ÖZYEĞİN UNIVERSITY

CS 445 Project Report

Autonomous Driving Turtlebot

S004228 – Ekrem ÇETİNKAYA

S004345 – Batuhan ERDEN

Table of Contents

1. Introduction

Autonomous driving robots have always been a dream for humanity since the first vehicles arrived. Popularity of the autonomous driving field increased thanks to rapid advancement in machine learning field in the recent years. With the lately emerging machine learning methods like reinforcement learning and deep learning, autonomous driving models are almost ready to replace human drivers in roads. A good example is Tesla Motors and their auto pilot systems although they still require a human driver for safety.

Deep reinforcement learning was first introduced in 2013 with the Playing Atari with Deep Reinforcement Learning [1] paper. However, it was not very successful in attracting attention of the researchers in the field. Deep reinforcement learning became the core interest topic in the reinforcement learning fields with the Human-level control through deep reinforcement learning [2] that was published two years after the first paper, in 2015.

Our aim in this project was to develop an autonomous driving robot, Turtlebot, that is capable of navigating in closed areas like warehouses, offices and college campuses without any collision. Given a destination, it should be able to move there by following a reasonable route. This robot can be used for transporting objects in closed areas like moving products to the correct shelf in warehouses or distributing documents to faculty members in campuses. We used deep reinforcement learning techniques to achieve our goal.

For this project, we have used Gazebo simulation to simulate the environment and actions of Turtlebot. We have used Robot Operating System (ROS) to control Turtlebot in the Gazebo environment. Following chapter includes details about the setup.

2. Setup

1. **Gazebo**

   Gazebo is a free robot simulation tool that provides variety of useful features. It provides a reliable physics engine; robust sensor data and its Python libraries are easy to adapt. In this project Gazebo 2.2.3 was used.

2. **Robot Operating Software (ROS)**

   ROS is a set of software libraries and tools that are designed for building robot applications. It is an open source project. In this project ROS Indigo was used.

3. **Turtlebot and Microsoft Kinect**

   Turtlebot is a personal robot kit with an open source software. Turtlebot 2 was used in the Gazebo simulation with a Microsoft Kinect integrated into it to obtain depth data from the environment.

Our setup consists of a host computer and a virtual machine running inside of host computer. Simulation runs on virtual machine and training algorithms are executed in host computer.

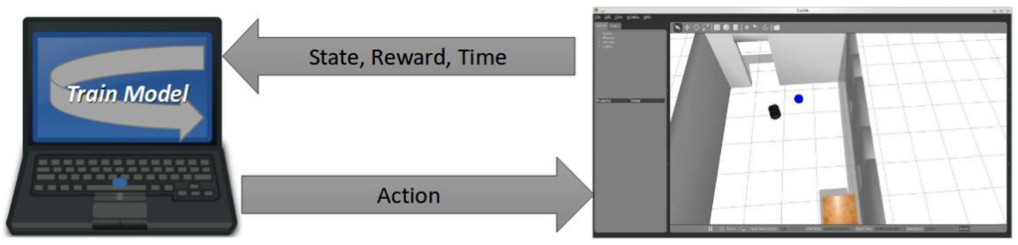


Figure 1: Setup Diagram

We have used a virtual machine image that was pre-installed with Gazebo and ROS for Turtlebot from Mathworks website [3]. An installation guide of the setup that we used can be found at the appendix part of the report.

Systems communicate through a TCP socket. Virtual Machine takes the action from host, performs that action and sends the resulting state, reward, time data to host again. In the next section of the report, information about training approach can be found.

3. Environment and State Space

The environment in the project was an office building with 3 rooms. There were some objects in the environment like bookshelves and balls.

State Space

States in the model are tuples with two entries:

$$State(t) = < Distance(t), Depth(t) >$$

$Distance(t)$ is the remaining distance to the destination as percentage. For example, if the distance between initial location of the agent and destination was 100m and the distance from current location of the robot to destination is 90m, then $Distance(t)$ will be 0.90 in this case. $Depth(t)$ is the depth data that is obtained from Kinect sensor at timestamp t. Obtained depth image from Kinect is a 480x640 matrix. Each value represents the distance to the corresponding pixel in the RGB image. In order to reduce the state size, depth values are compressed into a matrix with size 1x3. Depth matrix is divided into three parts; left is the columns 0-160, middle is 160-480 and right is 480-640. Final depth matrix has three elements; d_1 is the average depth in the left of robot's field of view, d_2 is the average in middle, d_3 is the average in the right.

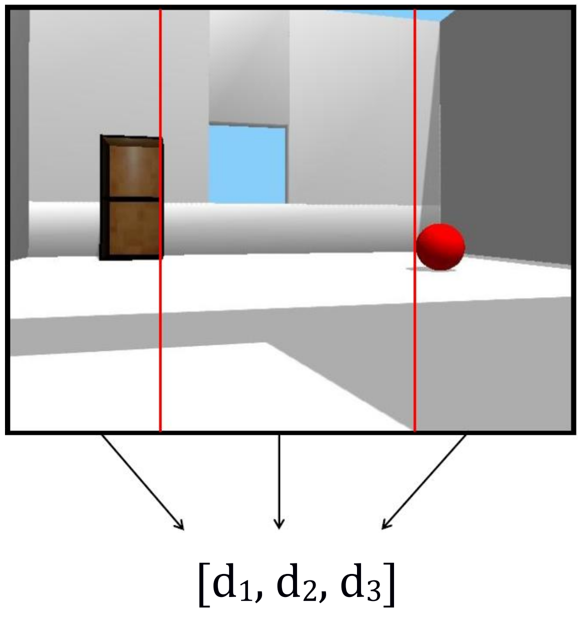


Figure 2: Compression of Depth Image

Reward

Rewards in the model are also tuples with two entries:

$$\mathrm{Reward(S)} = < \mathrm{R_{greedy}}, \mathrm{R_{safe}} >$$

$\mathrm{R_{greedy}}$ is the reward for greedy model which is responsible for distance. Models will be explained in detail in the following sections. Reward for greedy model is determined as following:

$$\boxed{\begin{gathered} S = Current\ State, S_T = Terminal\ State \\ R_{greedy} = \begin{cases} 100 & if\ S = S_T \\ -int(\max(0, Distance)) & otherwise \end{cases} \end{gathered}}$$

Figure 3: Greedy Reward

Note that, the distance values are casted into integers while giving reward. For example, distance values 0.965, 0.340 will give 0 reward and values 1.12, 1.59 will give -1 rewards here. The purpose is to punish the model when it is moving away from the original location.

$\mathrm{R_{safe}}$ is the reward for safe model which is responsible for depth this time. Reward for safe model is determined as following:

$$\boxed{\begin{gathered} S = Current\ State, S_C = Crash\ State \\ R_{safe} = \begin{cases} -100 & if\ S = S_C \\ -int\left(0.75 - \min\left(0.75, \mathrm{Depth_{min}}\right)\right) * 10 & otherwise \end{cases} \end{gathered}}$$

Figure 4: Safe Reward

As mentioned before, depth values are compressed into 1x3 matrix. $\mathrm{Depth_{min}}$ variable here is the **minimum** of those 3 values since we are interested only in the closest object in the field of view.

Action Space

Turtlebot is capable of rotating 360° and increasing its speed up to 0.7 mps. However, to simplify the problem, we have restricted its actions to three. There are three possible actions for Turtlebot. Going left, going forward and going right. Note that going left is not equal to turning left.

4. Training

Exploration and Exploitation

The environment and state space in our simulation is huge. In order to maximize the exploration rate and minimize the error at the same time, the value of ε (epsilon) starts from 1. At each time step, this epsilon value is multiplied by the Ψ (exploration rate), which is 0.999. This means that the exploration rate of the agent decreases by 99.9% percent at each time step. The multiplication is repeated until the value of epsilon reaches its minimum value, which is 0.2, and the epsilon value is never decreased again. However, having the epsilon value 0.2 is beneficial most of the times but the agent still needs to explore new paths. These kinds of situations are easily solved by the εε (multiple epsilon). Multiple epsilon is a method in which the value of the epsilon is multiplied by 1.5 with a probability of 25%. This method prevents the agent from using same paths and lets it explore new paths.

Multi-modeling Approach

There are 2 kinds of variables in this problem. One is the distance to destination point and the other one is how close the agent is to objects, the depth values. As described in the state space, the state is a tuple with 2 entities, which are the distance and the 1x3 depth values. Multi-modeling Approach basically means that 2 different models are trained simultaneously instead of 1 model. The idea comes from how brains work. Brains are divided into 2 portions and each portion is responsible for different actions. It is decided that the agent has a brain like design and it is divided into 2 portions. One of the portions is responsible for learning to minimize the distance to destination and acting based on what it has learned. This model is called the **greedy model** because it does anything to minimize the distance, it does not care about if there is any obstacle or not. On the other hand, the other model is responsible for learning how to avoid obstacles and acting based on what it learned. This model is called the safe model because it protects the agent from hitting to an obstacle. Moreover, two models are learning and affecting the agent simultaneously. However, if safe model detects that the agent has a risk of collision, it takes the control and does anything that it could do to avoid the collision.

As described above, the greedy model is eager to reach to the destination. So, its main goal is minimizing the distance whereas the safe model's main goal is to avoid obstacles because it's responsible for maximizing the distance between the robot and the objects. On the other hand, both models have their own memories. That is, one model does not remember the experience of the other model.

Neural Network Setup

There are four different neural networks in this project, one for each model. Two neural networks for greedy model consist of three layers, two hidden layers and one activation layer:

1. Dense Layer, hidden size = 100, activation function = ReLu
2. Dense Layer, hidden size = 100, activation function = ReLu
3. Dense Layer, hidden size = 3, activation function = linear

   Optimizer = Adam

   Learning Rate = 0.01

Neural networks for safe model has one more layer since depth state is more complex:

1. Dense Layer, hidden size = 100, activation function = ReLu
2. Dense Layer, hidden size = 100, activation function = ReLu
3. ***Dense Layer, hidden size = 100, activation function = ReLu***
4. Dense Layer, hidden size = 3, activation function = linear

   Optimizer = Adam

   Learning Rate = 0.01

Deep Double Q-Learning

The reinforcement learning algorithm that is used in the project is Deep Double Q-Learning. Q-Learning is a value gradient algorithm that is commonly used in reinforcement learning applications. Value gradient approaches directly learn the value of being in a state and act accordingly. The Q-learning algorithm directly approximate the optimal action-value function, independent of the policy being followed.

The reason for using double Q-learning is to prevent maximization bias. Maximization bias occurs in many reinforcement learning algorithms since they use the following approximation:

$$max(\boldsymbol{E}[a], \boldsymbol{E}[b]) = \boldsymbol{E}[max(a, b)]$$

However, this approximation is not correct, and the resulting bias is called maximization bias. Double Q-learning is one solution to this problem.

In the project, both of greedy and safe models are double Q-learning agents with different discount factors. Discount factor for greedy is 0.99 and it is 0.95 for safe. It is lower for safe agent since closer values are more important for it.

Policy

As described in *Section: Exploration and Exploitation*, the value of the epsilon is updated at each time step until it reaches its minimum value. On the other hand, the value of the epsilon is multiplied by 1.5 with a probability of 25%. This method is called the Multiple-Epsilon Method. As this is a ε-greedy policy, with a random probability less than the value of epsilon, the actions are chosen randomly. The time when the actions are not chosen randomly, the algorithm makes use of the 2 models to decide which action to choose.

As in Double Q-Learning algorithm, there are 2 approximate value functions for each model. Since there are 2 different models (Multi-modeling Approach), we have 4 approximate value functions, Qs. The policy for each model is based on the summations of the two action-value estimates. First, the final action-value estimate of each model is calculated. Then, the actual overall action-value estimate is calculated by adding the final action-value estimates of each model. Finally, using the actual overall action-value estimate, the policy returns the actions with the highest value.

Test results of a Random agent and our agent will be explained in the next section.

5. Results and Discussion

Test Environments

There are three test environments in the project. Two of the environments contains less objects therefore risk of collision is lower in those environments. Also, distance to destination in environments increase towards environment three. Destinations are marked with green lights in the following pictures:

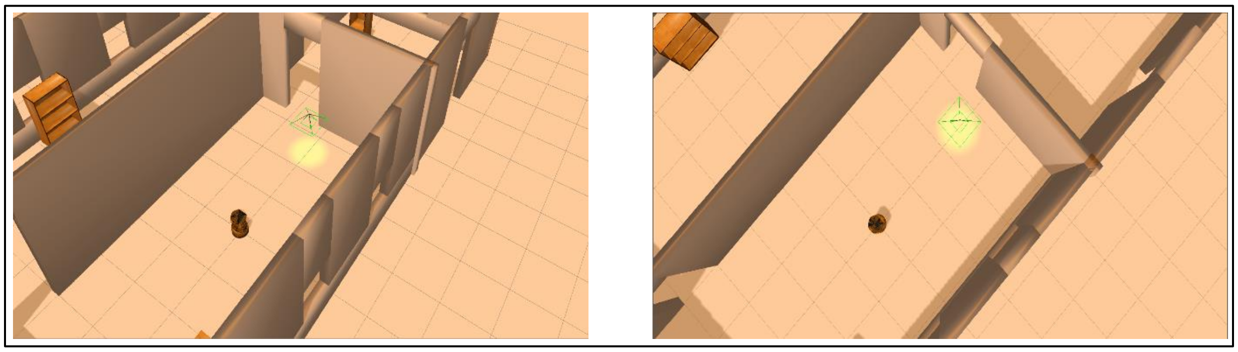


Figure 5: Test Environment One

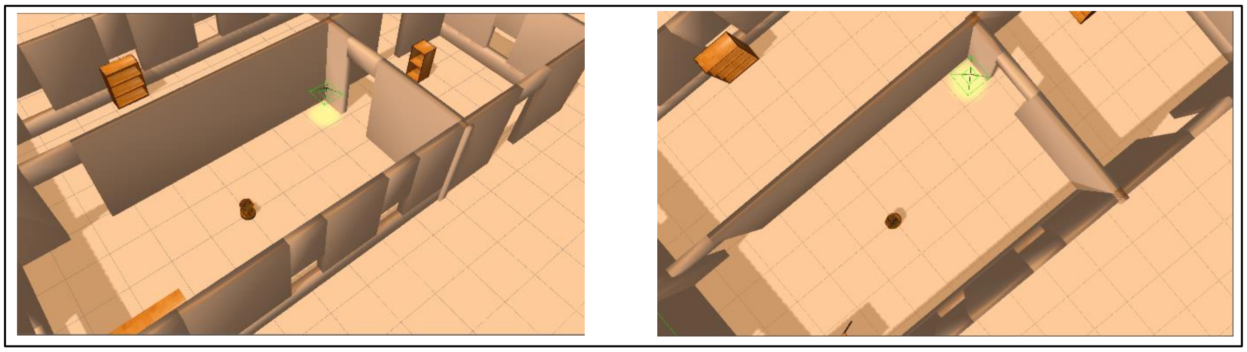


Figure 6: Test Environment Two

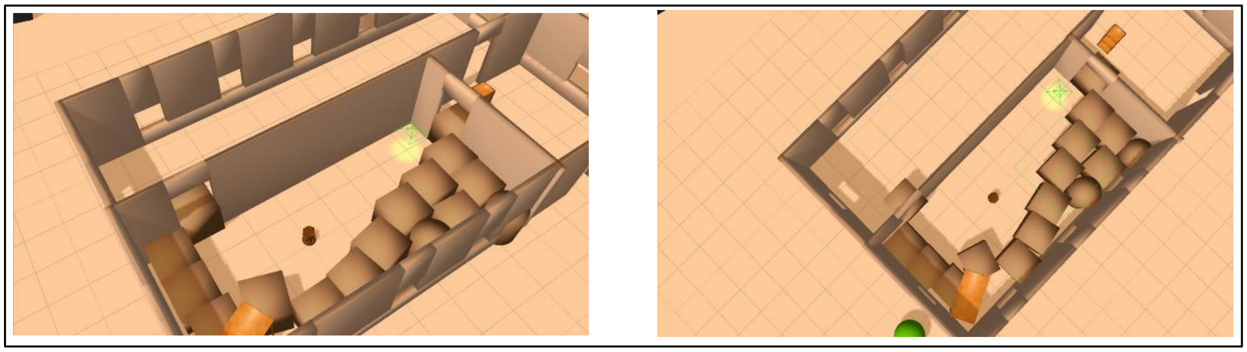


Figure 7: Test Environment Three

Test Results

Tests in Environment One

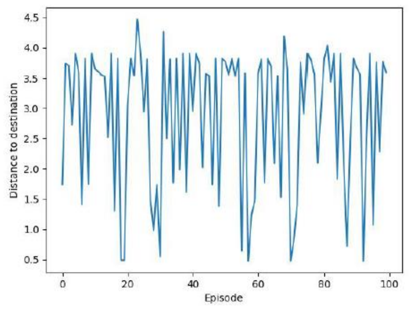
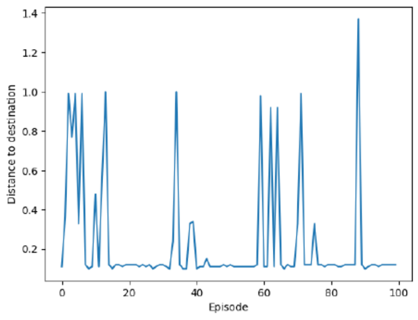
Random Agent	*Trained Agent*
	

Figure 8: Test Results in Environment One

Note that the trained agent was using Q-learning instead of double Q-learning. We have made modifications in the algorithm since it failed to perform in tougher situations. First test environment was relatively easy, the destination was close and there were not any objects between the robot and the destination. The improve in the agent can be seen easily in the graph.

Tests in Environment Two

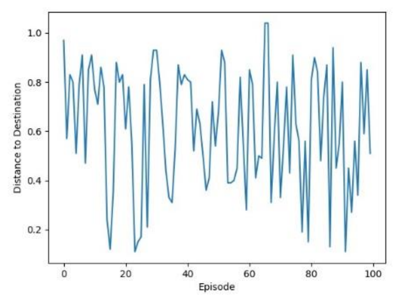
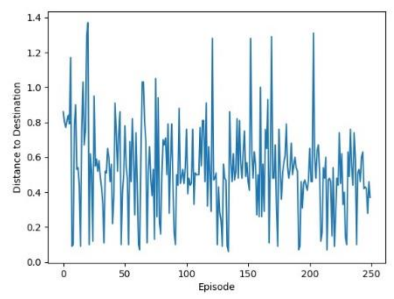
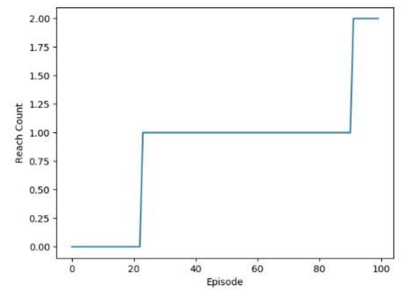
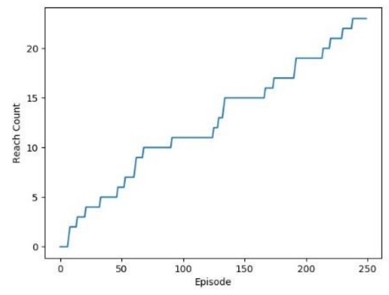
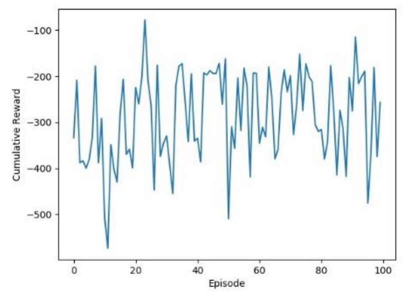
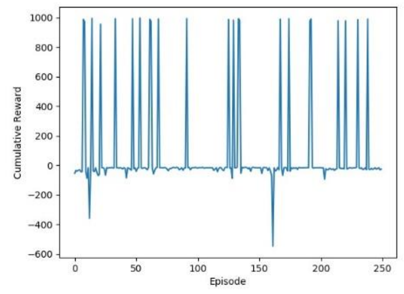
Random Agent	*Trained Agent*
	
	
	

Figure 9: Test Results in Environment Two

At this point, we have made changes in the agent and switched it to the algorithm that is mentioned throughout in this report. Those changes have been made because first agent could not perform in those environments. As can be seen in the graphs, the learning process was longer than the first environment since the distance to destination is increased. Also, change in the algorithm caused more need for epochs because it got more complex. Overall, the results were promising, and it outperformed the random agent in this case quite easily.

Tests in Environment Three

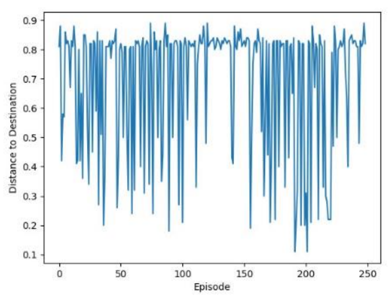
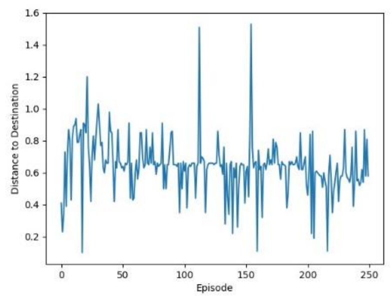
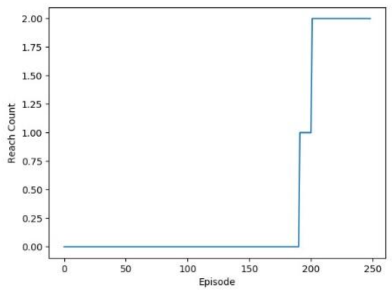
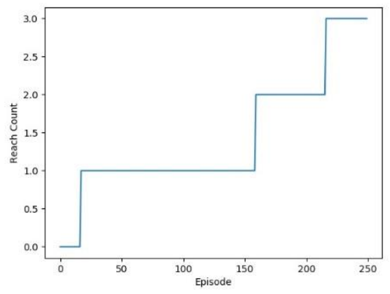
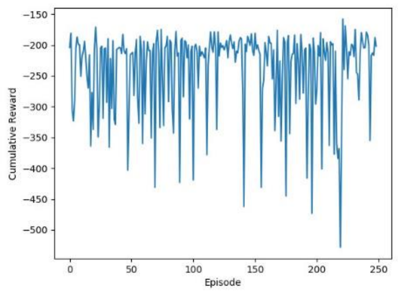
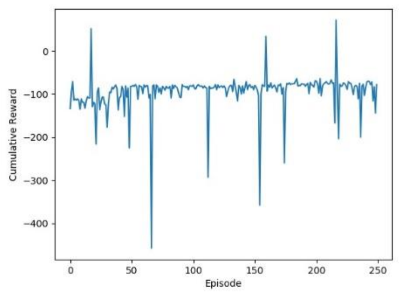
Random Agent	*Trained Agent*
	
	
	

Figure 10: Test Results in Environment Three

In Figure 10, the last test results can be seen. This was our most complex environment since it needs an agent that learns both distance and depth at the same time. If we look at both the random and the trained agent, we can see how dizzy the random results are. On the other hand, we can see that our agent learned to minimize the distance and the risk of collusion. For example, in the graph that shows the distance to destination, we can see that the distance decreased over time (less is better). On the other hand, the graph that shows the reach counts reveals that reach count incremented more over time. That is, as timestamps passed, agent has reached to the destination more frequently.

Discussion and Conclusion

Overall, this project has been a great experience for us. It definitely took our reinforcement learning skills to a higher level. However, it was too hard to work with gazebo simulation because learning how to communicate with the robot was something different. It was like learning syntax of a language. We imported the necessary libraries and learned how to communicate with the robot through ROS. For instance, in order to move the robot, the information of that command with the correct arguments should be published to rospy (The main library of ROS). Furthermore, each action should be published to rospy that listens these and acts based on what is published. On the other hand, we observed that Gazebo sometimes gives the depth values wrong or delayed. This obviously has a bad effect on our results. Although, we have faced with too many problems, we think that the results are promising. We have only run it for 250 epochs and in the graphs, we can see that the agent is learning with a small learning rate. Since there are lots of action-value pairs that our agent needs to learn, we think that the agent would need at least 1500-2000 epochs before it completely learns.

References

[1] V. . Mnih, K. . Kavukcuoglu, D. . Silver, A. . Graves, I. . Antonoglou, D. . Wierstra and M. A. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv: Learning,* vol. , no. , p. , 2013.

[2] V. . Mnih, K. . Kavukcuoglu, D. . Silver, A. A. Rusu, J. . Veness, M. G. Bellemare, A. . Graves, M. A. Riedmiller, A. K. Fidjeland, G. . Ostrovski, S. . Petersen, C. . Beattie, A. . Sadik, I. . Antonoglou, H. . King, D. . Kumaran, D. . Wierstra, S. . Legg and D. . Hassabis, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, no. 7540, pp. 529-533, 2015.

[3] Mathworks, "Virtual Machine with ROS Indigo and Gazebo for Robotics System Toolbox™," [Online]. Available: https://www.mathworks.com/supportfiles/robotics/ros/virtual_machines/v3/installation_instructions.htm.

Appendix

Installation Guide

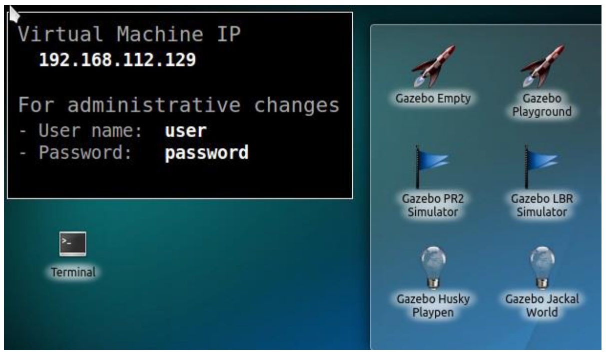
1. Download the appropriate virtual machine for your platform from and follow the instructions: https://www.mathworks.com/supportfiles/robotics/ros/virtual_machines/v3/installation_instructions.htm

2. After installation is complete, open the VM and run following commands:

   ◦ echo export ROS_MASTER_URI=http://192.168.112.129:11311 >> ~/.bashrc

   ◦ echo export ROS_HOSTNAME=192.168.112.129 >> ~/.bashrc

Note that **192.168.112.129** is the IP address of your VM that is shown in the desktop of VM.



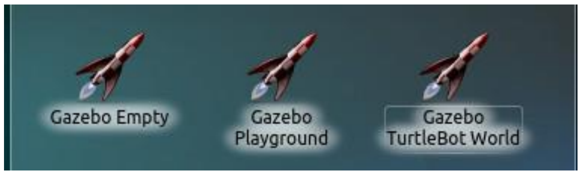
```
Virtual Machine IP
  192.168.112.129

For administrative changes
- User name:   user
- Password:    password
```

3. Open the **config.ini** file and change the host.addr to IP address of your PC and vm.addr to IP address of your VM.

```
[Network]
host.addr = 192.168.1.29
vm.addr = 192.168.112.129
port.header = 50000
```

4. Put ***agent.py, environment.py, memory.py, util.py, vm.py, vm_bringup.py, config.ini*** files to a folder in virtual machine.

6. Double click **Gazebo TurtleBot World** to run gazebo in VM.



6. Run **host_bringup.py** in the PC and **vm_bringup.py** in the VM.