Simulation of a drone to implement RL course algorithms

By Mansi Pawar

Motivation of the project: I came across a post on LinkedIn about delivery smart drones which delivers food and other items to various locations around the city. These drones used MC algorithm. The tasks included the map of the city, to exchange information across the drone stations and the control systems included the flying route plan. In this project, we'll train a Robot Operating System based drone to fly to a certain destination in space while staying as low as possible perhaps to escape detection and avoiding obstacles.

Over the semester, this course has helped me implement the state-of-art application of robotics i.e., artificial intelligence. The goal of this project was to broaden your robotics understanding of this cutting-edge and fascinating field of machine learning. The coding and simulations are done in software ROS (Robot Operating Systems). I was more interested in applying these algorithms to control the robot.

Concepts/Algorithms used: Multi armed bandit problem, State-Value and Action-Value functions, Markov Decision Processes, Bellman Equation, Dynamic programming, Monte Carlo methods and Temporal-difference learning. How it undergoes these reinforcement learning algorithms is that the drone(robot) is first learning. It is practicing by learning what next actions it will take with its space of action.

Motive of the project: The drone's (agent's) purpose is to fly all the way around the wall (in blue). Only one (in green) of the five tunnels (four in gray and one in green) is not closed. STATES are the other four (in gray). The drone takes off from the red location as shown in the figure below. The purpose here is implement the stated methodologies (algorithms) in our domain in order to teach the drone (robot) how to interact with the environment and make decisions without our assistance. I have used the URDF of **Drone Parrot** here.

Before the drone begins to fly, the intelligence of the agent/drone learns the best policy (strategy) to achieve the GOAL - fly all through wall (depending on the specified map).

Softwares used: ROS, OpenAI Gym and Gazebo.

Github Link to the project: https://github.com/mansipawar29/RL-Project

The ROS package that I have uploaded in my github is as follows

1. **launch** directory: main.launch file that we used to launch training.
2. **config** directory: configuration file qlearn_params.yaml with parameters for training. This is helping for tuning of parameters and keeping everything sequential.

3. training_results: results of training.
4. **utils** directory: Python file plot_results.py
5. **src** directory: code for the training of the drone.

Problem statement:

STEP 1: To use the Q-learning method of solving a three-obstacle maze environment. This implies the need to implement the **Q-learning algorithm** so the agent can learn the maze environment (Q-table). Once the learning process is complete, the agent will be able to navigate the maze without colliding. The agent will be using **SARSA algorithm**. In the maze environment, there are three obstacles (holes).

The output will look like this:

- Step :: 0 action :: right
- Step :: 1 action :: right
- Step :: 2 action :: up
- Step :: 3 action :: up
- Step :: 4 action :: up
- Step :: 5 action :: right

STEP 2: In the ROS simulation, deploy in the same way (with no modifications). ROS- Robot Operating Systems. The drone should be able to reach the goal. The agent's environment is depicted in the diagram below. First create the Q-learning algorithm before deploying it into the ROS simulation. Possible actions: Up, Down, Right and Left. The drone's movements are also depicted.

Q-learning in project environment.

12	13	Hole	**Goal**
8	6	10	14
4	Hole	Hole	7
Start	1	2	3

STEP 3: Compare SARSA and Q- algorithm using this.

STEP 4: Perform these tasks in ROS environment by implementing all the above algorithms.

The first goal was to study and implement the reinforcement learning problem, Multi armed bandit problem, State-Value and Action-Value functions, Markov Decision Processes, Bellman Equation.

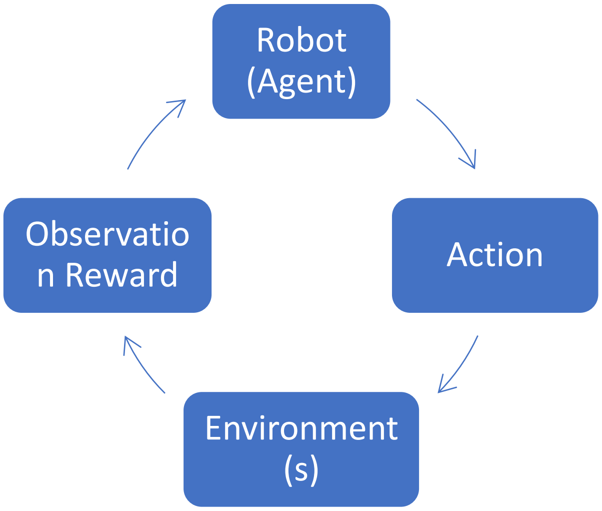


Fig 1. Agent’s internal states to test first flight of drone in ROS

The drone has the ability to fly to the left or right (action space equals 2). It specifies five phases in the environment (or cells). The agent's purpose is to figure out how to fly straight to the termination point (target), where he or she will obtain a positive reward of +1. The agent does not earn any compensation if the drone (agent) decides to fly to other locations. The agent's purpose is to fly straight to the destination with the fewest steps possible. The agent appears in eight episodes. The number of steps taken by the agent is displayed on the terminal after each episode.

Environment created in Gazebo simulator: Firstly, the environment specifies which actions are accessible to the agent. Secondly, calculate the reward based on the agent's actions and outcomes. And finally to acquire the agent's state of the world when those actions have been completed.

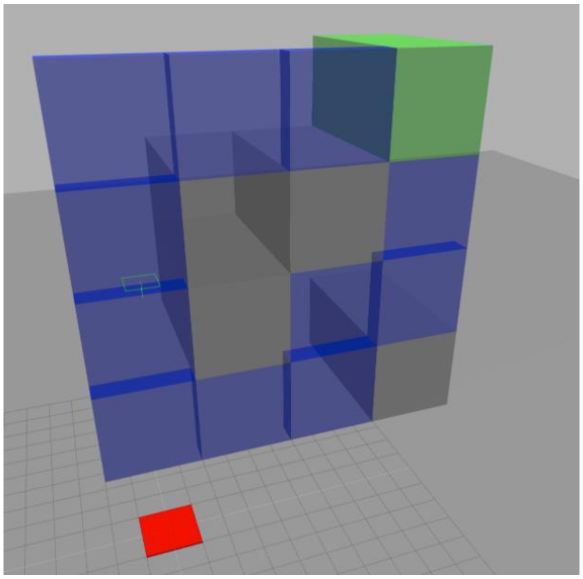
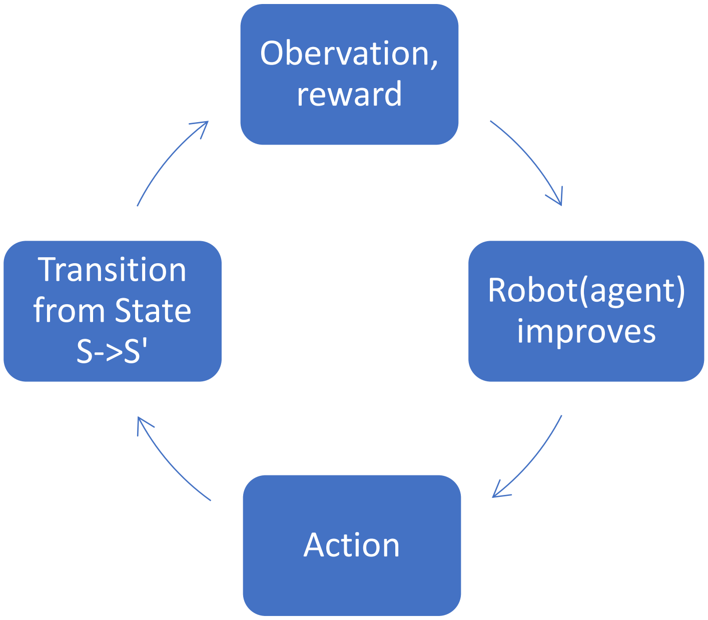


Fig 2. Gazebo screenshot of the 3D visualised maze.

It is helpful to examine the internal states of the agent before gaining a clearer grasp about what the agent is doing. First, the agent interacts with its surroundings, putting it in an interaction state, which collects data for training. Furthermore, the agent must assess its behavior i.e., the policy or plan depending on the actions done and rewards granted in the interaction state. The internal state of the agent changes to evaluation state. The learning state is the last possible state for the agent to enter. Here, the agent assesses its present behavior and considers how to enhance the decision-making process in order to increase overall performance.



Obervation, reward

Robot(agent) improves

Action

Transition from State S->S'

Fig 3. RL cycle used- Agent, Agent internal states, RL algorithm Bellman equation, MDP, Environment

I assumed 3 robotic arms first to implement the distribution of action value function and got the following output by using graphic tools in ROS:

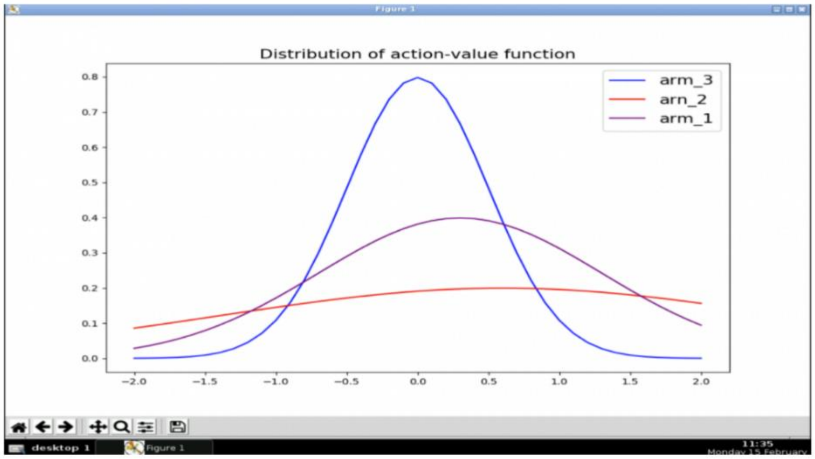


Fig 4. Action Value function

After this, I tried using bandit problem on the same. The reward distribution of each was examined:

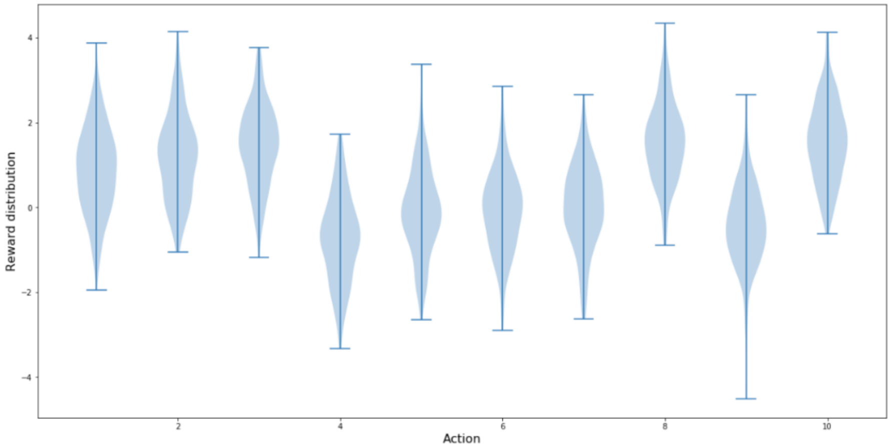


Fig 5. Reward vs distribution Bandit

This was my initial impression of the agent's brain, how it understands, and how it interacts with its surroundings. The elegance of the Bellman equation prompted me to build the essential processes of RL.

Q-learning in ROS:

Parameters: Number of episodes the agent will learn, learning factor alpha, epsilon and gamma factors and change the position of the obstacles. The agent updates the Q-table and learns it in episodes. In Ros I did not take around 100 episodes because it will give poor agent performance. So instead, I have used 2000 episodes. The outputs may differ due to the randomization used during SARSA algorithm. The Python script in the ROS simulation executes the simulation as per algorithm procedures, but the outcomes will be substantially different than predicted, thus we used a mechanism which analyzes calculated and predicted drone movements. If the findings are different, the predicted movements will be used. By analyzing both Q-learning and SARSA algorithm it was clear that the only difference between them was the target action.

Due to the randomness applied in the SARSA algorithm, the results can vary. In the ROS simulation, the Python script runs the simulation according to algorithm steps, however, the results will be very different than expected. So, I applied the mechanism that compares the computed drone motions and the expected ones.

Most challenging part was to implement these two algorithms and I used this:

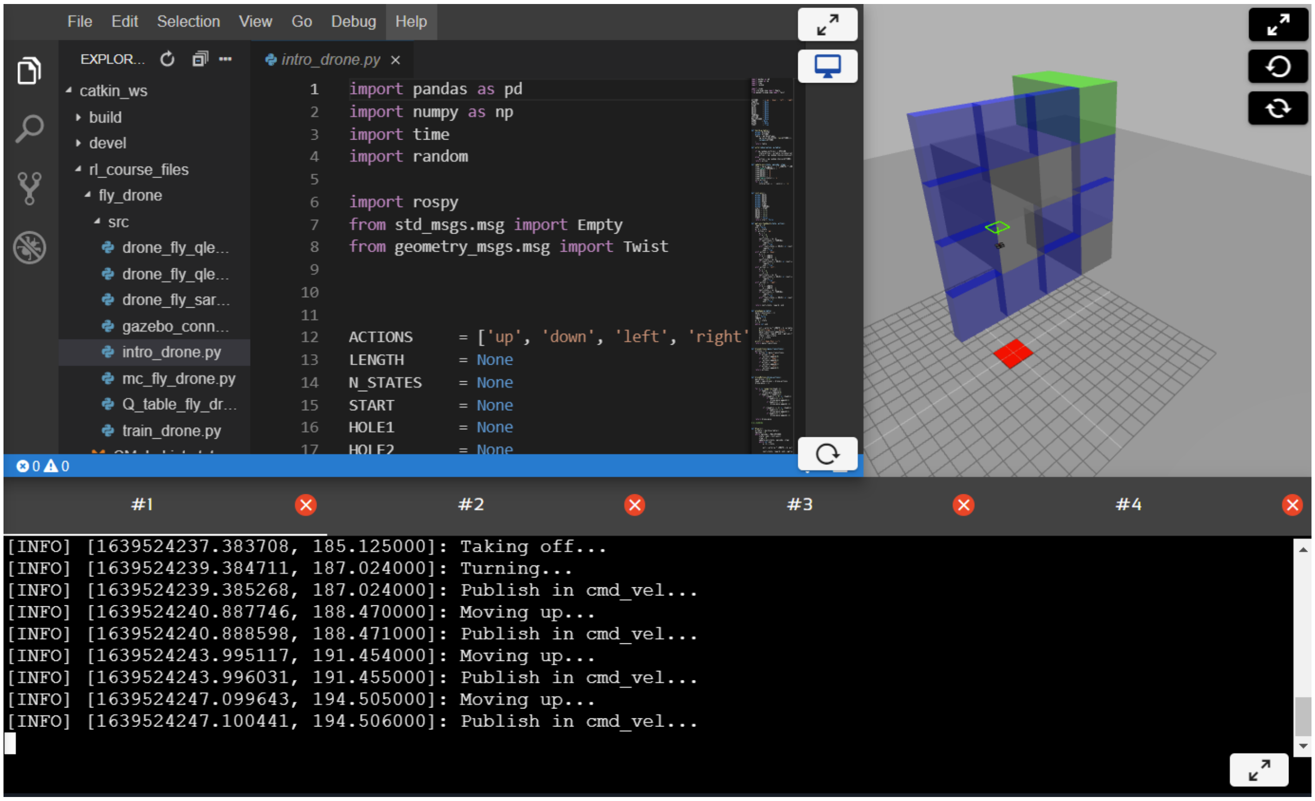
```
def learnSARSA():
    while not end:
        next_state, reward, end = get_env_feedback(state, act)
        a_, b_ = next_state
        act_ = actor(a_ * LENGTH + b_, q_table)
        q_predict = q_table.loc[a * LENGTH + b, act]
        if next_state != TERMINAL:
            q_target = reward + GAMMA * q_table.loc[a_ * LENGTH + b_, act_]
        else:
            q_target = reward
        q_table.loc[a * LENGTH + b, act] += ALPHA * (q_target - q_predict)
        state = next_state
        act = act_
        a, b = state
        update_env(state, episode, step)

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:
                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)
    episode += 1
return q_table
```

Training: The learning code and cycle is used in the training. Episode is the number of times the drone will try to complete the path and reach the goal. The drone is tested with the number of episodes. Next, is the number of loops, actions and steps taken by the drone. If all the steps are over, then it starts from the starting that means from a new episode.

Result video: I have uploaded the result video in the github link: https://github.com/mansipawar29/RL-Project. Please contact me if you are not able to open the video.

Screenshot of the output:



```
import pandas as pd
import numpy as np
import time
import random

import rospy
from std_msgs.msg import Empty
from geometry_msgs.msg import Twist



ACTIONS     = ['up', 'down', 'left', 'right'
LENGTH      = None
N_STATES    = None
START       = None
HOLE1       = None
HOLE2       = None
```

```
[INFO] [1639524237.383708, 185.125000]: Taking off...
[INFO] [1639524239.384711, 187.024000]: Turning...
[INFO] [1639524239.385268, 187.024000]: Publish in cmd_vel...
[INFO] [1639524240.887746, 188.470000]: Moving up...
[INFO] [1639524240.888598, 188.471000]: Publish in cmd_vel...
[INFO] [1639524243.995117, 191.454000]: Moving up...
[INFO] [1639524243.996031, 191.455000]: Publish in cmd_vel...
[INFO] [1639524247.099643, 194.505000]: Moving up...
[INFO] [1639524247.100441, 194.506000]: Publish in cmd_vel...
```

References:

[1] Openai_ros - ROS Wiki. (n.d.). openai_ros - ROS Wiki. http://wiki.ros.org/openai_ros.

[2] Article: Using OpenAI With ROS | The Construct. (2018, February 9). The Construct. https://www.theconstructsim.com/using-openai-ros/.

[3] Training a Drone Using ROS And OpenAI Gym. ResearchGate. https://www.researchgate.net/publication/338897000_Training_a_drone_using_ROS_and_OpenAI_Gym.

[4] Papers With Code - OpenAI Gym. (n.d.). OpenAI Gym | Papers With Code. https://paperswithcode.com/task/openai-gym.

[5] Evaluation of Reinforcement And Deep Learning Algorithms In Controlling Unmanned Aerial Vehicles by Jembre, Y. Z., Nugroho, Y. W., Raza Khan, M. T., Attique, M., Paul, R., Ahmed Shah, S. H., & Kim, B. (2021, August 6). MDPI. https://www.mdpi.com/2076-3417/11/16/7240/htm.

[6] Reinforcement Learning for Robotics | The Construct. (2021, June 3). The Construct. https://www.theconstructsim.com/robotigniteacademy_learnros/ros-courses-library/reinforcement-learning-for-robotics/.

[7] GitHub - Tobiasfshr/deep-reinforcement-learning-drone-control: A Drone Control System Based On Deep Reinforcement Learning With Tensorflow And ROS. GitHub. https://github.com/tobiasfshr/deep-reinforcement-learning-drone-control.

[8] Fisheye-Based Smart Control System for Autonomous UAV Operation. (2020, December 1). PubMed Central (PMC). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7768505/.