**Documentation and Instructions:**

**General Pointers:**

* To run the training or control algorithm, we need the environment, world, robot, launch file to integrate the environment and robot, and control algorithm to facilitate learning.

1. World file: found in gym-gazebo/gym\_gazebo/envs/assets/worlds. This file provides the world, etc.
2. Environment file: found in gym-gazebo/gym\_gazebo/envs/turtlebot. This file provides the environment, action space, observation space, etc.
3. Launch file: integrates the robot and environment when we start the control algorithm.
4. Init file: registers the environment with gym
5. Control algorithm: Q-learn, DQN, or whatever algorithm used.

* Gym follows certain naming conventions so when making custom worlds and environments ensure that those naming conventions are followed. To create a custom environment, you need the environment file (a python file containing a class whose name closely resembles the name of the environment). To register it, you need the Init file where you add in your environment name and path (follow the syntax of the already registered environments)

**Creating Environments:**

1. **Static Obstacles in existing worlds**

To create the environment,

1. Go to gym-gazebo/gym\_gazebo/envs/assets/worlds
2. Copy the code of the world file you want to add obstacles to and paste into a new text file
3. Open this world (duplicated world) in gazebo

To add the obstacles,

1. To add shapes, click the shapes on the top and place the cursor wherever you want to add them.
2. To add specific objects, click insert from the left bar. A list of models gazebo offers will appear. Click whichever one you want to insert and place your cursor there.
3. To scale the objects, click on the object and click the double arrow button on the top bar (fourth button/button near the round arrows). The 3 axes will appear and you can resize in your preferred direction.

To save your world,

1. Click File, Save World As.
2. You can override the world by saving it using the name you originally saved it as.

To check if your world has all these obstacles,

1. Type gazebo <world-name> on your terminal
2. If everything is correct, the new world with the obstacles will open in Gazebo.

**Making your world accessible to the launch file and control algorithm:**

* After creating the world, we need to make it accessible to the launch file so that it uses this specific world when we run the control algorithm
* To register/source your world, type gedit ~/.bashrc
  + You can see the worlds that have been sourced and registered by the gym-gazebo repository when you installed and made the repository.
* Provide a name for your world in the format of the existing worlds (Uppercase letters
* Copy the path name of the world and paste it below
* In the launch file

(gym-gazebo/gym\_gazebo/envs/assets/launch/GazeboCircuit2TurtlebotLidar-v0.launch), add the name of the world in the tag which says <arg name….default..>

1. **Dynamic Obstacles in existing worlds**

For dynamic obstacles, we need to add the object, and use some method to make it move.

Part I: Adding the objects

* Follow the same method as above

Part II: Making the objects move

To make the objects move, we can use different methods

1. Actor tag from Gazebo(<http://gazebosim.org/tutorials?tut=actor&cat=build_robot>)

* The actor tag is a feature in Gazebo we can use to make animations
* This uses a C++ plugin code to create the animation
* This method can only be used in Gazebo versions 8 and above. However, when we use the actor tag and try to run the control algorithm, the robot does not move. So, we cannot use this method for this repository.

1. Animate the object using a C++ plugin **add link**

* Add the line for the plugin **copy the line here** to the world file, right above the </sdf> tag
* The same issue is experienced here as well. Due to an issue with the plugin, either the robot does not move, or it moves but does not learn navigation even after 300 episodes of the control algorithm.

1. Updating the state of the object using ROS

* For this, we need a specific code (called ModelStatePublisher.py in the repository)
* This code publishes the state of the model every second to ROS.
* In this method, the obstacle, robot, and control algorithm work.

To run this setup,

1. Go to gym-gazebo/examples/turtlebot and python circuit2\_turtlebot\_lidar\_dqn.py
2. On another terminal, export the ROS\_MASTER\_URI for this simulation and type rosparam set /use\_sim\_time true

Gazebo simulation time and ROS time (controlled by the /clock topic) are different. So when we update the object’s state in ROS, we are using ROS time. However, our simulation runs using Gazebo’s simulator clock. After every time the turtlebot hits the wall, a new episode starts and the simulator clock is reset to zero. This does not match the ROS time and so, we will get an error saying “ROS time has moved backwards”. To avoid this, we set the use\_sim\_time parameter in ROS to true.

1. On another terminal, change to the appropriate directory, export the ROS\_MASTER\_URI again, and type python ModelStatePublisher.py
2. On a fourth terminal, export the GAZEBO\_MASTER\_URI, and type gzclient. Gazebo should open up and display the simulation running.

In the simulation, you should be able to see the robot moving, the lidar (blue color), and the object moving.

**Control Algorithm:**

An issue experienced is that the control algorithm is not robust enough to support learning with dynamic obstacles. So, when you use the dqn code in the gym-gazebo repository, the robot seems to be navigating effectively until about 200 episodes, but then seems to unlearn or forget anything it has learnt so far, and deliberately hits the wall closest to the starting point.

To resolve this, a more robust control algorithm could be found. The Reinforcement Learning library in Keras offers an easy, ready-to-use dqn agent with many preset values. (<https://github.com/keras-rl/keras-rl/tree/master/examples>). A good example to start with might be the dqn code used for training the Atari games (dqn\_atari.py)

**Import issues and running the dqn algorithm using Python 3:**

ROS 1 and gym-gazebo support Python 2.7 but the Keras-RL library only supports Python 3.5 and above. Running this dqn control algorithm using the existing versions of ROS and gym-gazebo cause strange import errors. To resolve this, the necessary modules would have to be imported in a certain order. This order did not have any particular logic and these import errors were not resolvable using any online guide or error discussion forum. So, please use the files from the repository as they are when running Python3 algorithms with this repository.

**Running the trained models on a real robot:**

[Note: The robot used to try this was iserared]

The primary use of using Gazebo is for us to see how a robot would perform under more ideal conditions when trained with a certain control algorithm. If we can transfer this trained model to a real robot, there would be less randomness or noise when we begin to train the robot itself.

1. ***Establishing connection between your computer and the computer and router which control the robot:***

This is done by sshing into the controlling computer.

* Open a new terminal and type ssh (name of user)@(IP address of the router controlling the robot)
  + In our case, we used ssh [isera2@192.168.1.136](mailto:isera2@192.168.1.136) or ssh [isera2@192.168.1.101](mailto:isera2@192.168.1.101)
* Enter the other system’s password
* Your terminal would say welcome and display Ubuntu version and documentation details **ADD PICTURE**

1. ***Initialize robot to run trained model:***

* To run any algorithm on the real robot, we need to launch 2 initial files: start\_mobile.launch and start\_intelligence.launch
  + On a second terminal
    - Type roslaunch sliding\_autonomy start\_mobile.launch robotname:=iserared
  + On a third terminal
    - Type roslaunch siding\_autonomy start\_intelligence.launch robotname:=iserared
  + Note: we need to launch startmobile first because this launch file starts roscore as well.

1. ***Pipeline of information/commands and the rostopics needed for running the model on the real robot***

We generally need to focus on 2 rostopics: one which takes in the input (image of surroundings or laser scan) and sends this information to the control algorithm and one which takes instructions from the control algorithm and transfers them to the motor.

* For Gazebo, the /scan topic takes scan information from /gazebo and the /mobile\_base/commands/velocity topic publishes Twist information to /gazebo. We need to replace /gazebo with the real robot’s scan and motor topics.
* For iserared, the scan topic is named /scan and its data type is the same as the /scan in the simulation. So, we do not need to change anything for the scan information.
* However, the motor control topics are named differently. In the simulation, it is called /mobile\_base/commands/velocity and in iserared, it is called /RosAria/cmd\_vel.
  + To make these two topics communicate, we create a ros node which subscribes to /mobile\_base/commands/velocity (this topic is its publisher) and publishes to /RosAria/cmd\_vel (this topic is its subscriber). The python code which does this is called NodeConnections.py and can be found in the repository.

1. ***Running the model on the real robot –* CONFIRM THIS PART**

* To run the model, we change directory to gym-gazebo/examples/turtlebot and type python runtraining…..
* Exporting the ROS\_MASTER\_URI from the learning algorithm will not work because we cannot have two ROS masters running.

The best way to run the trained model is to copy the model information and run just the model (with weights and architecture) as a h5 file on the isera2 system.