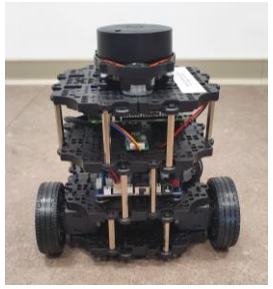
Deep Reinforcement Learning for sensor-based navigation



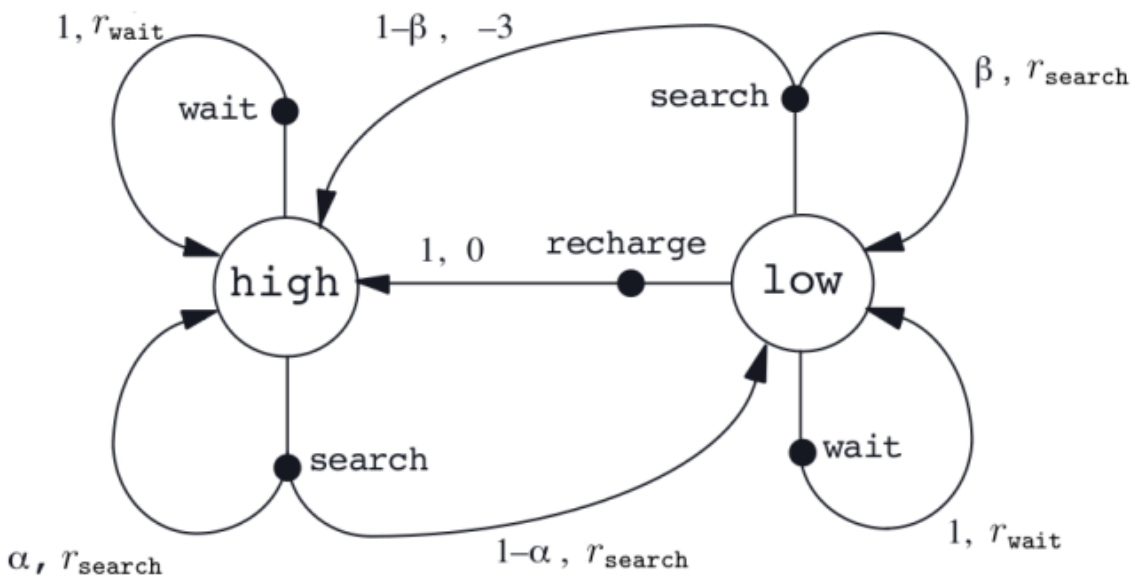
Tutor: Francesco Trotti
francesco.trotti@univr.it

Agenda

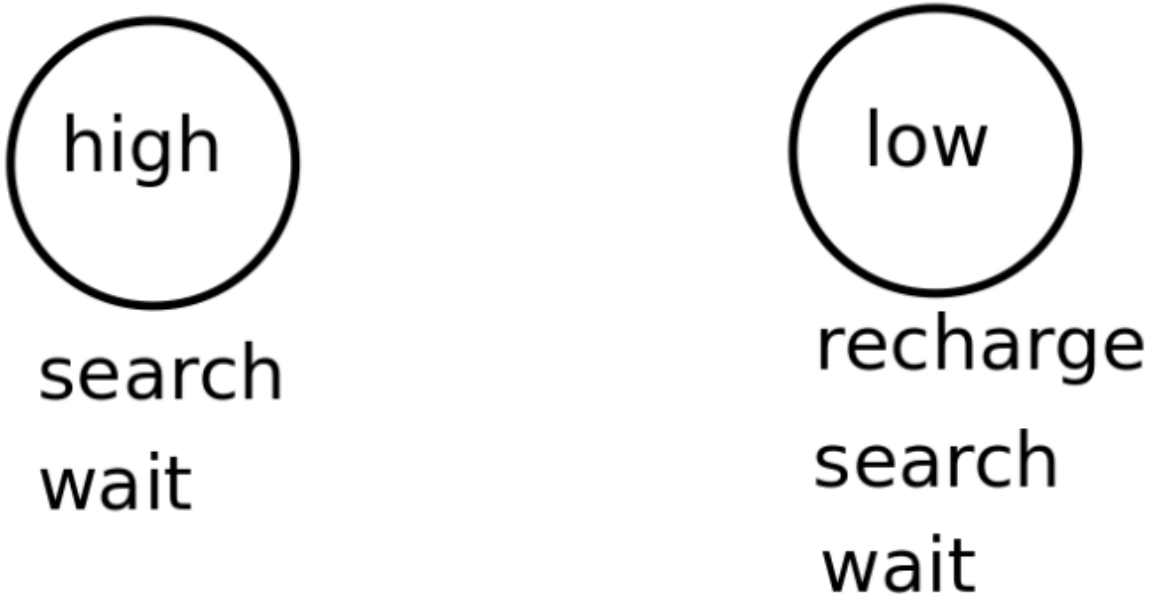
- Introduction to DRL and DQN
- Introduction to ML-Agent
- Exercise on simulated robot and turtlebot3

Introduction to DRL and DQN

Planning vs Learning



Planning



Learning

Model-Based vs Model-Free

Model-Based: build a model (**MDP**) and then solve the model (**compute best policy**)

- ✓ Avoid repeating bad states/actions
- ✓ Fewer execution steps
- ✓ Efficient use of data

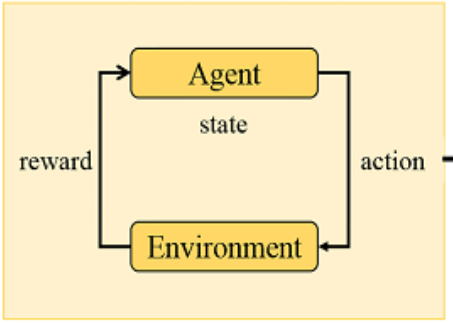
Model-Free: directly learn a **Q-function** and a **policy**

- ✓ Simplicity, no need to estimate and solve a model
- ✓ No bias in model estimation

Tabular Q-Learning

Algorithm 1 Tabular Q-Learning

1: Initialize $Q(s, a)$ arbitrarily
2: Initialize s {observe current state}
3: **loop**
4: Select and execute action a
5: Observe new state s' receive immediate reward r
6: $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
7: update state $s \leftarrow s'$
8: **end loop**



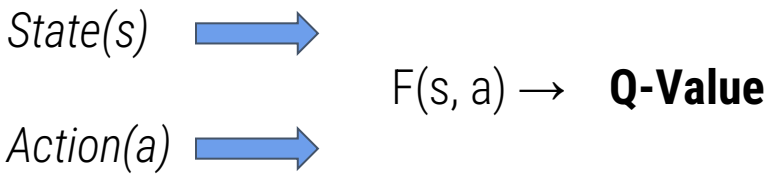
states	actions			
	↑	↓	←	→
(1,1)	0	0	0	0
(1,2)	0	0	0	0
...	...	...	...	...
(m,n)	0	0	0	0

Select action (ε-greedy)

- choose best action according to current Q-Value estimate,
- once in a while (i.e., with probability ε) choose a random action (with uniform probability across all actions)

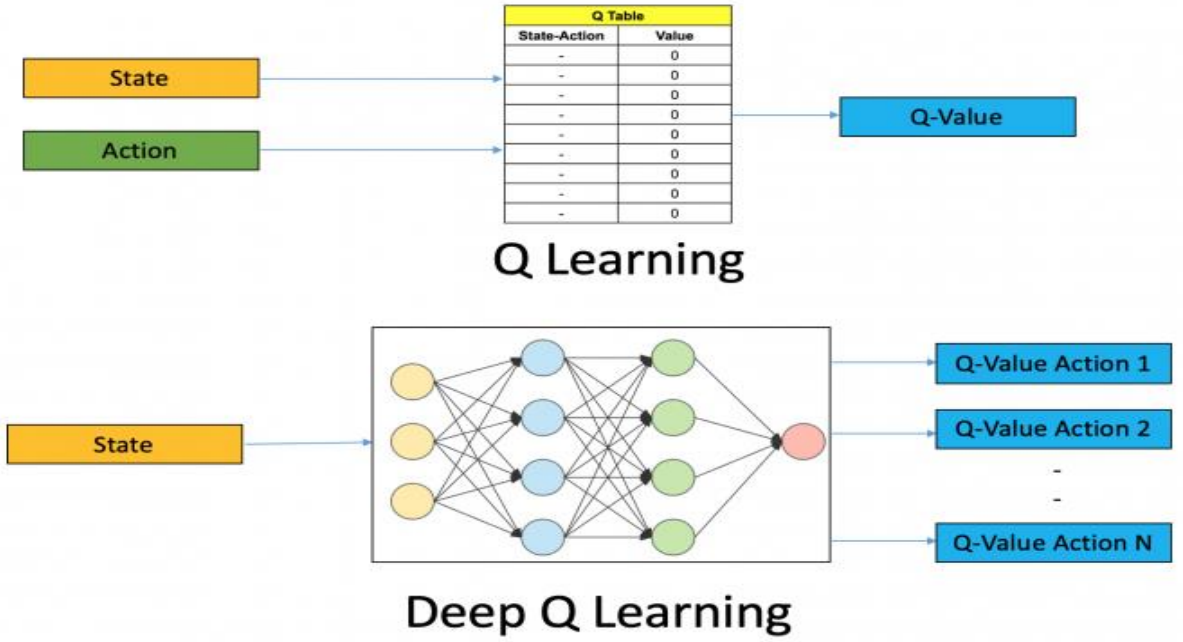
Deep Q-Learning (DQN)

- In a real-world scenario, the number of states could be huge, hence building the Q-table is unfeasible.
- To address this limitation, we use a **Q-function** rather than a Q-table, which maps state and action pairs to a Q value.



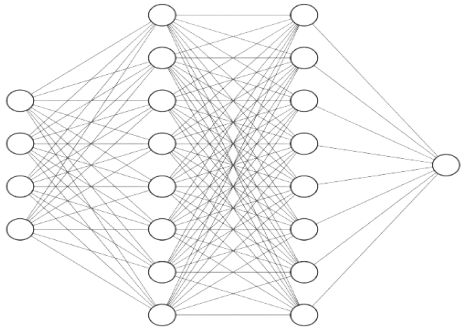
Deep Q-Learning (DQN)

- We can use a **deep neural network**, which we call a Deep Q Network, to estimate the Q-function

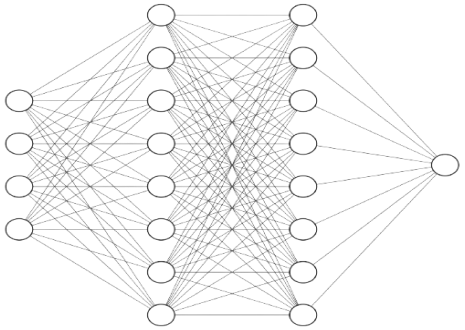


DQN main ingredients

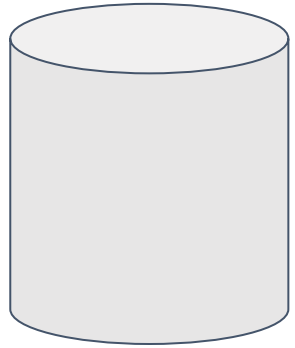
- The DQN approach is based on the following ingredients:
  - two DNNs, the **Q network** (or actor network) and the **Target network** to mitigate divergence: using a single network to choose and evaluate the action can led to divergence in the Q-Value estimation.
  - a component called **Experience Replay** or Memory buffer to store experiences (i.e., <s,a,s',r>). The experiences are then randomly sampled and proposed to the network. This helps de-correlating experiences and mitigating divergence.



Q Network



Target Network



Memory buffer

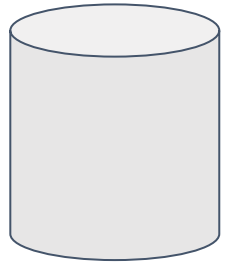
DQN for sensor based navigation

- The main idea is to train a DQN network that allows the robot to reach a target position avoiding obstacles
- Note that the robot does not have a map of the environment, but it learns the best action to reach the goal avoiding the obstacles based on sensor readings: **maples navigation**
- The DQN is trained over multiple time-steps and many episodes, and it goes through a sequence of operations in each timestep

action

Reward, new_state

We store all the information in a memory buffer



Memory buffer

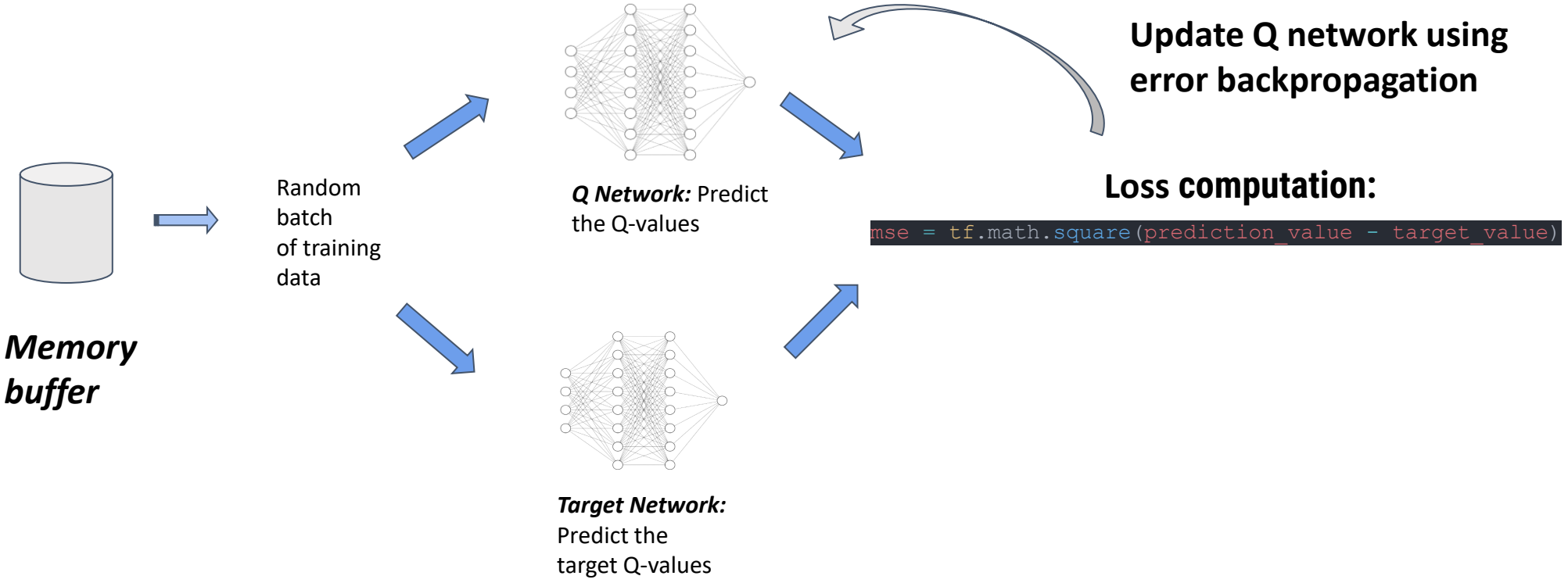
Deep Q-Network, standard approach

1: Initialize weights $\boldsymbol{w}$ and $\overline{\boldsymbol{w}}$ randomly in $[-1, 1]$
2: Initialize s {observe current state}
3: **loop**
4: Select and execute action a
5: Observe new state s' receive immediate reward r
6: Add (s, a, s', r) to experience buffer
7: Sample mini-batch MB of experiences from buffer
8: **for** $(\hat{s}, \hat{a}, \hat{s}', \hat{r}) \in MB$ **do**
9: $\frac{\partial Err(\boldsymbol{w})}{\partial \boldsymbol{w}} = (Q_{\boldsymbol{w}}(\hat{s}, \hat{a}) - \hat{r} - \gamma \max_{\hat{a}'} Q_{\overline{\boldsymbol{w}}}(\hat{s}', \hat{a}'))\frac{\partial Q_{\boldsymbol{w}}(\hat{s}, \hat{a})}{\partial \boldsymbol{w}}$
10: update weights $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Err(\boldsymbol{w})}{\partial \boldsymbol{w}}$
11: **end for**
12: update state $s \leftarrow s'$
13: every c steps, update target: $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}$
14: **end loop**

Human-level control through deep reinforcement learning
(V. Mnih et al., Nature 2015)

Our DQN-based approach (1/2)

- we update the actor network **only if we reach a terminal state** (i.e., if the agent reaches the target position or crashes)



Our DQN-based approach (2/2)

- Every step we update also the target network using a soft update rule:

$$\Theta_{target} \leftarrow (1-\tau) \cdot \Theta_{target} + \tau \cdot \Theta_{actor}$$

- Where:
  - Θ target/actor are the network weights
  - T is a coefficient for the soft update

Overall effect: increased **stability** and **robustness**: learn only when the agent achieves a final reward, learn slowly (soft update at every step).

DQN parameters

The DQN approach relies on various parameters:

- **gamma** (i.e., discount factor): is a value in range [0, 1) that is used to give more credit to immediate rewards (values closer to 0) or to delayed rewards (values closer to 1).
- **eps_decay**: how fast to decay the exploration parameter.
- **memory_size**: buffer memory
- **batch_size**: is the size of the sample that is used to train
- **network structure**: number of hidden layers, number of nodes (in the hidden layers), initial weights.

DQN components

- To train our model we need to define:
  - **States**: a specific configuration of the lidar and the distance to the target
    - The configuration is based on the use of a set of rays per direction in a ranges between -90 and 90 (e.g. using 3 rays for direction we will have 7 total rays, 1 ray every 30°)
    - The distance to the target includes the euclidean distance and the angle displacement
  - **Actions**: we define 3 discrete actions: 0 (move forward for 5 cm), 1 (perform a 30° left rotation) and finally 2 (perform a 30° right rotation)
  - **Reward function**: the reward function is crucial for training as it tells the agent what is correct and what is wrong.

DQN reward

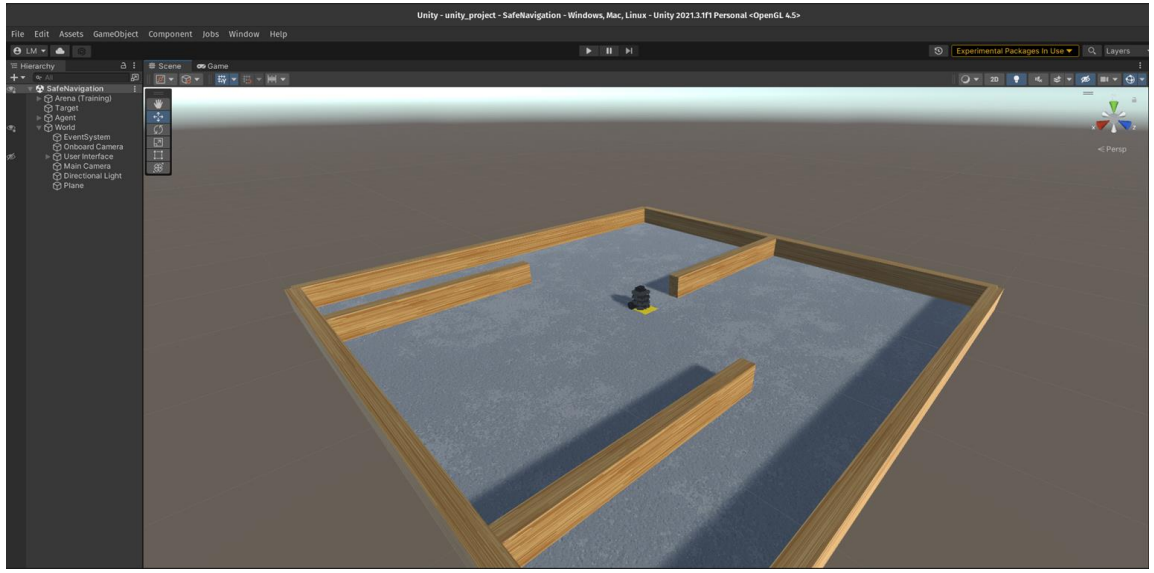
$$\begin{cases} reward\_multiplier \cdot distance(tg_{t-1}, tg_t) - step\_penalty & \text{if not done} \\ 0 \quad \text{if crashed} \\ 1 \quad \text{if tg reached} \end{cases}$$

- reward_multiplier is a constant (hint: 3 should be a good value) that multiplies the difference between the distance to the target at timestep t-1 and the new distance to the target (i.e., if the robot goes far away from the target gets a negative reward)
- step_penalty is a small penalty for each step to incentive the agent to reach the target faster (we suggest 0.001)

Introduction to ML-Agent

ML-Agent

- The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents
- We can use Unity and ML-agents to create our personal project for mapless navigation using our TurtleBot 3 as an agent.



Ingredients for ML-Agent

- Components for a ML-Agent system:
  - Agent (Turtlebot3)
  - Observation, i.e the lidar sensor of the robot
  - Walls and obstacles
  - A Deep Reinforcement Learning (DRL) algorithm to train our agent
- All these elements will be in our Unity ML-agent system

Exercise

Exercise, set-up

- Clone the repository
  - https://gitlab.com/TrottiFrancesco/mobile_robotics_lab.git
- In Unity Hub open the new cloned project called "turtlebot3DQN"
- In colcon_ws copy the new package called "turtlebot3_DQN"
- In DQN you can find the python code with the DQN network and the training/testing scripts
- In MobileRoboticsDQN you can find the “requirements.txt” to install all necessary library
  - In terminal run “*pip install –r requirements.txt*”
  - sudo apt install ros-foxy-tf-transformations

Exercise, training the DQN model

- The first step of the exercise is to train a DQN model
  - for the exercise in class use few steps of training, after the lesson you can increase the steps to create a better model
- To do this you have to change and implement different components:
  - **Implement** the reward function
  - **Set up** the Unity environment
  - **Set up** the step number for training
- After you have trained your model, you must save it
- You can then load your model and test it

Exercise, test a pre-trained network and test on the robot

- To avoid long training time during the class we provide a **trained model** that has a good success rate. You can test the trained model to see the correct behavior
  - Load the trained model as you would do for your model
- Finally, you have to write a ROS2 node to test the network on the simulated/real robot

Exercise, implementing the reward function

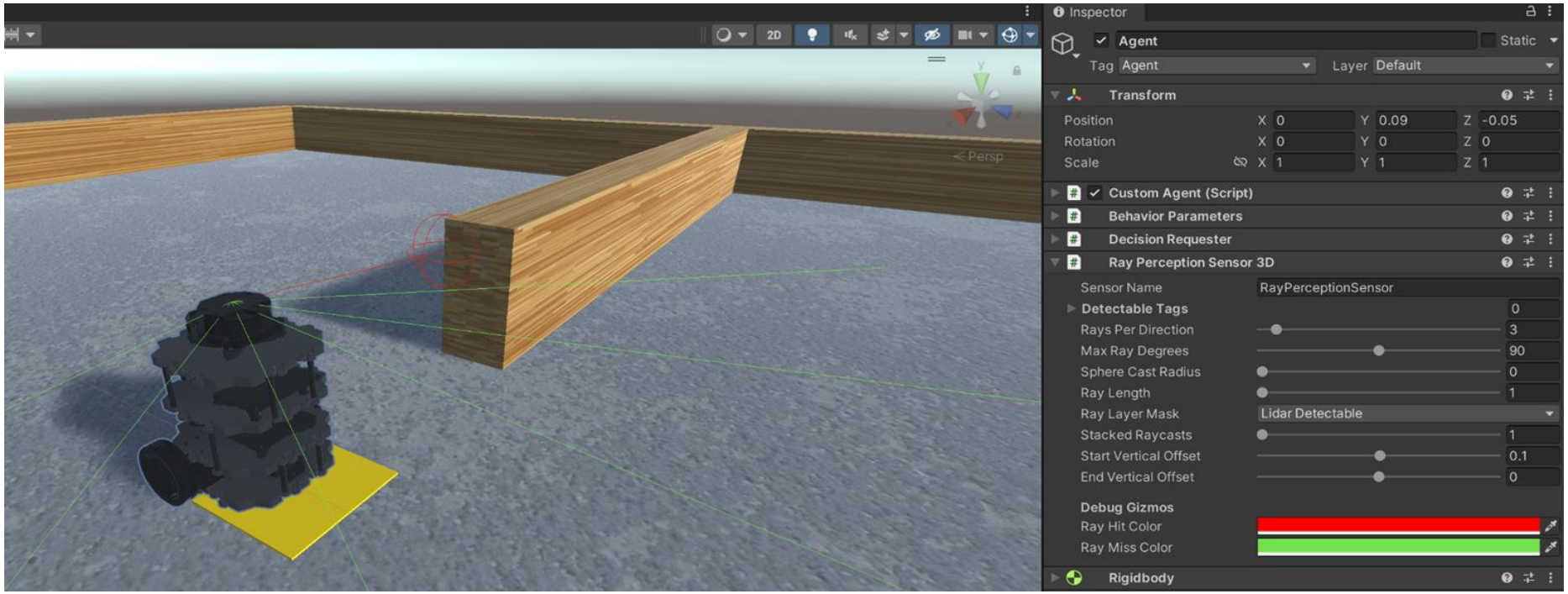
- To implement the reward function you have to:
  - Open the file *DQN/env/robotic_navigation.py*
  - Go to function called *override_reward()*
  - Write the function that computes the reward based on the distance from the goal
    - The variables *target_distance* and *new_distance* contain the distance from the goal to the position of the robot before and after executing the action

```
    # Here it's possible to override the reward given by the Unity Engine
    # default returns the standard reward from the environment
    reward = self.override_reward( state, reward, action, done )

    #
    return state, reward, done, info


def override_reward( self, state, reward, action, done):
    return reward
```

Exercise, setting up the unity environment

To set up the Unity environment you have to:

- Set the correct number of rays for the lidar: select the Agent game-object, Select the Agent inside the TrainingArena, Check the window on the right: “RayPerceptionSensor3D”
- Set a max length of rays and radius of collision sphere (we suggest 1 and 0.0)



Exercise, setting the number of training steps

- To set the number of training steps you have to:
  - Open the file DQN/config.py
  - Modify the parameter called “Episode length”
  - This number is fundamental to obtain a good model, for the exercise in class you can set a low number to avoid long training time but you should optimize this parameter to achieve a good training

Exercise, running the training phase

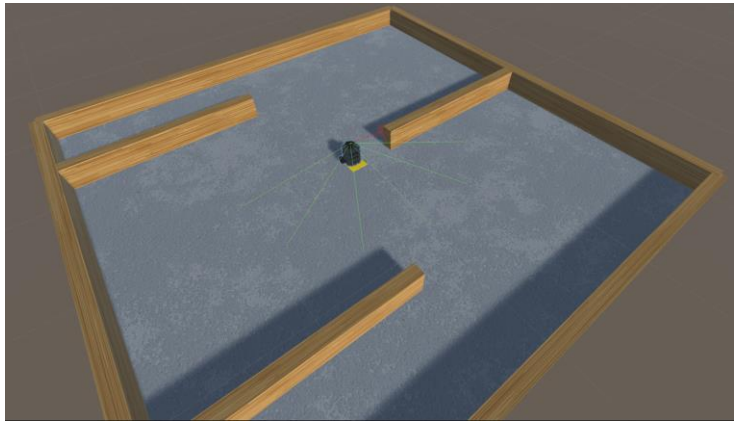
- Once you have set the training parameters you can run the training phase
  - Go to folder DQN
  - Run training script
    - *python training.py*
  - Click play button in Unity
- Now in Unity you will see the robot moving in the environment trying to reach the goal position

Exercise, load and test a trained model

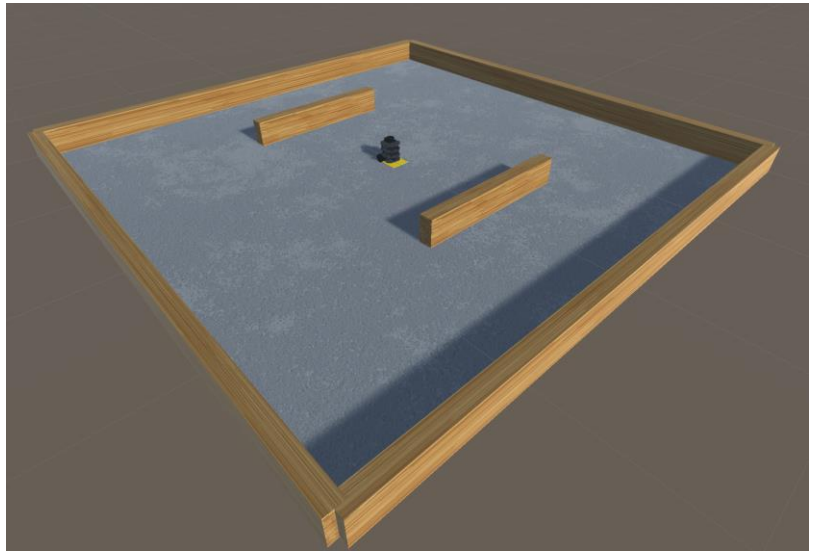
- When your training is finished you will find in folder DQN the model with the extension .h5
- To try your trained model, you have to modify the “testing.py”
  - add your trained model name, save and exit
- Run the script
  - *python testing.py*
- Do the same process with the pre-trained model that you can find in model_testing

Exercise, training and testing environments

Train environment



Testing environment



Exercise, testing on the simulated TB3 robot, set-up the state

- Now you have to integrate the model in a ROS2 node to execute the trained model on the simulated turtlebot3
- Modify the template node that we provide you
- In the ROS2 package you will find three main files:
  - turtlebot3DQN.py
  - agent.py
  - turtlebot3.py

Exercise, testing on the simulated TB3 robot, set-up the state

- In turtlebot3DQN.py you find the core of the node
  - The following functions are available:
    - Get the robot position (get_odom())
    - Get the goal info and the distance for the input of the network (get_goal_info())
    - Get the lidar information (get_scan())
    - Call the model passing the scan value, distance and angle
    - Get model action and apply it to the robot (move())

Exercise, testing on the simulated TB3 robot, set-up the state

- In agent.py you can find the definition of the network and the instantiation
- In turtlebot3.py you can find the function to get information and control the robot
  - These functions are useful to define the state and action of the robot
    - Lidar callback
    - Odometry information
    - Publisher for the velocity command

Exercise, testing on the simulated TB3 robot, set-up the state

- You have to implement the lidar callback (get_scan()) to get the lidar ranges as in the training phase in Unity.
- Select the lidar ranges as you have done in Unity.
  - If you used 3 rays per direction (7 total) in a max degree of (-90, 90) and you had 1 lidar every 30°, you have to extract the same number of rays in the laser callback function.

Exercise, testing on the simulated TB3 robot, set-up the state

- You have to implement the heading distance function.
- This function calculate the distance from the target to feed to the network and it is composed of:
  - The Euclidean distance between the robot pose and the target
  - The angle displacement between the robot pose and the target, this is obtained by using the atan2 function (to compute the direction towards the goal) and subtracting the current robot heading

Exercise, testing on the simulated TB3 robot, set-up the actions

- Implement the mapping between the actions and the robot movements
- For the trained network the actions are:
  - 0: Move forward for 5 cm (we suggest 0.2 m/s but you can change this value)
  - 1: 30° left rotation (we suggest 2.20 but you can change this value)
  - 2: 30° right rotation (we suggest - 2.20 but you can change this value)
- Note the target position is already fixed in the code

Exercise, testing on the simulated TB3 robot

- Run the Unity simulator
  - Open the old Unity project with turtlebot3 and ROS2
- Build, source and run your node
  - Colcon build
  - . install/setup.bash
  - ros2 run turtlebot3_DQN turtlebot3_DQN

Exercise, testing the trained model on the TB3

- To test a trained model on the real robot you have to define the target position
  - Measure the real distance in the arena between the target position and the initial robot position
  - Save the frame in the predefine variable (goal_x, goal_y)
- If everything is correct you can build and source the node
  - Colcon build and . Install/setup.bash
- Run bringup on the robot
- Run the node in your pc
  - ros2 run turtlebot3_DQN turtlebot3_DQN

References

- ML-Agent
  - https://github.com/Unity-Technologies/ml-agents
- Lidar message
  - https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html