **given the new s' and all possible actions at that new state**

Figure 3: The Bellman equation

6.2 Deep Q-learning

Deep Reinforcement Learning popularly referred to as Deep Q- Learning uses Deep Q Neural network (NN) that will approximate, given a state, the different Q-values for each available action (i.e.) Deep Q-Learning uses NN to approximate the Q-function.

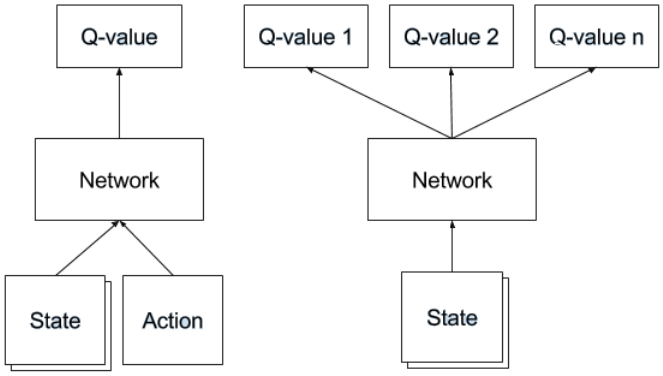

Figure 4: Q- Learning vs Deep Q-Learning architecture

Q-Learning suits for a limited state space/action space. Let's consider robot walking in which each frame is treated as a single state; the action space could have millions of states or unknown depending on the environment. To overcome the above-stated situation the Deep Q-Learning makes uses a neural network as a function approximator to calculate the Q-values.

At Deep Q-Learning training, the network inputs a state and produces the Q-values for every action in the action space. At prediction, the trained model is used to predict the next effective action to take in the environment.

Effective Action = Max (NN predicted Q-values)

7 Related Work

Generally, problems in robotics are tackled by reinforcement learning, which enables a robot to find an optimal behavior autonomously by having trial-and-error interactions with the environment.

There are many related works in the literature which implement different reinforcement learning techniques, each using different simulation environments. In [13], for example, Q-Learning is implemented in V-Rep simulator to make a robot learn how to find a goal, using a Q-table. However, Q-Learning is better used for environments where a small set of states are needed to be retrieved. Since our task is to make the robot move in a real environment, the state size is large, having real values. Creating and updating a Q-table for that environment would not be

efficient. Such Q table does not scale well in this case, so neural networks are preferred since it instead approximates Q-values for each action based on a state.

Instead of Q-Learning, Lei and Ming [14], implement deep Q-Learning in Gazebo to make a Turtlebot robot move in different corridor environments using depth information from an RGB-D sensor. However, it does not train using dynamic obstacles.

Lillelund [15], proposes the use of modern game engines with highly realistic rendering for simulating robots. The training uses image data and deep Q-Learning in Unity3D to make a Turtlebot 2 robot find and drive into a blue ball. However, the study is more useful for problems that deal with self-driving cars, agricultural weed detection or grasping objects, once these tasks need realistic graphics that many robot simulators cannot render.

One interesting work from Open Robotics [16], addresses the scenario where a turtlebot 3 is trained using a DQN model to move while avoiding both static and dynamic obstacles. The simulation environment used is Gazebo, however instead of an infrared sensor like Orbbec, they use a LIDAR sensor.

8 System Architecture

8.1 Hardware architecture

Most of the work (setting the simulation environments and training) was done in the 64-bit computers at the University labs comprised of Intel® Core™ i7-7700 CPUs @ 3.60GHz × 8 with 15,5 GiB RAM memory, dedicated GeForce GT 730/PCIe/SSE2 graphics card and 250,9 GB disk storage.

The second phase was to train the model in a cloud environment which consists of Ubuntu 16.04.5 LTS, 16 cores and 31 Gigabytes of memory.

The NUC of the turtlebot was by default running a 32 bit version of Ubuntu, so an upgrade had to be made to the 64 bit version of Ubuntu in order to run tensorflow which is officially supported on 64 bit systems only.

The robot came with an Intel RealSense camera, which uses infrared depth related scans for detecting the environment and for mapping. It also came with a Lidar sensor that is a laser scanner (more precise than Kinect or Orbbec infrared cameras), which was not used in the scope of this project. Orbbec Astra Cam was used only during the mapping of the physical environment.

Other Turtlebot 2i components that are integrated are the Accelerometer/Gyro/Compass for odometry detection, sensors for edge detection and safety so that the robot stops moving when

approaching an obstacle.

8.2 Software architecture

At the beginning the OS installed was Ubuntu 18.04.1 LTS Desktop version, however we had to downgrade it to 16.04 once we discovered that the version of ROS that we would be working on, Kinetic, only supported Wily (Ubuntu 15.10), Xenial (Ubuntu 16.04) or Jessie (Debian 8) for Debian packages.

9 Machine Learning Implementation

9.1 Parameters and general information

Both Q-learning and Deep Q-learning implementations had some common hyperparameters with values:

Hyper Parameter name	**Description**
learning_rate	Learning speed. It is too small so it can learn slow and accurate.
discount_factor	How much future events lose their value according to how far away.
epsilon	The probability of choosing a random action.
epsilon_decay	The reduction rate of epsilon.
EPISODES	Number of episodes.
episode_step	Time step of one episode.
epsilon_min	Minimum epsilon value.
batch_size	Size of the group of training samples.
train_start	If the replay memory size is greater than 64, start training.
target_update	Update rate of the target network.
memory	Size of the replay memory.

Other parameters used:

Parameters	**Description**
target_update	Update rate of the target network.
load_episode	Number of the loaded episode from the previous saved.
load_model	Boolean value that indicates if the saved model will be used.
state_space_size	Number of states.
num_of_actions	Number of actions.
parameter_dictionary	Contains the epsilon value to be saved in the json file.
model	The model used for updating the target_model. (Its weighs are copied)
target_model	The main model updated rarely.

The num_of_actions are used by the model to define the last layer's number of nodes, and the state_space_size are used by the model to define the first layer's number of nodes.

This algorithm makes use of a specified number of episodes defined in the variable EPISODES. For each episode, the robot can take only a limited number of steps, defined in episode_step. Inside each step, q values for each of the 5 actions are predicted by the model based on the current state values and an action index for the biggest q value is chosen. The state values are the scan_range, direction of movement, the current distance of the robot to the goal, distance to the closest obstacle, and angle to the closest obstacle.

After the action is chosen, the robot makes a small move (step), and the next_state and reward are calculated and it checks if it is done for the episode (crashed, found the goal or reached the number of episode steps). Then, the state, action, reward, next_state, and the done status are appended in the memory array to serve as previous experiences, which will be used later in training.

Every given number of episodes, the training saves in an md5 and a JSON file the model configuration and the epsilon values respectively, so to avoid the training being lost in case of an interruption. The md5 files store information about the current episode's layers of the model, like the weights and biases. If the load_model variable is set to True, the code will load the configurations of the saved episode which number is defined in load_episode. That way, the training can start from that point instead of starting over.

The epsilon value is used in order to choose how random action will be chosen, while the decay is used to decrease the chances of getting random actions by discounting it from the epsilon value in each episode. Getting random actions is useful at the beginning of the training since the robot needs to experiment with unknown actions. Once it has good knowledge, it does not need to try much and can start performing better by using its acquired knowledge.

The target model is updated (weighs are copied from the model) when the number of global steps reaches the number defined in target_update inside an episode and also every time a robot completes an episode (when it is done). This is done for stability since it keeps the target function idle for a while once its weights are kept frozen most of the time.

9.2 Q-learning Implementation

To start off simple we wanted the robot to have two objectives:

- Avoid obstacles
- Go from point A to point B on a map

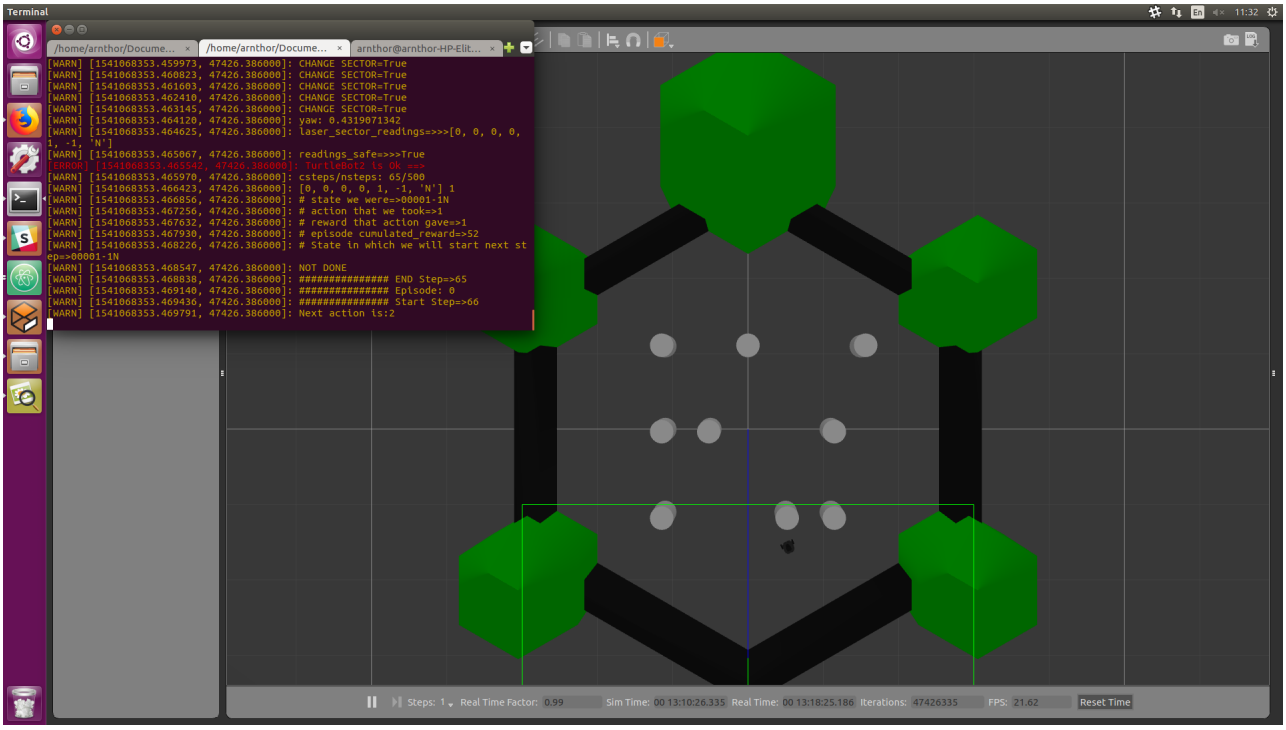

Figure 5: Q-Learning in gazebo simulation

In order to solve these problems, the states and reward function needs to be designed to with that in mind.

9.2.1 Environment Created

The simulation was done in Gazebo. The environment was an enclosed maze or room with nine obstacles. The robot starts on one side of the room, and the objective is to get to the other side of the room without crashing into anything. The Turtlebot was equipped with a Lidar laser scanner which gives distance to obstacles around the robot. It is also equipped with accelerometer and gyroscope, so it is possible to track its movements and know its coordinates and orientation in the environment (odometry).

9.2.2 States

The first part of the problem is to avoid obstacles. So laser scan values from a Lidar sensor was added to the states. Five values (from 5 separate angles) were sampled. The values were evenly distributed. These values were then discretized to make the state space simpler. So each of these five laser scan values are given a new discrete value. If the laser scan value is:

- 0m - 0.2m it gets a new value of 3
- 0.2m - 0.3m it gets a new value of 2

- 0.3m - 0.4m it gets a new value of 1
- Higher than 0.4m it gets a new value of 0

For simplicity, the objective for the robot was to get from point A to point B in only one dimension (y-axis), or to the other side of the room (it does not care about the position in x-axis). That is why when the position is added to the state, only the position in the y-axis is added. The y-position is also rounded to the nearest integer to make the state space smaller.

The orientation is the last variable added to the state. The orientation is transformed into either 'N', 'W', 'E' or 'S' (north, west, east and south) based on what direction the robot is facing. The orientation is provided in the odometry ROS topic.

So the state will be:
[laserscan_1, laserscan_2, laserscan_3, laserscan_4, laserscan_5, y-position, orientation]
This array is then turned into a string. For example, if the state array is [0, 0, 0, 0, 1, -1, 'N'] it is turned into the string '00001-1N' which represents the state the robot is in, and this value is stored in the Q-table (as a row). So the Q-table can, for instance, give us the Q-value of taking action 0 (forwards) when the robot is in state '00001-1N'.

Having seven different attributes for representing the state and each attribute can take 4-5 different values means that the state space is quite big. The five laser scan attributes can have four different (discrete) values, and the orientation also has 4 different values (N', 'W', 'E' and 'S'). In our environment, the y-position attribute can have five different values. This means that there are $4^6 \cdot 5 = 20.480$ different possible states.

9.2.3 Actions

The robot has 4 actions:

- Forwards:
  - Linear speed: 0.35m/s
  - Angular speed: 0 rad/s
- Backwards:
  - Linear speed: -0.35m/s
  - Angular speed: 0 rad/s
- Left:
  - Linear speed: 0.05m/s
  - Angular speed: 0.6 rad/s

- Right:
  - Linear speed: 0.05m/s
  - Angular speed: -0.6 rad/s

The robot will try to learn what action is best for each state it is in.

9.2.4 Rewards

The rewards we give the robot are:

- -200 for crashing
- 200 for finding the goal
- -150 for completing maximum amount of steps (500 in our case)
- 20 for moving to a 'higher' y-position (this means the robot is moving closer to the goal)
- -20 for moving to a 'lower' y-position (this means the robot is moving away from the goal)
- 5 for facing and moving in a 'correct' direction. This means that the robot is facing incorrect direction ('north' in our case) and is moving forward
- -5 for facing and moving in the wrong direction. This means the robot is facing in the wrong direction (south) and is moving forward, or is facing incorrect direction (north) and is moving backward. This is to create an incentive for the robot to the face the goal.

9.2.5 Training

We trained for 17.500 episodes. The reward graph below shows the robot finds the goal quite often and by looking at the simulation, the robot seems to sense the objects around it but has a hard time finding a way to avoid them and go around them. However, looking at the training graph, there is no trend for the rewards to become higher over time. So it does not seem the robot has managed to learn much and improve during these 17.500 episodes. As mentioned before this can be because the state space is enormous and that the robot might need more time training.

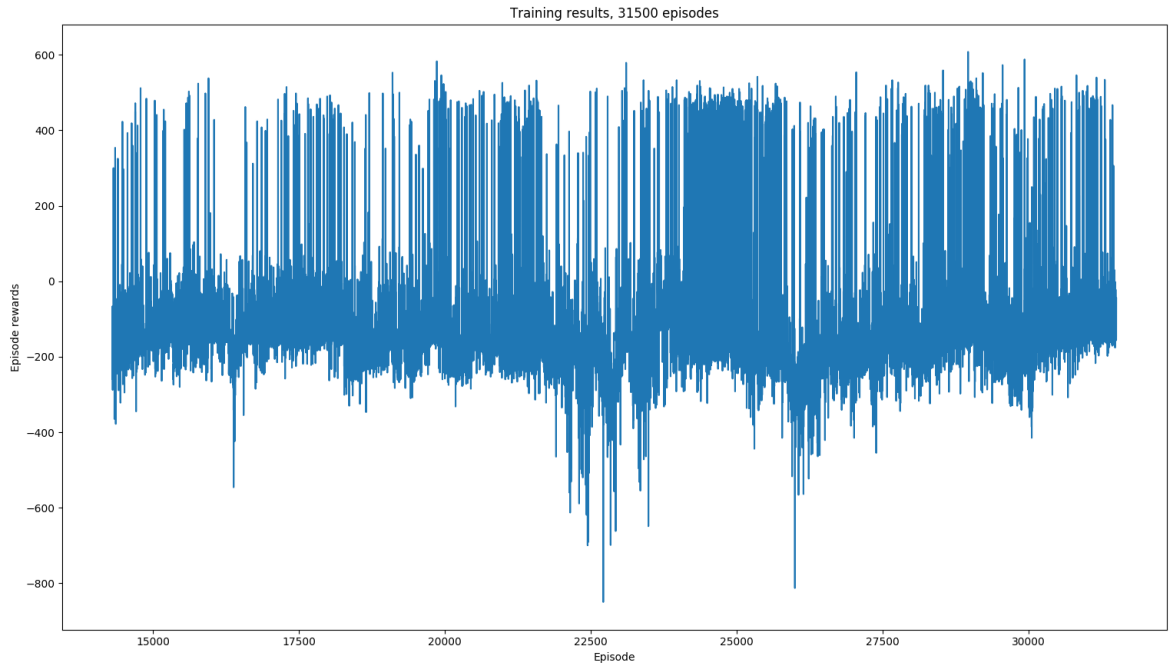

Figure 6: Q-Learning training graph

9.3 Deep Q-Learning Implementation

9.3.1 Simulation environment

The simulation was done in Gazebo. The simulation environment was an enclosed room with two humans (static) and four moving boxes. The Turtlebot 2i starts in the middle of the room, and a goal is generated at random somewhere in the room. The four obstacles move in circles around the middle where the Turtlebot starts. The aim is to reach the goal without crashing into the obstacles (moving boxes, humans or walls). If the robot finds the goal, a new one is generated someplace else. The episode is terminated either if the robot crashes or if it reaches the maximum amount of steps (500). If the episode is terminated, the simulation environment is reset.

Figure 7: Our gazebo simulation

9.3.2 State space

The Turtlebot Machine Learning Navigation Training (TMNT) uses 28 values as input state space. The first 24 values are the sampled scan readings from Orbbec Astra camera, and the last four input values are goal angle, goal distance, closest obstacle distance, and closest obstacle angle.

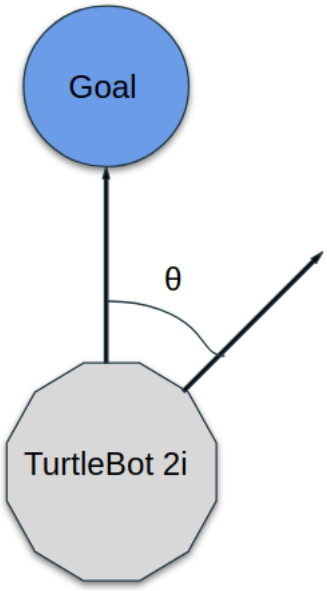

Figure 8: Robot's state space

9.3.3 Action Space

The Turtlebot 2i robot has five actions with constant linear speed (0.15 m/s) and varying angular speed for each action.

Actions	**ANGULAR SPEED (Radians / sec)**
0	-1.5 (LEFT)
1	-0.75
2	0
3	0.75
4	1.5 (RIGHT)

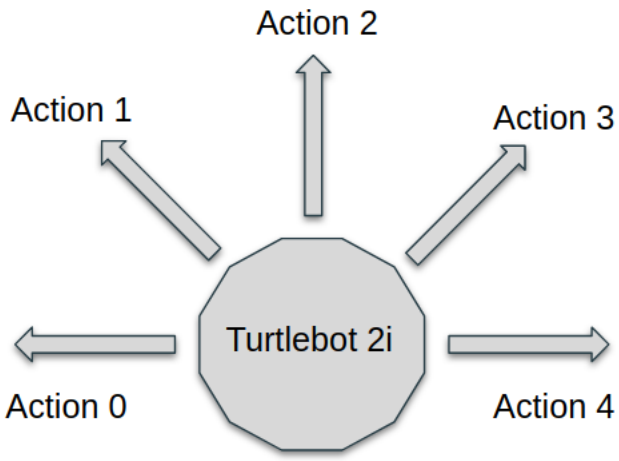

Figure 9: Robot's action space

9.3.4 Reward Function

The robot/agent learns from the consequences of its decision rather than being explicitly taught. The agent selects its actions based on its exploitation (past experiences) and exploration (new decisions) which is commonly referred to as trial and error learning. The agent receives reinforcement signals in terms of measurable reward, which depicts the success rate of an action or decision and the robot always seeks to select optimal actions to maximize the accumulated reward over time. The Turtlebot 2i implements a reward schema based on the below-provided guidelines

- On collision, the agent receives a reward of -150.
- On reaching goal, the agent receives a reward of 200.
- For each step, the agent receives a distance reward and an angular reward.
- Distance reward of $2^{Dc/Dg}$, where Dc is the current distance to goal and Dg is the initial distance to the goal (when the goal is set).
- An angular reward is based on the goal angle theta. If theta is 0 (the robot is facing towards the goal) the robot gets a maximum amount of angular reward, 5, and if theta is 180 degrees (the robot is facing the opposite direction of the goal) the robot gets the minimum amount of angular reward, -5.
- The distance reward and angular reward are then multiplied together.

The reason the distance reward decreases as the robot gets closer to goal is that if it was opposite (increases as the robot gets closer), the agent tends going close to the goal and then circling it to get extra rewards [17]. To encourage the robot to go straight to the goal instead we give it a decreasing distance reward as the robot gets closer to the goal.

The angular function, as mentioned before, has a range between -5 to 5 and is at the maximum when goal angle theta is 0 and minimum when the goal angle theta is in 180 degrees, so it behaves just like a cosine function or $5 \cdot cos(\theta)$. This is to encourage the robot to face towards the goal.

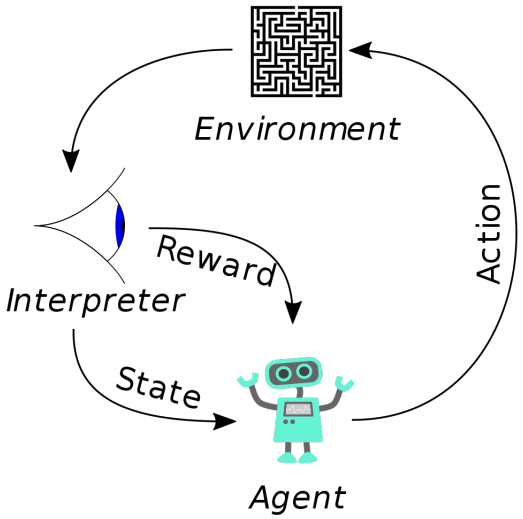

Figure 10: Reward pattern

9.3.5 Activation Function

The activation function [18] that we used is called Rectified Linear Unit (ReLu), a commonly used activation function for deep learning models. There are two main reasons for using this activation function at first place rather hyperbolic tangent activation or Sigmoid activation function. Firstly, ReLu is a non-linear activation function which makes to backpropagate the error easily and have multiple layers of neurons being activated by the ReLU function. Secondly, by using Rectified Linear unit we eliminated the vanish gradient problem and non-zero centered problem.

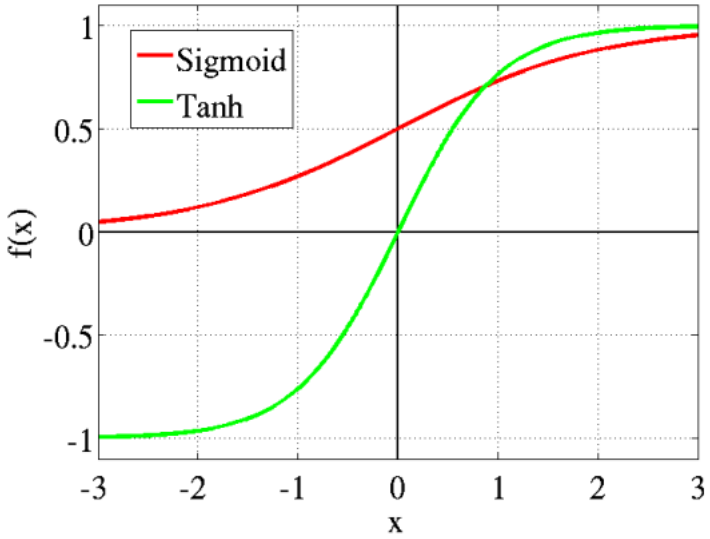

Figure 11: Sigmoid and Tanh activation function

9.3.6 ReLu activation function

ReLu is the most used activation function in a convolutional neural network or deep learning. The activation function f(z) is zero when Z is less than zero and f(z) equals to be Z, when Z appears to hold a value equal to or greater than zero. The other benefit of ReLUs is sparsity. Sparsity arises when $Z \leq 0$.

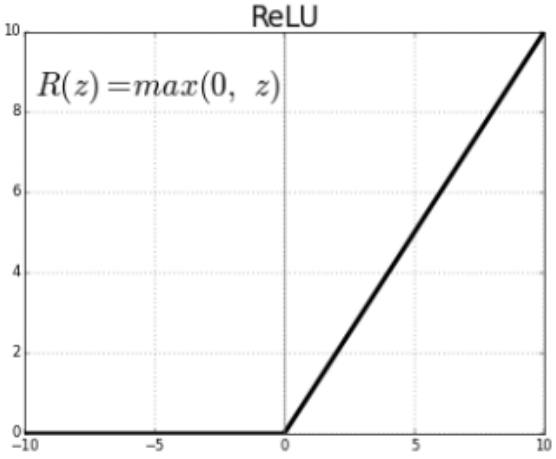

Figure 12: ReLu activation function

9.3.7 Advantages of ReLu

- Avoids vanishing gradient problem - It is a first problem occurs when training a neural network model using gradient-based optimization techniques, like hyperbolic tangent activation or sigmoid activation function. Both functions have a property in which the neuron's activation saturates either at zero or one. This means that the gradient in these regions is almost equal to zero and that has a drawback during the backpropagation. Imagine that the whole gradient will be multiplied with that specific neuron's gate, this means that if the gradient is significantly small, it will affect the information that passes through the neuron, in a way, that the information will be so tiny which will be like having no signal.
- Solves non-zero centered activation - It is a second problem occurs when sigmoid activation is used. This is unappealing since neurons in later layers of processing will receive data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights w will during backpropagation become either all be positive, or all negative. The phenomenon as mentioned above might introduce undesirable fluctuations in the gradient updates for the weights.
- Efficient Computation - ReLu uses a simplistic formula in computation, there is advanced activation which is complex compared like LeakyRelu and PRelu.
- Sparse activation - The more such units that exist in a layer, the more sparse the resulting representation. Sigmoids, on the other hand, are always likely to generate some non-zero value resulting in compact representations. Sparse representations seem to be more beneficial than compact representations.

9.3.8 Disadvantages of ReLu

- The presence of negative inputs never fires the ReLu activation function resulting in zero output, and as such, the gradient will always be 0. This behavior occludes the neural network from learning.

9.3.9 Machine Learning Tools

Keras - A powerful high-level tool for Neural Network API. Keras is written in python and allows to use it on the top of:

- Tensorflow - An open source deep learning library for numerical calculation using data flow charts. The graphical edges represent multidimensional data sequences (tensors) transmitted between them and the nodes in the graph represent mathematical operations. The flexible architecture allows distributing the calculation to one or more CPUs or GPUs on a desktop, server, or mobile device with a single API. TensorFlow was initially being developed by Google's Google Brain team to conduct machine learning and in-depth neural network research.

- Microsoft Cognitive Toolkit (CNTK) - An open source deep learning toolkit
- Theano - A python library which allows to optimize and evaluate mathematical expressions in which multi-dimensional arrays. With Theano it is possible to use GPUs and execute efficient symbolic differentiation.

Keras Sequential model is a linear stack of layers; Keras uses an essential building block called a sequential model for deep learning.

The Keras sequential model is building by calling the keras_model_sequential() function.

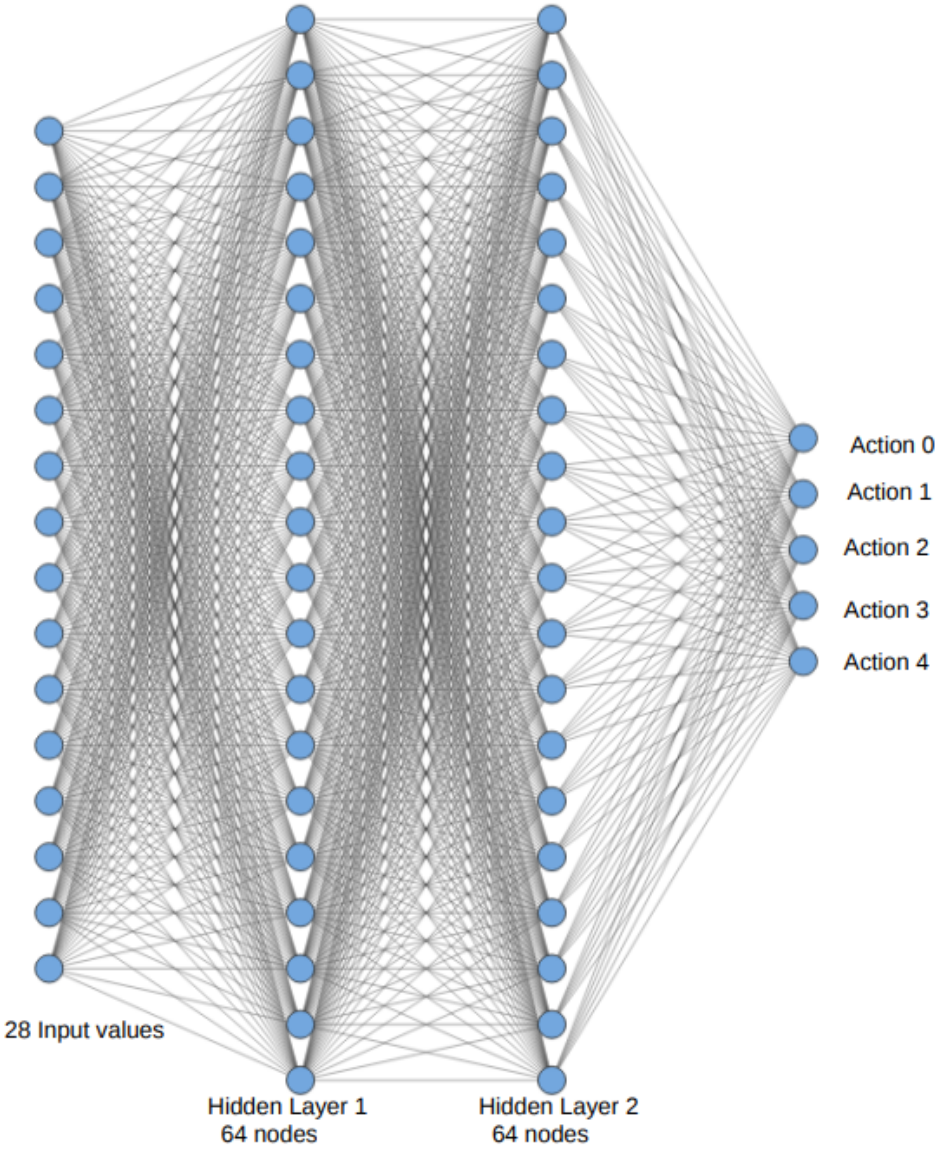

Figure 13: neural network structure

In order to build our model, we used Keras Sequential model. After setting the sequential model from Keras, the second thing that the models have to know is the input shape. For the above-cited reason, the first layer in the sequential model needs to receive the information about its input space. Then, the models will have an automatic way of shape inference for the next layers. We used a dense layer; the basic idea is that each neuron receives input from all preview layers, in this way the nodes are densely connected. The structure of a layer consists of a weight matrix, an activation, and a bias vector. The output is the activation function, which takes the dot of the

input and the weight matrix plus the bias. The following image is from Keras documentation and is the out that previously explained.

```
output = activation(dot(input, kernel) + bias)
```

There are many ways to set the input shape. In our case, it has been used 28 input values. These values are translated to one dimension; therefore, we have to use the comma after the variable self.state_size. We used this approach way in the case that we want to expand our input shape to multi-dimensions.

```
model.add(Dense(64, input_shape=(self.state_size,), activation='relu',
    kernel_initializer='lecun_uniform'))
```

10 Real World Environment

10.1 Environment Mapping

Mapping aims to allow the robot to scan the environment, build a map, identify the obstacles and frontiers in the map, and continue to explore those unknown areas until all the areas are fully explored and mapped.

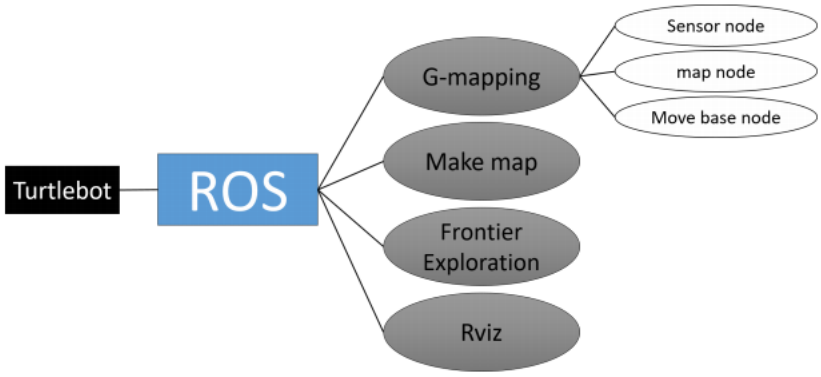

Figure 14: ROS mapping architecture

We used the TurtleBot robot with Kobuki base and Kinect camera. The system is running in the ROS Indigo distribution within Ubuntu 16.04 OS. The figure above shows the system architecture. In the make map node, an occupancy grid map is updated and published over time. The Figure below shows the occupancy map generated by the make map node.

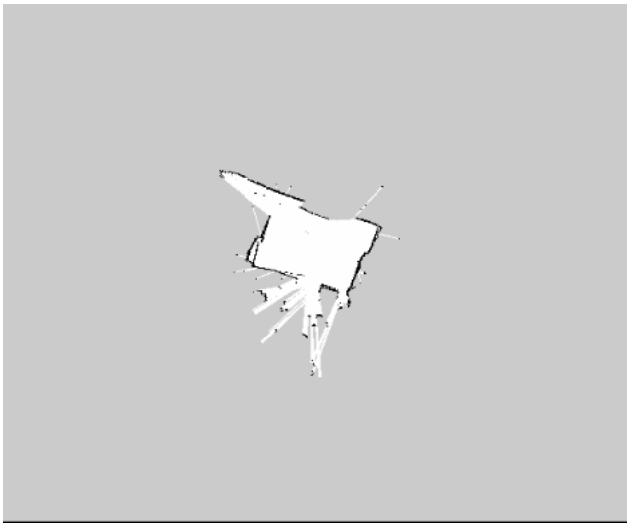

Figure 15: mapped floor plan

In the make map node, an occupancy grid map is updated and published over time. The figure above shows an occupancy map generated by the make map node. It subscribes to the topic scan to get the laser scan information. The scan here is translated from the depth camera data. The 3D sensor node, which brings up by the gmapping launch file, would handle that translation. In our project, we get the robot pose information by subscribing to odom (odometry) topic. The gmapping node also provides a transformation that transforms the robot pose into the map frame.

10.2 Moving From Gazebo To Real World

We were then able to move from the simulation environment in Gazebo to using the Turtlebot 2i in our office in Uppsala University. We made some changes first before moving to a real environment. In the simulation environment, we can reset the environment every time the episode terminates, and the robot suddenly appears in another position. This is certainly not possible in the real robot, so we made the episode never terminate, so the robot never resets into a new position. If the robot is about to crash into another object, it stops and turns until there is no object in front of it, and continues from that point on. This recovery technique is not always perfect, for example when dealing with moving obstacles, so we always had several people supervising and guarding the robot for safety.

10.3 Augmented Reality

For this project, we explored development of an augmented reality android application. The purpose of this application was to visualize "safety zones" around humans to reinforce trust between humans and the robot.

In order to achieve this we made use of ARCore framework which is an augmented reality development framework developed by Google [19] and Tensorflow YOLO [20]. ARCore enabled us build an augmented reality experience without having to solve complex problems like motion tracking, environment understanding and light estimation. Tensorflow YOLO (You Only Look Once) is a state-of-the-art, real-time object detection system and it gaveus the ability to perform human detection in real time.

ARCore takes the phone's camera data as input and detects interesting points called feature points and tracks how these points move over time (anchors). Using these points and the phone's sensors, it builds an understanding of the world interms of orientation and position. This helps it place objects in the real world seamlessly and maintain their position in the world regardless of how much the camera moves.

TensorflowYOLO takes the image rendered by ARCore and performs human detection on this image. Once the human has been detected, coordinates of a bounding box around the human can be accessed. Using these coordinates, we perform some arithmetic and get the coordinate at the bottom center of the bounding box which is a representation of a point between the human's legs (approximate).

Once we have the desired position where we want to place our visualization of the safety zone, we then render a 3D object that represents the safety zone at an anchor position that is closest to the desired position. We do this because an anchor is needed so that a position is trackable by ARCore in order for the object rendered not to float around but be placed realistically in the real world.

11 Container Based Deployment

The important hurdle TMNT team experienced managing the same work space installation among its peers. Running applications in containers instead of virtual machines is a momentum for the team. Docker containers were used to overcome the mentioned hurdle.

Docker allows to build, test and deploy applications quickly through the use of containers that keep everything the software needs to run including libraries, tools, code, and runtime. It is particularly versatile and useful for fast deployment and escalation of applications into any environment.The kappavita/projectcs18 docker image were build by Travis. It replicates every step of the deployment process [21].

As concerning the security aspect, Docker ensures that applications that are running on containers are completely segregated and isolated from each other, granting you complete control over traffic flow and management. No Docker container can look into processes running inside another container. From an architectural point of view, each container gets its own set of resources ranging from processing to network stacks.

The school provided us with a cloud where we were able to train for long peroids.

12 Conclusions

12.1 Challenges

During the training process of gazebo simulation, we observed some unusual behavior in the reward diagram. This is due to the fact that gazebo simulation stopped publishing laser scan values for some unknown reason. This was a significant challenge for our team since we did not know what the reason was and when would simulation crash. One potential approach that we thought was efficient is to reduce the lost training time, by saving the model for every 100 episodes. In this way, we could continue from the point where the simulation has crashed.

12.1.1 Gazebo Simulation

The following diagram illustrates an unusual behavior during training. This training was one of the most extended training that has lasted more than a week. As it can be seen for the below diagram after 5500 episodes, the gazebo simulation has stopped working. Therefore, the training process could not be adequately completed.

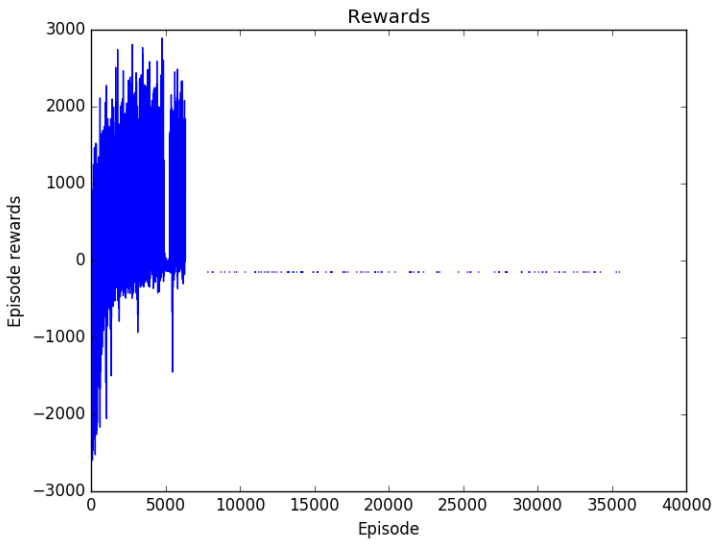

Figure 16: reward graph

Besides, there were some cases where the laser scan values were stopped publishing for a while and then the values started publishing again. The next diagram shows a similar case.

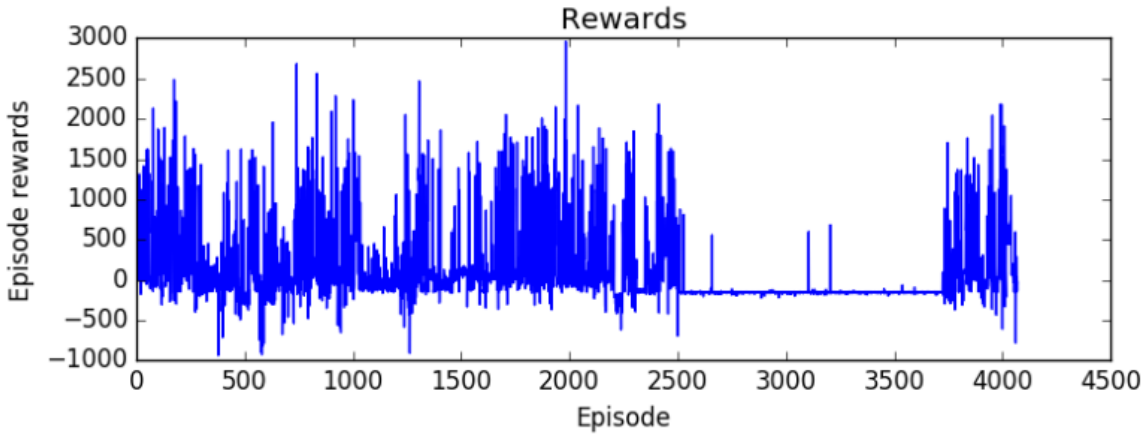

Figure 17: Reward graph

12.1.2 Cloud

Due to the fact that Gazebo simulator crashes and is a slower training process. We moved to the cloud, by using dockers to run the training process and we were able to observe the results without having access to the simulation window.

During the training process on the cloud, we had some crashes on the server as well. The error message was a broken pipe, which in general means that there was a network disconnect for a reason. Next, we modified the ssh connection time out to fix the issue but, still the same issue was appeared.

During the limitation of this thesis, we could not investigate more about this error but, is worth running it in a different cloud to observe if same problems appearing.

12.2 Results

We trained in Gazebo both on the cloud and our local machines for couple of days. We tried different activation functions and tuned some hyper parameters. Training with Leaky ReLu activation function, setting the epsilon decay as 0.999 and training for 3000 episodes proved to give the reward graph shown below. The only measurement we used was the reward graph and watching the simulation in Gazebo. From the reward graph (see figure 18) we can see that in the first 700-800 episodes the rewards are improving gradually. After that it seems it doesn't improve much overtime (converges).

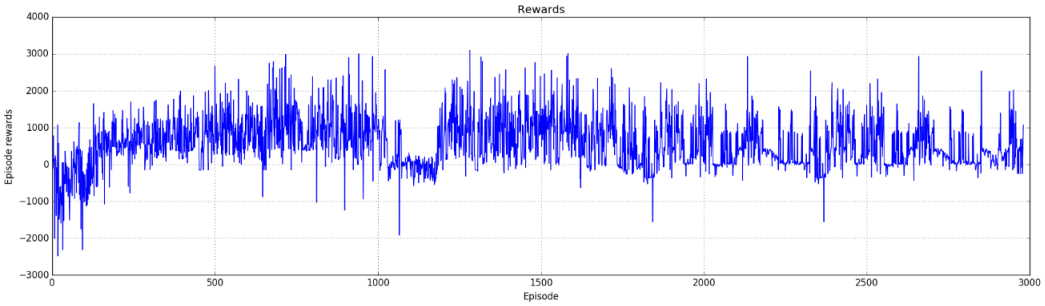

Figure 18: Reward graph

For comparison we added the also trained with ReLu (the rest of the hyper parameters were same) for 5000 episodes (see figure 19). This gave a similar result.

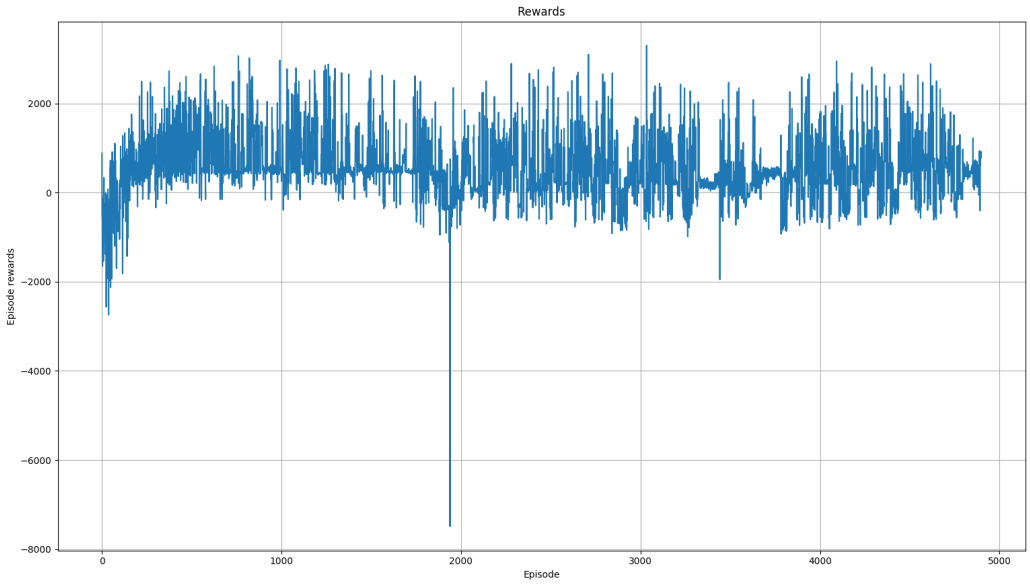

Figure 19: Reward graph 5000 episodes

13 Future Work

Due to the limitation of this project mainly regarding time, many improvements were left aside. Such improvements that could be worked on in future works include:

- Lidar
  Currently, our robot uses only the Orbbec camera to figure out and avoid obstacles. The values obtained from it are transformed into position coordinates and odometry angles for direction. However, this camera offers a limited angle of 60 degrees. Even though the training is not much affected by this limitation, it would be better to use a Lidar to get a broader range of values.

- Complex tasks
  Currently, the robot has a simple task of reaching a goal while avoiding obstacles. More complex tasks could be added in the future. Such tasks could involve the robot's arm and interaction with other robots. e.g., collecting a ball that was left by another robot, moving to another position and leaving the ball on the floor.

- Comparison study and optimization
  In this project, we suggest and implement a solution and train a model however a possible further work could be to do some further comparison study by comparing different solutions and/or optimize the hyperparameter settings.

- Prediction of dynamic obstacles
  Since we're training with dynamic obstacles it would be interesting and probably improve the model if we would be able to predict where the obstacles are heading and try to predict its position at time t+1. If we would be able to successfully do that prediction then we could train the model with that additional information and hopefully the robot will make smarter decision. To be able to predict that is however a complex task and comes with some limitations like there probably has to be conducted in a controlled environment and with fixed number of obstacles to track. There has been some previous studies done related to this [22].

- Integration with the acceleration group

We trained our robot in the lab's computers. A better approach would be to use special hardware for computation such as GPUs. Therefore our model could make usage of the solution devised by the Project in Computer Systems group which worked with acceleration.

References

[1] "Agile scrum methodology," https://www.softwaretestinghelp.com/agile-scrum-methodology-for-development-and-testing/, accessed: 2018-09-14.

[2] "Trossen robotics," https://www.trossenrobotics.com/interbotix-turtlebot-2i-mobile-ros-platform.aspx, accessed: 2018-12-01.

[3] "Trello," https://trello.com, accessed: 2018-12-01.

[4] "Tmnt," https://github.com/EricssonResearch/tnmt, accessed: 2018-12-12.

[5] "Next unit of computing," https://en.wikipedia.org/wiki/Next_Unit_of_Computing, accessed: 2019-01-06.

[6] "Gym," https://gym.openai.com/, accessed: 2018-12-21.

[7] "Robot operating system," http://www.ros.org/, accessed: 2018-12-10.

[8] "Gym," http://wiki.ros.org/kinetic, accessed: 2018-12-21.

[9] "Irobot," https://www.irobot.se/, accessed: 2018-12-20.

[10] "amcl," http://wiki.ros.org/amcl, accessed: 2018-12-20.

[11] "Gazebo," http://gazebosim.org/, accessed: 2018-1-18.

[12] "Vrep," http://www.coppeliarobotics.com/, accessed: 2018-12-18.

[13] "V-rep: Virtual robot experimentation platform," https://github.com/ar0ne/v-rep-robotics/tree/master/q-learning, accessed: 2018-12-17.

[14] T. Lei and L. Ming, "A robot exploration strategy based on q-learning network," in *Real-time Computing and Robotics (RCAR), IEEE International Conference on*. IEEE, 2016, pp. 57–62.

[15] C. B. Lillelund, "Transferring deep reinforcement learning from a game engine simulation for robots," in *Transferring Deep Reinforcement Learning from a Game Engine Simulation for Robots*. IEEE, 2018, pp. 17–21.

[16] "Robotics e-manual," http://emanual.robotis.com/docs/en/platform/turtlebot3/machine_learning/#machine-learning, accessed: 2018-12-15.

[17] L. Matignon, G. J. Laurent, and N. Le Fort-Piat, "Reward function and initial values: better choices for accelerated goal-directed reinforcement learning," in *International Conference on Artificial Neural Networks*. Springer, 2006, pp. 840–849.

[18] "Activation functions: Neural networks," https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, accessed: 2018-12-12.

[19] “Arcore,” https://developers.google.com/ar/discover/, accessed: 2018-12-04.

[20] “Yolo: Real-time object detection,” https://pjreddie.com/darknet/yolo/, accessed: 2018-12-20.

[21] “Docker containers,” https://dzone.com/articles/top-10-benefits-of-using-docker, accessed: 2018-12-17.

[22] M. Everett, Y. F. Chen, and J. P. How, “Motion planning among dynamic, decision-making agents with deep reinforcement learning,” *CoRR*, vol. abs/1805.01956, 2018. [Online]. Available: http://arxiv.org/abs/1805.01956