COMP 542 Machine Learning

**Project Final Report**

**Title:** Simulated Autonomous Vehicle

**Team Member’s Names:** Subhobrata Chakraborty, Yvette Martinez

1. **Introduction:**

**1.1-** **Problems:**

Endowing robots with human-like abilities to perform motor skills in a smooth and natural way is one of the important goals of robotics. A promising way to achieve this is by creating robots that can learn new skills by themselves, similarly to humans. However, acquiring new motor skills is not simple and involves various forms of learning.

Over the years, the approaches for teaching new skills to robots have evolved significantly, and currently, there are three well-established types of approaches: direct programming, imitation learning and reinforcement learning. All of these approaches are still being actively used, and each one has its own advantages and disadvantages and is preferred for certain settings.

Reinforcement learning is the process of learning from trial-and-error, by exploring the environment and the robot’s own body. The goal in RL is specified by the *reward function*, which acts as positive reinforcement or negative punishment depending on the performance of the robot with respect to the desired goal. RL has created a well-defined niche for its application in robotics.

The main motivation for using reinforcement learning to teach robots new skills is that it offers three previously missing abilities:

* to learn new tasks, which even the human teacher cannot physically demonstrate or cannot directly program (e.g., jump three meters high, lift heavy weights, move very fast, *etc.*);
* to learn to achieve optimization goals of difficult problems that have no analytic formulation or no known closed form solution, when even the human teacher does not know what the optimum is, by using only a known cost function (e.g., minimize the used energy for performing a task or find the fastest gait, *etc.*);
* to learn to adapt a skill to a new, previously unseen version of a task (e.g., learning to walk from flat ground to a slope, learning to generalize a task to new previously unseen parameter values, *etc.*). Some imitation learning approaches can also do this, but in a much more restricted way (e.g., by adjusting parameters of a learned model, without being able to change the model itself).

Reinforcement learning also offers some additional advantages. For example, it is possible to start from a “good enough” demonstration and gradually refine it. Another example would be the ability to dynamically adapt to changes in the agent itself, such as a robot adapting to hardware changes—heating up, mechanical wear, growing body parts, *etc*.

**1.2- Related (prior) works:**

Several different works were used as references for our approach. Below is a list of the different scholarly papers and what each one’s focuses is on:

* **“Solving Markov Decision Processes via Simulation” by Abhijit Gosavi:** Presents an overview of simulation-based techniques for solving Markov decision problems/processes (MDP).
* **“A generalization of Bellman’s equation with application to path planning, obstacle avoidance and invariant set estimation” by Morgan Jones and Matthew M Peet:** In this paper, they utilize a generalization of Bellman’s equation, for a solution of a Multi-Stage Optimization Problem (MSOP), with a monotonically backward separable function.
* **“Optimal obstacle avoidance based on the Hamilton-Jacobi-Bellman equation” by S. Sundar and Z. Shiller:** This paper solves the online obstacle avoidance problem using the Hamilton-Jacobi-Bellman (HJB) theory.
* **“Applying Reinforcement Learning to Obstacle Avoidance” by Josh Beitelspacher :** This paper applies reinforcement learning techniques to an asteroid-type game. They used both Q-Learning and Sarsa to learn obstacle avoidance.
* **“Reinforcement Learning Algorithms in Global Path Planning for Mobile Robot” by V. N. Sichkar:** This paper is devoted to the research of two approaches for global path planning for mobile robots, based on Q-Learning and Sarsa algorithms.
* **“Scalable Decision Making with Sensor Occlusions for Autonomous Driving” M. Bouton, A. Nakhaei, K. Fujimura and M. J. Kochenderfer:** This paper demonstrates a decomposition method to bypass the computational cost of scaling the formulation to avoid multiple road users, leveraging the optimal avoidance strategy for a single user.

While there are several previous works that implement the methods we have chosen, what makes ours different is the combination of these methods to maximize the optimization of obstacle avoidance using reinforcement learning. For example, In Sundar and Shiller’s paper, while they only used reinforcement learning techniques Q-Learning and Sarsa for obstacle avoidance, in our method we train the drone using Bellman Equation Approach before, along with dynamic programming for the optimal path, the Monte Carlo algorithm for the optimal solution, Sarsa to updates the equation and lastly Q-learning so agent will be able to learn the maze environment. The same can be said about all the other works provided. While we were able to find individual methods for these approaches to obstacle avoidance, our method is a combination of these different techniques, making it unique.

**1.3- Brief approach and result:**

Firstly, we used the Bellman Equation Approach to train our drone.

### **Policy:** The agent's goal was to find/learn the optimal (here, shortest) way to the goal. The agent learns the policy (meaning a strategy) in order to succeed (maximize the reward) and the agent can collect from the environment. the agent in each state of the environment has a set of actions to choose from.

A **policy** is a mapping from a **state - s** to an **action - a** (the policy can also be deterministic or stochastic) :

### **State value function:** When the agent learns the policy, the most important thing for the agent is to evaluate the **state value**, which specifies how good it is for the agent to be in a particular state with the policy π. The agent that wants to learn the policy needs to follow some “common sense” and the MDP.

### **Action-value function (Q- function)**: Besides the agent being able to estimate the value of the state, the agent also needs to be aware of taking proper action in states under policy π. The state-value function helps to distinguish the value of each state in the environment, however, it does not provide the mechanisms to take the optimal action.

Then we used Dynamic Programming approach to find the optimal path for the drone

In order to use a DP for solving a given problem, **two main features** have to be fulfilled.

1) The main problem can be divided into **overlapping subproblems**. As we discussed above, the DP is mainly used when results of the same sub-problems are needed again and again (while an algorithm solves the main problem). In DP, computed sub-solutions, as we said, are in the memory so results don’t have to be recomputed when needed. It means that the DP is not useful when there are no common (overlapping) sub-problems because there is no point in storing the solutions if they are not needed again. A good example here is binary search algorithms, where it is not possible (there is no need) to divide the main problem (in this case, search) into common sub-problems.

2) **Optimal Substructure**. We can say that a given main problem has an optimal substructure feature if the optimal solution of the given problem can be obtained by using optimal solutions of its sub-problems. For example, the Shortest Path problem has the following optimal substructure property: If a node x lies on the shortest path from a source node A to destination node B, then the shortest path from A to B is a combination of the shortest path from A to x and shortest path from x to B.

We used Monte Carlo methods to find the optimal solution

**We use model-free RL**, so we will see how the agent can learn in order to be able to estimate the value of policies, similar to the policy evaluation method.

This is often called the **prediction problem** because we are estimating value functions, and these are defined as the expectation of future discounted rewards, that is, they contain values that depend on the future, so we are learning to predict the future in some sense.

Finally we used the Temporal Difference methods:

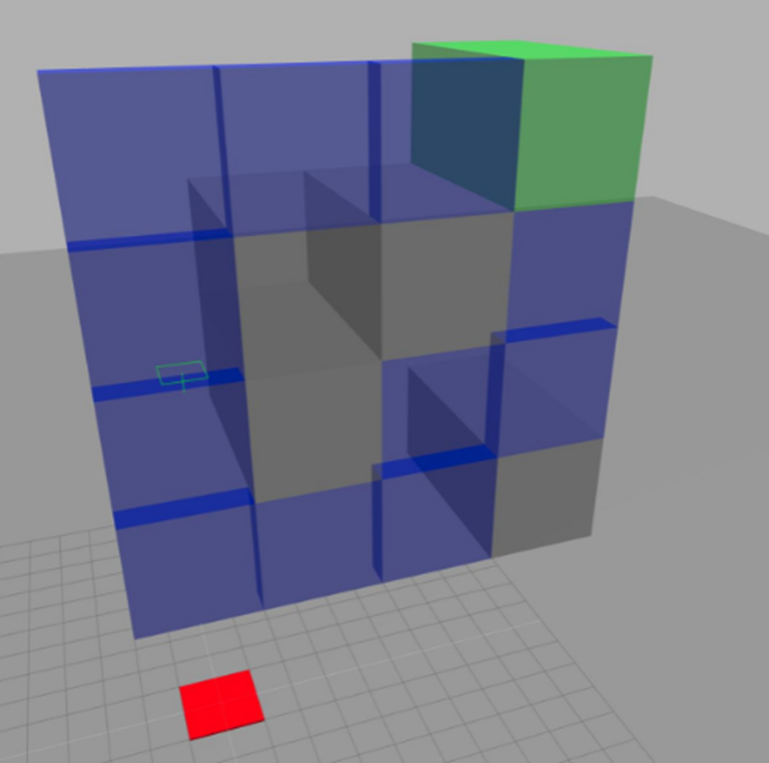
**SARSA and Q Learning**

In the beginning, we are going to go through the SARSA algorithm, which afterwards will be used in the ROS environment to learn the agent's brain (drone), to find a way to the goal.

Finally we implemented the Q learning algorithm so the agent will be able to learn the maze environment (learn Q-table) When the agent is finished with the learning process, the agent is able to traverse the maze without collision.

The only difference between **SARSA** and **Q-learning** is the action used in the target.

The mission for the drone (agent) is to fly throughout the wall (in blue). There are five tunnels (4 in gray and one in green), however only one (in green) is NOT CLOSED. The other 4 (in gray) are TRAPS. The drone starts from the place in red (see below)



1. **Background:**

We have used ROS Development Studio for the simulation and the environment.

**2.1- Language Used:**

Python

ROS

* Concept of subscribers and publishers
* Communication between each node
* Publishing of information form one node to another
* Subscribing of information from the parent node
* Multiple subscription of the same published information by multiple nodes
* Multiple publishing to a single or multiple nodes
* Server Client communication
* Basic troubleshooting of ros topics
* Rqt\_graph
* Tf
* Basic Robot kinematics

**2.2- ROS Version Used:**

ROS Noetic

**2.3- IDE Used:**

VS Code

**2.4- Simulator:**

Gazebo

**2.5- Reinforcement Learning:**

Markov Decision Process

Markov Chain

Dynamic Programming

Bellman Equation

Monte Carlo Methods

Temporal Difference Methods:

* SARSA
* Q-Learn

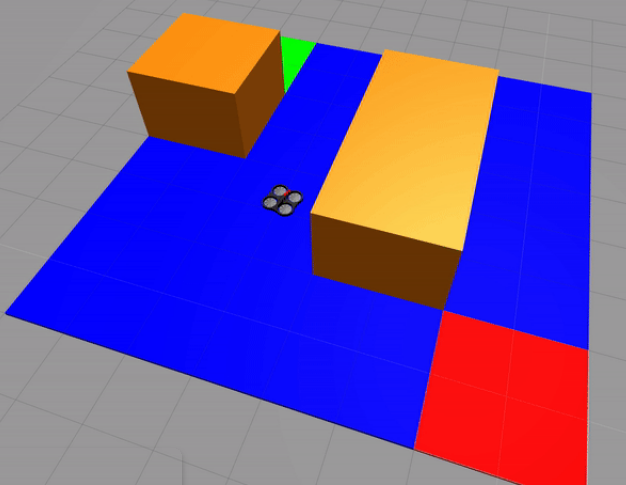
1. **Data and Model:**

**3.1- Dataset:**

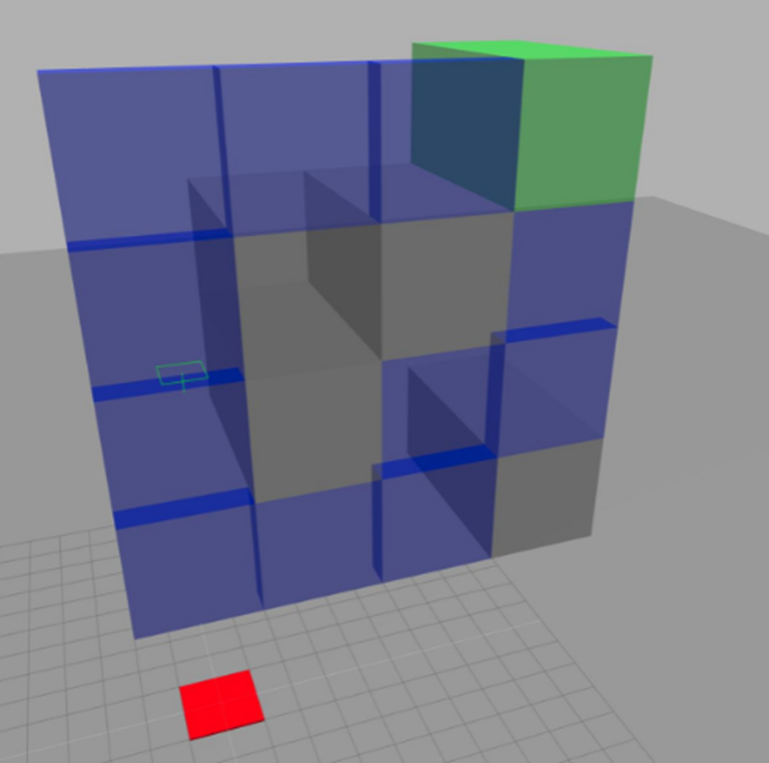
For this project we did not have any dataset. Instead, we used an environment from the ROS Development Studio.

Below is the environment that we used and the drone is supposed to land in the green zone.

Red zone is the starting point of the drone and the yellow objects are the obstacles.

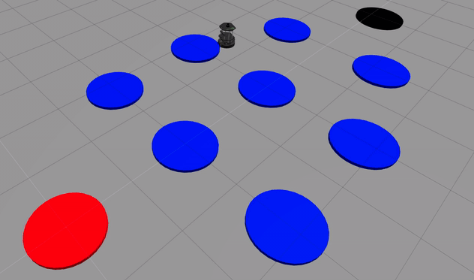


For the training of the drone we used the following environment:



The Red Zone marks the starting point of the robot. The grey zones are tunnels and marked as traps. If the drone flies into that zone it will be trapped. The successful outcome will be the one when the drone lands in the green zone.

In order to find the optimal path using dynamic programming, we used the following environment. The bot which we can see in the below image is called a turtlebot. The red zone is the starting point. Blue zones mark the steps to reach the goal. The goal is marked in black.



**3.2- Learning Models:**

### It is very important that in RL, the agent follows the optimal policy and maximizes the reward. The agent, as a rule, chooses the action to this state with the higher value. In this case, when the agent is in state A and the action 0 and 1 are available, the agent takes the action where the state has a higher value. In this case, the agent takes the action 1. The action-value table is updating while the agent learns the policy.

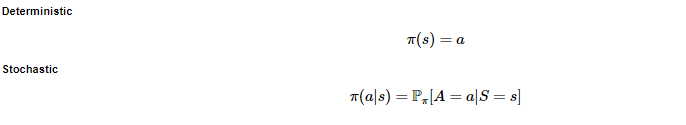
We are mainly going to focus on SARSA and Q-Learn algorithms. But before that we are going to explain how we decide to go with the temporal difference methods.

### **Bellman equation approach:**

### **Policy:**

While we consider the environment presented so far, like FL, the agent's goal was to find/learn the optimal (here, shortest) way to the goal. The agent learns the policy (meaning a strategy) in order to succeed (maximize the reward) and the agent can collect from the environment. The agent in each state of the environment has a set of action to choose from. The policy in this case can be associated with the function that maps the states to actions. It can be expressed as follows:

A **policy** is a mapping from a **state - s** to an **action - a** (the policy can also be deterministic or stochastic) :



### 

### **State value function:**

When the agent learns the policy, the most important thing for the agent is to evaluate the **state value**, which specifies how good it is for the agent to be in a particular state with the policy. The agent that wants to learn the policy needs to follow some “common sense” and the MDP. It only depends on time and the discount factor. We can combine these two components and determine the new mapping function **state value function**, which defines the value of a state value **s** when the agent follows policy. As we previously noticed, it gives the overview of how good it is for the agent in that state, which is measured by expected return following the policy. We can condensate our discourse as follows:



where E is the expectation over policy pi.

Value function describes how valuable it is to be in a specific state s under a certain policy π.

If we consider the **return** function, we can rewrite the above equation as follows:



### **Action-value function (Q- function):**

Besides the agent being able to estimate the value of the state, the agent also needs to be aware of taking proper action **a** in state **s** under policy π. The state-value function helps to distinguish the value of each state in the environment, however, it does not provide the mechanisms to take the optimal action.

On the other hand, we can say that the action-value function specifies how good it is for the agent to perform a particular action **a** in a state **s** with the policy π.The action-value function is also known as a **Q function**. The agent has to recognize the actions to take (under the same policy) and improve the policies the agent follows.

Similar to the state-value function, we can define the action function as follows:



### **Bellman Equation:**

An important recursive property emerges for both **state-value** and **action-value** functions if we expand them.

### 

### 

### 

### 

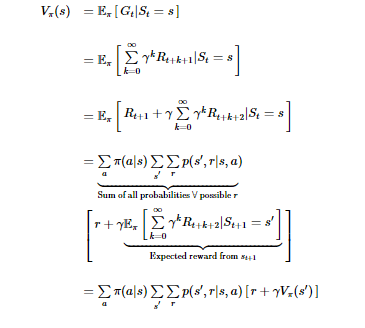
### 

### 

### 

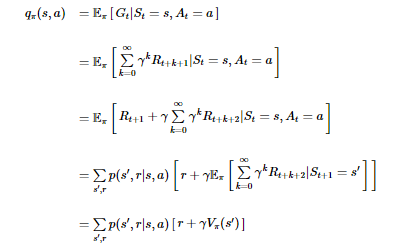
### 

### **State - Value Function, Bellman Equation**



### **Action - value function. Bellman Equation for action values.**

Similarly, we can do the same for the Q function



Dynamic Programming:

In order to use a DP for solving a given problem, **two main features** have to be fulfilled.

1) The main problem can be divided into **overlapping subproblems**. As we discussed above, the DP is mainly used when results of the same sub-problems are needed again and again (while an algorithm solves the main problem). In DP, computed sub-solutions, as we said, are in the memory so results don’t have to be recomputed when needed. It means that the DP is not useful when there are no common (overlapping) sub-problems because there is no point in storing the solutions if they are not needed again. A good example here is binary search algorithms, where it is not possible (there is no need) to divide the main problem (in this case, search) into common sub-problems.

2) **Optimal Substructure**. We can say that a given main problem has an optimal substructure feature if the optimal solution of the given problem can be obtained by using optimal solutions of its sub-problems. For example, the Shortest Path problem has the following optimal substructure property: If a node x lies on the shortest path from a source node A to destination node B, then the shortest path from A to B is a combination of the shortest path from A to x and shortest path from x to B.

Policy Iteration Algorithm Steps:

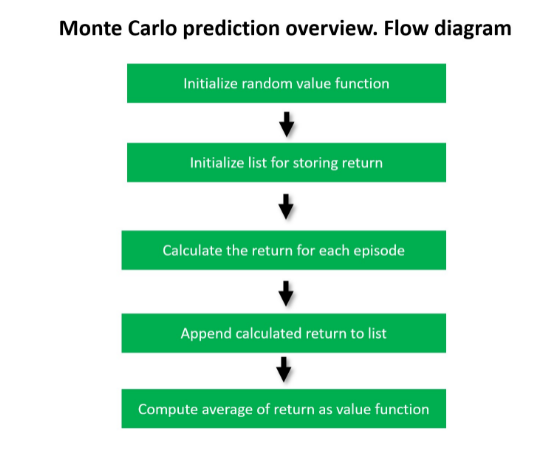
* Initialize random policy
* For each state, Q(S,A) is calculated
* Update of action (max, Q(S,A)) for each state
* Check if improvement is needed
* If yes, repeat steps 2 to 4

**Monte Carlo Methods:**

In our discussion about RL, we often refer to reward, which can be associated with one-step payments given to the agent for the action that the agent performs in the state. **Here we need to highlight that the reward is the core of RL, but it is not what the agent is trying to maximize! In RL, the agent is trying to maximize the return, which refers to the total discounted rewards!** Returns (or cumulative rewards) are calculated from any state and usually go until the end of the episode that the agent performs. Return is a better factor of agent performance since it contains a long-term sequence of rewards (discounted). This is, however, a general RL attitude, which refers to the expectation of returns. That means that the agent wants high returns, but is high in expectation (on average). So, if the agent explores (takes an action), it is in a very unexpected environment, so the agent tries to maximize the expected total discounted reward: **value functions** (which as we remember is a the way we can estimate the value of the state - the value indicates how much reward the agent can expect from a state until the end of an episode.).

As we specified above, the agent now interacts with the environment without MDP (without defined transitions to environment states. The goal for the RL agent is to estimate the value of a policy. The agent needs to know how much total reward is expected if the agent follows a certain policy. Our first idea to tackle the given problem (estimate the policy value) was to follow the average approach. “Averaging” exactly means (in this case) to run several episodes with a given policy, collecting hundreds of trajectories.

The method we discuss now and which we are going to deploy on our next practical example is called **Monte-Carlo prediction (MC).**



**Temporal Difference Methods:**

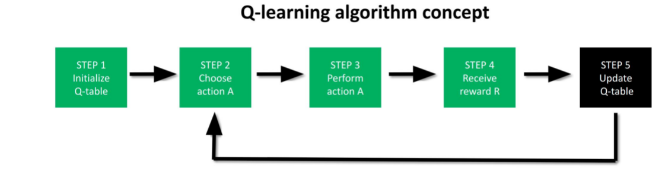
Similar to the Monte Carlo methods, the TD methods provide two approaches: passive **TD prediction** and active **TD control**. In our case, the passive approach is connected with the fact that the agent has a policy π that follows this policy and learns the value function V(s) for this specific policy π. We also define the active approach - **control** where the agent is familiar about the state availability. The agent learns the environment by interaction without also having a policy (like for a passive approach). Now we will discuss more closely with both of these approaches related to the TD method.

Similar to what we did in Monte Carlo prediction, in TD prediction we try to predict the state values. In Monte Carlo prediction, we estimate the value function by simply taking the mean return. But in TD learning, for the update following the general update rule is used:



To be consistent, the next type algorithms (SARSA and Q-learning) work as follows:

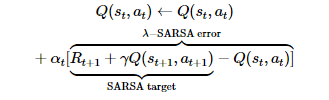
1. We first initialize our Q-table by assigning a value of 0 to all entries.
2. Next, the agent chooses an action, from the table the agent initiates and updates.
3. The agent performs the action.
4. The agent receives the reward.
5. The Q table is updated (how ? ==> Depending on applicability of the algorithm we are going to discuss)



## **SARSA. Improving policies after each step:**

The SARSA algorithm was introduced in 1994 by Rummery and Niranjan in the article “On-Line Q-Learning Using Connectionist Systems.” Our goal is to update the agent policy after each step (action the agent takes). We remember our **Q function**, which we used to estimate the vest policy (**NOTE: it is highly recommended that you take a look at the previous module of this course and update information about the Q-table**).

The Q-function requires a state-action pair as input. The TD algorithm for control is straightforward, so give a look at the update rule:



As you remember, in the previous module (MC methods), we discussed the Q-table. However, the mechanism before is slightly different from the current one. In order to complete our discussion about the SARSA approach and get important intuition behind the algorithm, we considered again a single episode of an agent moving in a world.

The robot starts at s0 and after seven visits, it reaches a terminal state at s5. For each state, we have an associated action. Moving forward, the algorithm takes into account only the state at t and t+1. In the standard implementation of SARSA, the previous states are ignored, as shown by the shadow on top of them in the graphical illustration. This is in line with the TD framework as explained in the TD(0) section.

Now let us summarize all the steps of the **SARSA** algorithm:

1. First, we initialize the Q values to some arbitrary values.
2. We select an action by the epsilon-greedy policy (ϵ>0) and move the agent from one state to another
3. We update the Q value's previous state by following the update rule (simplified above equation):



where a' is the action selected by an epsilon-greedy policy

## 

## 

## **Q-Learning: Learning to act optimally:**

## **The SARSA algorithm we used in the previous paragraph is the ON-policy learning.**

The agent learns the same policy it uses for generating experience, like a human being who tries to be self-sufficient and does not accept any other experiences and learns only on his own experiences (successes and failures). However, it seems to be better to apply the strategy when we can also learn from the mistakes or “successful roads” from the others. Here we can apply the **OFF–policy** learning, which can be described as a sort of “learning from others”. The most important thing is to highlight one more time that wWhen the agent learns the off-policy, the agent learns about a policy that is different from the policy generating experiences.

In off-policy learning, there are two policies: a **behavior policy**, which is used by the agent to generate experiences, to interact with the environment, and second: a **target policy**, which is the policy the agent is learning about.

The **Q-learning algorithm** is considered the most important mechanism in the whole Reinforcement Learning framework.

**SARSA Algorithm and Q Learning Algorithm Code:**

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**import** **time**

**import** **random**

**from** **IPython.display** **import** clear\_output, display

ACTIONS = ['up', 'down', 'left', 'right']

LENGTH = **None**

N\_STATES = **None**

START = **None**

HOLE1 = **None**

HOLE2 = **None**

TERMINAL = **None**

EPSILON = **None**

MAX\_EPISODE = **None**

GAMMA = **None**

ALPHA = **None**

FIRST = **True**

*# Initial Q-Table*

**def** build\_q\_table():

**global** N\_STATES

**global** ACTIONS

table = pd.DataFrame(

np.zeros((N\_STATES, len(ACTIONS))),

columns=ACTIONS

)

**return** table

*# Actor*

*# ε-Greedy*

**def** actor(observation, q\_table):

*#self.check\_state\_exist(observation)*

*# action selection*

**if** np.random.uniform() < EPSILON:

*# choose best action*

state\_action = q\_table.loc[observation, :]

*# print("####")*

*# print(state\_action)*

*# some actions may have the same value, randomly choose on in these actions*

action = np.random.choice(state\_action[state\_action == np.max(state\_action)].index)

**else**:

*# choose random action*

action = np.random.choice(ACTIONS)

**return** action

*# Enviroment Visual*

**def** update\_env(state, episode, step):

view = np.array([['\_ '] \* LENGTH] \* LENGTH)

view[tuple(TERMINAL)] = '\* '

view[HOLE1] = 'X '

view[HOLE2] = 'X '

view[tuple(state)] = 'o '

interaction = ''

**for** v **in** view:

interaction += ''.join(v) + '**\n**'

*# Enviroment Feedback*

**def** init\_env():

**global** HOLE1

**global** HOLE2

**global** FIRST

**global** START

**global** TERMINAL

start = START

HOLE1 = (1,2)

HOLE2 = (2,2)

FIRST = **False**

**return** start, **False**

**def** get\_env\_feedback(state, action):

reward = 0.

end = **False**

a, b = state

**if** action == 'up':

a -= 1

**if** a < 0:

a = 0

next\_state = (a, b)

**if** next\_state == TERMINAL:

reward = 1.

end = **True**

**elif** (next\_state == HOLE1) **or** (next\_state == HOLE2):

reward = -1.

end = **True**

**elif** action == 'down':

a += 1

**if** a >= LENGTH:

a = LENGTH - 1

next\_state = (a, b)

**if** (next\_state == HOLE1) **or** (next\_state == HOLE2):

reward = -1.

end = **True**

**elif** action == 'left':

b -= 1

**if** b < 0:

b = 0

next\_state = (a, b)

**if** (next\_state == HOLE1) **or** (next\_state == HOLE2):

reward = -1.

end = **True**

**elif** action == 'right':

b += 1

**if** b >= LENGTH:

b = LENGTH - 1

next\_state = (a, b)

**if** next\_state == TERMINAL:

reward = 1.

end = **True**

**elif** (next\_state == HOLE1) **or** (next\_state == HOLE2):

reward = -1.

end = **True**

*#print("::next ::", next\_state, " action ::: ", action)*

**return** next\_state, reward, end

**def** playGame(q\_table):

maze\_transitions = []

state = (3,0)

end = **False**

LENGTH = 4

a, b = state

i = 0

**while** **not** end:

*#a, b = state*

*#print("state ::", state)*

act = actor(a \* LENGTH + b, q\_table)

print("step::", i ," action ::", act)

maze\_transitions.append(act)

next\_state, reward, end = get\_env\_feedback(state, act)

state = next\_state

a, b = state

i += 1

print("==> Game Over <==")

**return** maze\_transitions

*## following function replaces name of actions (string) to number (int)*

*## function is used for ther control motion of the drone, NOT for the SARSA algorithm!*

**def** droneActions(maze\_transitions):

actions = []

**for** action **in** maze\_transitions:

**if** action == 'up':

actions.append(0)

**if** action == 'down':

actions.append(1)

**if** action == 'right':

actions.append(2)

**if** action == "left":

actions.append(3)

**return** actions

*## function is used for the control motion of the drone, NOT for the SARSA algorithm!*

*## function replaces sequences of motion commands 0,1,2 or 3 to commands -1, 0, 1 depending of*

*## actual drone heading and next action which has to be taken by drone*

*## if the drone heading is 0 (drone flies UP) and next coming action is 0 so no action is taken*

*## if the drone heading is 0 and next coming action is 2 so the action will be -1*

*## we can say the control system receives in each state 3 command: continue the direction you fly -action 0*

*## turn left 1 or right -1*

**def** droneMotions(drone\_actions):

pos\_drone = 0

head = [pos\_drone] + drone\_actions

drone\_move = []

**for** i **in** range(len(head)-1):

**if** head[i] == head[i+1]:

drone\_move.append(0)

**if** head[i] != head[i+1]:

**if** ((head[i] == 0) **or** (head[i] == 1)):

**if** head[i+1] == 3:

drone\_move.append(1)

**if** head[i+1] == 2:

drone\_move.append(-1)

**if** ((head[i] == 2) **or** (head[i] == 3)):

**if** head[i+1] == 0:

drone\_move.append(1)

**if** head[i+1] == 1:

drone\_move.append(-1)

**return** drone\_move

*# Learn*

*######learn SARSA algorithm*

**def** learnSARSA():

*#build the Q-table (see the definition of the build\_q\_table() function )*

q\_table = build\_q\_table()

episode = 0

*#main learing loop*

**while** episode < MAX\_EPISODE:

state, end = init\_env()

step = 0

*#update the environment after each episode*

update\_env(state, episode, step)

*# take a position of first state (start state)*

a, b = state

*# based on start state and actual Q-table agent take a action (by calling method actor() )*

act = actor(a \* LENGTH + b, q\_table)

*#when the agent does not win or drop into the HOLE run this loop*

**while** **not** end:

*#agent took a action: act and the agent receives the feedback from environment:*

*#next\_state, reward, end*

next\_state, reward, end = get\_env\_feedback(state, act)

*#position of next state the agent transits (state ==> next\_state ::: a,b ==> a\_, b\_*

a\_, b\_ = next\_state

*#agent takes new action based on new target state*

act\_ = actor(a\_ \* LENGTH + b\_, q\_table)

*# the agent takes predicted values from Q-table*

q\_predict = q\_table.loc[a \* LENGTH + b, act]

*# agent estimates the value of Q target (the agent is still in a,b but the agent*

*#computes new state a\_, b\_)*

**if** next\_state != TERMINAL:

q\_target = reward + GAMMA \* q\_table.loc[a\_ \* LENGTH + b\_, act\_]

**else**:

q\_target = reward

*######################################################*

*### SARSA - compare with above formula (above cell)###*

*######################################################*

q\_table.loc[a \* LENGTH + b, act] += ALPHA \* (q\_target - q\_predict)

*#agent formaly transits to new state and environment update*

state = next\_state

act = act\_

a, b = state

step += 1

update\_env(state, episode, step)

*#print("step", step)*

**if** step > 30: *# feel free to change this parameter*

*#print("END")*

end = **True**

episode += 1

**return** q\_table

*###################################### END OF SARSA*

LENGTH = 4

N\_STATES = LENGTH \* LENGTH

START = (LENGTH - 1, 0)

TERMINAL = (0,3)

EPSILON = .9

MAX\_EPISODE = 200 *## Feel free to udjust*

GAMMA = .9

ALPHA = .01

q\_table = learnSARSA()

print("====== Q TABLE AFTER LEARNING ======")

print(q\_table)

print(" ")

print("======ACTION TAKEN BY AGENT TO REACH THE GOAL======")

maze\_transitions = playGame(q\_table)

actions = droneActions(maze\_transitions)

*### Qlearning algoritm*

**def** Qlearn():

q\_table = build\_q\_table()

episode = 0

**while** episode < MAX\_EPISODE:

state, end = init\_env()

step = 0

update\_env(state, episode, step)

**while** **not** end:

a, b = state

act = actor(a \* LENGTH + b, q\_table)

next\_state, reward, end = get\_env\_feedback(state, act)

na, nb = next\_state

q\_predict = q\_table.loc[a \* LENGTH + b, act]

**if** next\_state != TERMINAL:

*### Qlearning algoritm*

*###################################################################*

q\_target = reward + GAMMA \* q\_table.iloc[na \* LENGTH + nb].max()

**else**:

q\_target = reward

q\_table.loc[a \* LENGTH + b, act] += ALPHA \* (q\_target - q\_predict)

state = next\_state

step += 1

update\_env(state, episode, step)

*#if step > 30: # the same punish like presented in previous module. Choose to apply or not*

*# end = True*

episode += 1

**return** q\_table

*##### END OF Q LEARNING*

**4. Experiments and Analysis of Results**

We tried implementing several algorithms before we finally decided to use SARSA and Q-Learning for our project.

Training of the drone:

#! /usr/bin/env python

import rospy

import time

import pandas as pd

import numpy as np

from std\_msgs.msg import Empty

from geometry\_msgs.msg import Twist

from gazebo\_connection import GazeboConnection

#seting environemt and training parameters

N\_STATES = 5 # states of environment (available position of the drone)

ACTIONS = ['fly\_left', 'fly\_right'] # available actions in dron environment

EPSILON = 0.5 # greedy policy

ALPHA = 0.1 # learning rate

GAMMA = 0.9 # discount factor

EPISODES = 8 # number episodes which the drone is going to "play"

TRANSITION\_TIME = 0.1 # transition time from one state to other

gazebo = GazeboConnection()

"""

Q table - the simple "brain" of the drone.

During the training the Q table is going to be updated

"""

def build\_q\_table(n\_states, actions):

table = pd.DataFrame(

np.zeros((n\_states, len(actions))), # q\_table initial values

columns=actions, # actions's name

)

return table

"""

For each of the state the drone is, the brain of the drone (agent) decides

about the action to take - based on updated Q table.

"""

def choose\_action(state, q\_table):

# This is how to choose an action

state\_actions = q\_table.iloc[state, :]

if (np.random.uniform() > EPSILON) or ((state\_actions == 0).all()):

action\_name = np.random.choice(ACTIONS)

else: # act greedy

action\_name = np.argmax(state\_actions)

return action\_name

"""

Feedback from the environment.

The drone (agent) receives the rewards 0 or 1.

"""

def get\_env\_feedback(S, A):

# This is how agent will interact with the environment

if A == 'fly\_right':

if S == N\_STATES - 2:

S\_ = 'goal'

R = 1

else:

S\_ = S + 1

R = 0

if A == 'fly\_left':

R = 0

if S == 0:

S\_ = S

else:

S\_ = S - 1

return S\_, R

"""

Update of the environent.

Environment/drone transits to other state.

Update of the terminal.

"""

def update\_env(S, episode, step\_counter):

# This is how environment be updated

env\_list = ['\_']\*(N\_STATES-1) + ['Goal']

if S == 'goal':

interaction = 'Episode :: %s:: total\_steps = %s' % (episode+1, step\_counter)

print('\r{}'.format(interaction))

time.sleep(2)

print('\r ')

gazebo.resetSim()

else:

env\_list[S] = 'X'

interaction = ''.join(env\_list)

print('\r{}'.format(interaction))

time.sleep(TRANSITION\_TIME)

"""

Class for definition of drone movements.

Learning function implementation (Bellman equation).

"""

class MoveSquareClass(object):

def \_\_init\_\_(self):

self.ctrl\_c = False

self.rate = rospy.Rate(10)

def publish\_once\_in\_cmd\_vel(self, cmd):

"""

This is because publishing in topics sometimes fails teh first time you publish.

In continuos publishing systems there is no big deal but in systems that publish only

once it IS very important.

"""

while not self.ctrl\_c:

connections = self.\_pub\_cmd\_vel.get\_num\_connections()

if connections > 0:

self.\_pub\_cmd\_vel.publish(cmd)

#rospy.loginfo("Publish in cmd\_vel...")

break

else:

self.rate.sleep()

# function that stops the drone from any movement

def stop\_drone(self):

#rospy.loginfo("Stopping...")

self.\_move\_msg.linear.x = 0.0

self.\_move\_msg.angular.z = 0.0

self.publish\_once\_in\_cmd\_vel(self.\_move\_msg)

# function that makes the drone turn 90 degrees

def turn\_drone(self):

#rospy.loginfo("Turning...")

self.\_move\_msg.linear.x = 0.0

self.\_move\_msg.angular.z = 1.0

self.publish\_once\_in\_cmd\_vel(self.\_move\_msg)

# function that makes the drone move forward

def move\_forward\_drone(self):

#rospy.loginfo("Moving forward...")

self.\_move\_msg.linear.y = 0.5

self.\_move\_msg.angular.z = 0.0

self.publish\_once\_in\_cmd\_vel(self.\_move\_msg)

#self.stop\_drone()

#time.sleep(10)

def move\_forward\_drone\_opposite(self):

#rospy.loginfo("Moving opposite...")

self.\_move\_msg.linear.y = -0.5

self.\_move\_msg.angular.z = 0.0

self.publish\_once\_in\_cmd\_vel(self.\_move\_msg)

#self.stop\_drone()

#time.sleep(10)

"""

Implementation of learning process for the drone(agent).

"""

def learn(self):

# helper variables

r = rospy.Rate(1)

# define the different publishers and messages that will be used

self.\_pub\_cmd\_vel = rospy.Publisher('/cmd\_vel', Twist, queue\_size=1)

self.\_move\_msg = Twist()

self.\_pub\_takeoff = rospy.Publisher('/drone/takeoff', Empty, queue\_size=1)

self.\_takeoff\_msg = Empty()

self.\_pub\_land = rospy.Publisher('/drone/land', Empty, queue\_size=1)

self.\_land\_msg = Empty()

q\_table = build\_q\_table(N\_STATES, ACTIONS)

for episode in range(EPISODES):

# make the drone takeoff

i=0

while not i == 3:

self.\_pub\_takeoff.publish(self.\_takeoff\_msg)

time.sleep(1)

i += 1

# define the seconds to move in each side of the square (which is taken from the goal) and the seconds to turn

sideSeconds = 1

turnSeconds = 1.8

step\_counter = 0

S = 0

is\_terminated = False

update\_env(S, episode, step\_counter)

#########################

### LEARNING PROCESS ####

#########################

while not is\_terminated:

A = choose\_action(S, q\_table)

if A == 'fly\_right':

self.move\_forward\_drone()

time.sleep(sideSeconds)

if A == 'fly\_left':

self.move\_forward\_drone\_opposite()

time.sleep(sideSeconds)

S\_, R = get\_env\_feedback(S, A) # take action & get next state and reward

q\_predict = q\_table.loc[S, A]

if S\_ != 'goal':

q\_target = R + GAMMA \* q\_table.iloc[S\_, :].max() # next state is not terminal

else:

q\_target = R # next state is terminal

is\_terminated = True # terminate this episode

#########################

### BELLMAN EQUATION ###

#########################

q\_table.loc[S, A] += ALPHA \* (q\_target - q\_predict)

S = S\_ # move to next state

update\_env(S, episode, step\_counter+1)

step\_counter += 1

gazebo.resetSim()

self.stop\_drone()

i=0

while not i == 3:

self.\_pub\_land.publish(self.\_land\_msg)

time.sleep(1)

i += 1

return q\_table

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('move\_square')

move\_square = MoveSquareClass()

try:

q\_table = move\_square.learn()

print('\rQ(a,s)::')

print('-----------------')

print(q\_table)

except rospy.ROSInterruptException:

pass

Output:

rosrun fly\_drone train\_drone.py

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 1:: total\_steps = 4

X\_\_\_Goal

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

/usr/local/lib/python2.7/dist-packages/numpy/core/fromnumeric.py:56: FutureWarning: 'argmax' is deprecated. Use 'idxmax' instead. The behavior of 'argmax' will be corrected to return the positional maximum in the future. Use 'series.values.argmax' to get the position of the maximum now.

return getattr(obj, method)(\*args, \*\*kwds)

Episode :: 2:: total\_steps = 9

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 3:: total\_steps = 4

X\_\_\_Goal

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 4:: total\_steps = 7

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

\_\_X\_Goal

\_X\_\_Goal

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 5:: total\_steps = 10

X\_\_\_Goal

\_X\_\_Goal

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 6:: total\_steps = 6

X\_\_\_Goal

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 7:: total\_steps = 5

X\_\_\_Goal

\_X\_\_Goal

\_\_X\_Goal

\_\_\_XGoal

Episode :: 8:: total\_steps = 4

Q(a,s)::

-----------------

fly\_left fly\_right

0 0.000326 0.008036

1 0.000245 0.046136

2 0.000783 0.197047

3 0.012005 0.569533

4 0.000000 0.000000

Simulation is provided in the presentation.

Reinforcement Learning is a very distinct area of ML, which defines the Agent (in our case, the robot or application) and the environment (the area where the Agent acts). The Agent performs certain actions/interacts with the environment (for example, movement – according to policy) to maximize the reward. The foundation of optimal behavior of the Agent is defined by the Bellman equation, which is a widely used method for solving practical optimization problems. To solve the Bellman optimality equation, we use dynamic programming. The learning process of the Agent is performed in the environment, giving the Agent a reward if the action the Agent performs is correct or punishes it if the action is wrong. The Agent performing a number of such episodes can automatically create the Agent's knowledge about the environment (rules) and how to act in order to maximize the reward (return).

Q-Learning is a Reinforcement Learning technique that works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. One of the strengths of Q-Learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Reinforcement Learning is an approach where the agent needs no teacher to learn how to solve a problem. The only signal used by the agent to learn from his actions in a reinforcement environment is the so-called reward, a number which tells the agent if his last action was good (or) not. Q-Learning is a recent form of Reinforcement Learning algorithm that does not need a model of its environment and can be used on-line.

* Q-learning directly learns the optimal policy, whilst SARSA learns a near-optimal policy whilst exploring. If you want to learn an optimal policy using SARSA, then you will need to decide on a strategy to decay ϵϵ in ϵϵ-greedy action choice, which may become a fiddly hyperparameter to tune.
* Q-learning (and off-policy learning in general) has higher per-sample variance than SARSA, and may suffer from problems converging as a result. This turns up as a problem when training neural networks via Q-learning.
* SARSA will approach convergence *allowing* for possible penalties from exploratory moves, whilst Q-learning will ignore them. That makes SARSA more conservative - if there is risk of a large negative reward close to the optimal path, Q-learning will tend to trigger that reward whilst exploring, whilst SARSA will tend to avoid a dangerous optimal path and only slowly learn to use it when the exploration parameters are reduced. The classic toy problem that demonstrates this effect is called **cliff walking.**

In robotics, the ultimate goal of reinforcement learning is to endow robots with the ability to learn, improve, adapt and reproduce tasks with dynamically changing constraints based on exploration and autonomous learning. We give a summary of the state-of-the-art of reinforcement learning in the context of robotics, in terms of both algorithms and policy representations. Numerous challenges faced by the policy representation in robotics are identified.

Applying reinforcement learning in robotics demands safe exploration which becomes a key issue of the learning process, a problem often neglected in the general reinforcement learning community (due to the use of simulated environments).

While learning, the dynamics of a robot can change due to many external factors ranging from temperature to wear thereby the learning process may never fully converge (i.e. how light conditions affect the performance of the vision system and, as a result, the task’s performance). This problem makes comparing algorithms particularly hard.

Reinforcement learning algorithms are implemented on a digital computer where the discretization of time is unavoidable despite that physical systems are inherently continuous-time systems. Time discretization of the actuation can generate undesirable artifacts (e.g., the distortion of distance between states) even for idealized physical systems, which cannot be avoided.

In order for robot reinforcement learning to leverage good results the following principles should be taken into account:

* Effective representations
* Approximate models
* Prior knowledge or information

Much of the success of reinforcement learning methods has been due to the clever use of approximate representations. The need of such approximations is particularly pronounced in robotics, where table-based representations are rarely scalable.

* Smart State-Action discretization: Reducing the dimensionality of states or actions by smart state-action discretization is a representational simplification that may enhance both policy search and value function-based methods.
* Value Function Approximation: A value function-based approach requires an accurate and robust but general function approximator that can capture the value function with sufficient precision while maintaining stability during learning (e.g. ANNs).
* Pre-structured policies: Policy search methods require a choice of policy representation that controls the complexity of representable policies to enhance learning speed.

**Analysis:**

**Output of our Code Mentioned in Models Section:**

user:~$ rosrun fly\_drone drone\_fly\_qlearn.py

('step::', 0, ' action ::', 'right')

('step::', 1, ' action ::', 'right')

('step::', 2, ' action ::', 'up')

('step::', 3, ' action ::', 'up')

('step::', 4, ' action ::', 'up')

('step::', 5, ' action ::', 'right')

==> Game Over <==

('maze\_transitions ::', [2, 2, 0, 0, 0, 2])

('drone motion ::', [-1, 0, 1, 0, 0, -1])

[INFO] [1638321526.930358, 0.000000]: Taking off...

[INFO] [1638321527.931817, 1757.805000]: Taking off...

[INFO] [1638321528.933401, 1758.801000]: Turning...

[INFO] [1638321528.934674, 1758.802000]: Publish in cmd\_vel...

[INFO] [1638321530.436522, 1760.299000]: Moving forward...

[INFO] [1638321530.437273, 1760.300000]: Publish in cmd\_vel...

[INFO] [1638321533.741454, 1763.581000]: Moving forward...

[INFO] [1638321533.742082, 1763.581000]: Publish in cmd\_vel...

[INFO] [1638321537.046439, 1766.868000]: Turning...

[INFO] [1638321537.047331, 1766.868000]: Publish in cmd\_vel...

[INFO] [1638321538.548980, 1768.364000]: Moving forward...

[INFO] [1638321538.550222, 1768.366000]: Publish in cmd\_vel...

[INFO] [1638321541.855272, 1771.649000]: Moving forward...

[INFO] [1638321541.855775, 1771.649000]: Publish in cmd\_vel...

[INFO] [1638321545.160282, 1774.937000]: Moving forward...

[INFO] [1638321545.161111, 1774.938000]: Publish in cmd\_vel...

[INFO] [1638321548.465242, 1778.214000]: Turning...

[INFO] [1638321548.465997, 1778.215000]: Publish in cmd\_vel...

[INFO] [1638321549.967386, 1779.712000]: Moving forward...

[INFO] [1638321549.968093, 1779.713000]: Publish in cmd\_vel...

[INFO] [1638321553.270890, 1783.002000]: Stopping...

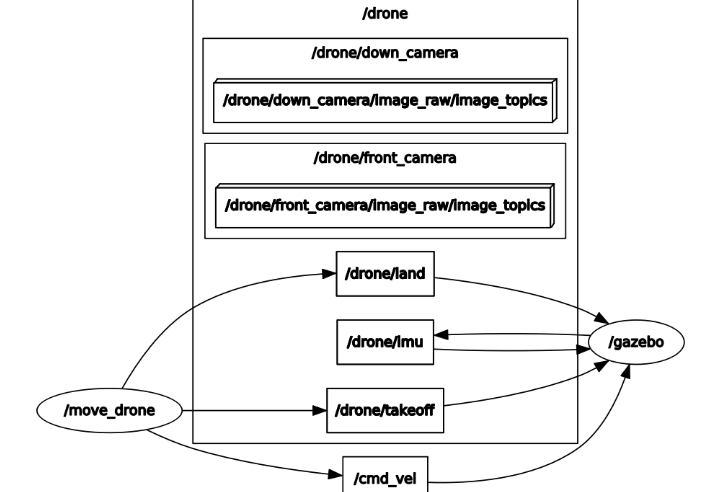
[INFO] [1638321553.271479, 1783.002000]: Publish in cmd\_vel...

[INFO] [1638321553.272523, 1783.003000]: Landing...

[INFO] [1638321554.273864, 1783.996000]: Landing...

[INFO] [1638321555.275793, 1784.996000]: Landing...

**Rqt Graph for the final Q-Learn Algorithm Output:**

****

**Influence of variables**

**Learning Rate:**

The learning rate or step size determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities). In fully deterministic environments, a learning rate of alpha \_{t}=1 is optimal. When the problem is stochastic, the algorithm converges under some technical conditions on the learning rate that require it to decrease to zero. In practice, often a constant learning rate is used, such as alpha \_{t}=0.1 for all t

**Discount factor:**

The discount factor gamma determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e.r\_{t} (in the update rule above), while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge. For gamma =1, without a terminal state, or if the agent never reaches one, all environment histories become infinitely long, and utilities with additive, undiscounted rewards generally become infinite. Even with a discount factor only slightly lower than 1, Q-function learning leads to propagation of errors and instabilities when the value function is approximated with an artificial neural network. In that case, starting with a lower discount factor and increasing it towards its final value accelerates learning.

**Initial conditions (Q0):**

Since Q-learning is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. High initial values, also known as "optimistic initial conditions",can encourage exploration: no matter what action is selected, the update rule will cause it to have lower values than the other alternative, thus increasing their choice probability. The first reward r can be used to reset the initial conditions.According to this idea, the first time an action is taken the reward is used to set the value of Q. This allows immediate learning in case of fixed deterministic rewards. A model that incorporates reset of initial conditions (RIC) is expected to predict participants' behavior better than a model that assumes any arbitrary initial condition (AIC). RIC seems to be consistent with human behaviour in repeated binary choice experiments.

**Implementation:**

Q-learning at its simplest stores data in tables. This approach falters with increasing numbers of states/actions since the likelihood of the agent visiting a particular state and performing a particular action is increasingly small.

**Function approximation:**

Q-learning can be combined with function approximation.This makes it possible to apply the algorithm to larger problems, even when the state space is continuous.

One solution is to use an (adapted) artificial neural network as a function approximator. Function approximation may speed up learning in finite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

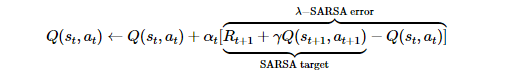
**Quantization:**

Another technique to decrease the state/action space quantizes possible values. Consider the example of learning to balance a stick on a finger. To describe a state at a certain point in time involves the position of the finger in space, its velocity, the angle of the stick and the angular velocity of the stick. This yields a four-element vector that describes one state, i.e. a snapshot of one state encoded into four values. The problem is that infinitely many possible states are present. To shrink the possible space of valid actions multiple values can be assigned to a bucket. The exact distance of the finger from its starting position (-Infinity to Infinity) is not known, but rather whether it is far away or not (Near, Far).

Just for consistency in this module, we can again compare both algorithms: **SARSA**, which you could be inspired by with the implementation in the previous module, with the **Q-algorithm**, which is your scope of work in this final project.

As you can see, the only difference between **SARSA** and **Q-learning** is the action used in the target.

**SARSA** updates the equation, which uses the action actually taken in the next state to calculate the target:



**5. Conclusion:**

While we tried optimizing our method to the best of our ability, our approach had its shortcomings. Some of the limitations to the Monte Carlo method were waiting until the end of an episode before return is known, high variance, can only learn from complete sequences, and only works for episodic (terminating) environments. In addition, Q-learning is expensive for the agent, especially in the beginning steps. Q-learning also has a higher per-sample variance than SARSA, which an cause problems when training neural networks. Furthermore, in a real world setting, Q-learning would not be ideal as it ignores possible penalties for exploring its environment.

**Reference:**

Gosavi, Abhijit. (2014). Solving Markov Decision Processes via Simulation. Rolla, MO; Springer. <https://web.mst.edu/~gosavia/gosavi_rev3.pdf>

Jones, Morgan, and Matthew M. Peet. “A Generalization of Bellman's Equation with Application to Path Planning, Obstacle Avoidance and Invariant Set Estimation.” *Automatica*, Pergamon, 20 Feb. 2021, <https://www.sciencedirect.com/science/article/pii/S0005109821000303>.

S. Sundar and Z. Shiller, "Optimal obstacle avoidance based on the Hamilton-Jacobi-Bellman equation," in IEEE Transactions on Robotics and Automation, vol. 13, no. 2, pp. 305-310, April 1997, doi: 10.1109/70.563653.

Beitelspacher, Josh. Applying Reinforcement Learning to Obstacle Avoidance. Norman, OK; <https://mcgovern-fagg.org/amy_html/courses/cs5973_fall2005/Beitelspacher_Final_Paper.pdf>

V. N. Sichkar, "Reinforcement Learning Algorithms in Global Path Planning for Mobile Robot," 2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 2019, pp. 1-5, doi: 10.1109/ICIEAM.2019.8742915.

M. Bouton, A. Nakhaei, K. Fujimura and M. J. Kochenderfer, "Scalable Decision Making with Sensor Occlusions for Autonomous Driving," 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 2076-2081, doi: 10.1109/ICRA.2018.8460914.

“Wiki.” *Ros.org*, http://wiki.ros.org/ROS/Tutorials.

“The Construct.” *YouTube*, YouTube, https://www.youtube.com/c/TheConstruct/playlists.

“Self-Driving Cars with Ros 2 & Autoware.” *YouTube*, YouTube, https://www.youtube.com/playlist?list=PLL57Sz4fhxLpCXgN0lvCF7aHAlRA5FoFr.