ECE 4424 - Machine Learning

Final Report

UAV Navigation Using Q-Learning

Team: Christopher Pham, Dingwen Chen, Moqi Zhang, Zidong Li

Table of contents

Overview

Introduction

There has been an increased interest in drone development technology due to the simplicity of design of the aerial machine with endless potential use cases. One such case is navigation and detection. Such application can allow autonomous aerial filming for scenic views, movie production, criminal chases, and geographical scanning of uncharted territories. In this project, the goal is to reproduce the Q-Learning algorithm that being used to control an autonomous Unmanned Aerial Vehicle (UAV) (from research paper [1] that we chose) equipped with cameras and sensors to navigate, detect and follow the target object in a particular area in three dimensional space. The reproduction include Q-learning implementation then simulation and integration of the said algorithm in ROS (Robotic Operating System).

Related Work

Beside the main research paper that we used for our project, we also came across several drone project which were relevant to our research topic. Among those, we found the follower technology from DJI Mavic series called called ActiveTrack to be the most interesting. This technology requires strong GPS signals and Drone Vision system in which the user manually inputs a bounding box on the object of interested. The drone will then autonomously follow the object at a desired distance. The device is built with safety features in unforeseen events and autonomously goes back to its starting position in the event of low battery power.

Problem Formulation

Problem Definition

The problem is to program a completely autonomous drone capable of recognizing with a high confident the targets in its Field of Vision (FoV). In the recognized objects, it should be able to differentiate different objects and decide the target object. Given the target object, the drone should be able to follow the target while maintaining the ideal distance and height. It learns to do this on-line.

Assumptions

Due to time constraints and difficulty of some of the problems, the following assumptions were made:

Object Detection:

- ➢ The bounding box fed for Q-Learning is fixed in a size of 1x1 grid and the Camera view is a 5x5 grid
- ➢ The object is always in the FoV of the drone's camera

Path Planning:

- ➢ There are no obstacles in the path of the drone
- ➢ The starting action of the drone is to move to [x,y,z] := [0, 0, 1]
- ➢ The drone's camera is perpendicular to the moving target leading to

Q-Learning

Objective

With the correct object, Q-Learning is executed to decide on actions in which the reward would be to maintain a proper distance to the target and centralize the target in the FoV. The cost function could potentially be the bounding box size in which the reward will be the ideal bounding box size in the image frame in the ideal frame. When the size is too small, this will skew the weight to suggest that the object is too far. Whereas, when the size is too large, the weights will be updated to suggest that the object is too close. Thus, motivating actions that keep the distance constant and keeping the target in the middle of the FoV.

Approach

Input to agent is from camera bounding box which had already been divided into equally smaller region of pixels. In our case it was a 5x5 grid. This grid represents state space of the UAV's FoV. Each state is a cell in the grid associated with a set of action available. Actions list contain the 5

following actions: ‘U’ or Up, ‘D’ or Down, ‘L’ or Left, ‘R’ or Right, ‘S’ or Stay. A successful execution of action will lead from one state to another. In simulation or real life, state space will include all situations in which the UAV can be. Since the goal of UAV is to keep the target in the center, the training helps with UAV maneuver such that other state have a reward of -1. Thus, the more actions and steps that the UAV takes, the lower the score for that episode. From experiment with multiple rewards for goal state I found that 5 is the most reasonable. Since the grid is 5 by 5, the agent executes at most 3 actions before it can reaches the goal position so a reward of 5 for the goal position will guarantee a positive score for each episode (with the least being 2) , hence making it easier to compute total reward as well as compare between each states. A visual representation of state can be found below:

(Note that 0-4 on x axis and y axis assigned for each cells to be their coordinate when tracking location)

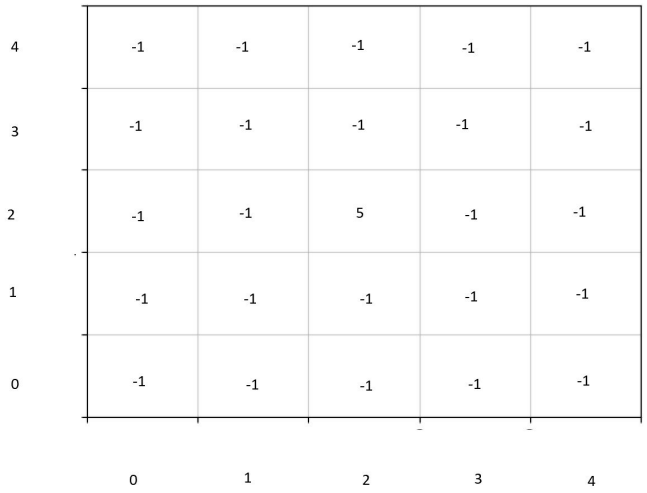

Figure 2a - State Table

Implementation

Since Q-Learning stem from MDP, we first need to define a set of state space and possible action associated with each state. As mentioned previously in approach section, our state space comprised of a 5x5 grid table with each cell represents a state (or object location) hence there is a total of 25 possible states in the state space. Depend on the location of each cell, a set of possible actions : ‘U’,

'D', 'L', 'R', 'S' is assigned. Note that not all state has all 5 possible actions because there exist a boundary around the table (the camera bounding box).

After defining state and action, we need to consider building a Q-table for the problem. Q-table acts as a policy keeper similar to MDP however unlike MDP where all policies are fixed, the Q-table is constantly updated with new training data and more suited to use in an unknown environment problem where reward for each state as well and transitional probabilities are both absent. In our case, Q-table keeps track of all the action for each individual state as well as Q-value associated with action. This Q-value is initiated with 0 in the beginning but will change/update as we get training data coming. An example of visual presentation of Q-table can be found below:

Before training:

Q-Table		Up	Down	Left	Right	Stay
States	1	0	0	0	0	0
	2	0	0	0	0	0
	3	0	0	0	0	0
	4	0	0	0	0	0
	...	0	0	0	0	0
	25	0	0	0	0	0

After training:

Q-Table		Up	Down	Left	Right	Stay
States	1	3.5	4.5	2.4	3.1	2.4
	2	4.2	1.7	2.7	4.4	3.3
	3	4.1	4.3	1.2	3.4	2.1

	4	3.1	4.5	2.1	2.4	3.2
	...	3.5	4.1	1.3	4.7	4.8
	25	4.9	4.5	3.2	1.6	3.7

Figure 2b - Q-table (before and after training)

After Q-table, we also need to consider how to explore the state space. In order to do this, we use 'epsilon' parameter which is initially set to 1 in beginning as well as a decay_rate of 0.01. The idea behind this was to decide which action to take base on these two parameters. As we generating a random number in range (0,1), if this number greater than epsilon then we do exploitation by taking the action that give us the current highest Q_value of the said state otherwise if this number less than epsilon then we do exploration and taking a random action in the action set of the said state. As we can see, 'epsilon' parameter at the beginning is 1 so we will do a lot of exploration early on, however as time goes by we need to decrease this parameter as we have more inform information of the best action to take. This is done by the following formula:

epsilon = Min_epsilon + (Max_epsilon - Min_epsilon) * 1/(exp(Decay_rate * Episode))

- epsilon: This parameter is gradually decreased by some factors including Decay_rate and the current Episode we are in.
- Max_epsilon: The original epsilon which equal to 1
- Min_epsilon: This parameter is the floor limit of epsilon which equal to 0.01 and therefore epsilon can not be lower than this value.
- Decay_rate: The rate at which epsilon is decreasing.
- Episode: Current episode we are in during training iteration.

Now that we have a foundation (state definition, action set for each states, Q-table, a way of exploring inside the state space), we can process to next step: training the agent. The training algorithm is build around calculating Q-value after executing an action and observe the outcome state (s') and reward R(s,a) then update new value to Q -Table for each state after each episode (a series of action that will lead to the goal) using the following formula:

New_Q(s,a) = Previous_Q(s,a) + learning rate * [R(s,a) + Gamma * Max Q(s',a') - Previous_Q(s,a)]

- New_Q(s,a) : Q-value to be updated of the state given an action
- Previous_Q(s,a): Existing Q_value of the state given an action (Initialize to be 0 at the beginning)
- Learning rate: Define how fast the agent can learn from new information. A learning rate of 0 makes the agent learn nothing while '1' make the agent to always consider the newest training data therefore

I chose 0.7 to be the rate since it is of a middle ground in the [0,1] range and also reflect my design choice to have a fast learning agent.

- ➢ R(s,a): Reward given a state and an action which executed from that state).
- ➢ Gamma: Discounted rate, this make the new information learn less valuable as time goes by and help eliminate infinite horizon scenario. A discounted rate of 1 will make the agent always consider long term reward while a discounted rate of 0 will values immediate reward only. I chose 0.618 in to be the rate because it is the middle ground and reflect my design choice to have an agent that moderately consider a long term reward while still be able to balance out short term reward as well.
- ➢ Max_Q(s',a'): The highest q value of next state with it best possible action.

For agent training, the process is divided into a training set and testing set. The training set contain 100 episodes (an episode has a randomly generated starting location and the agent has to find a way to reach the goal state from that point), each with 100 steps. During a training episode, the agent will choose either a random action or best action to take which lead to a next state where agent can get reward as well as knowing the Q-value of next state. We update the Q-table using new information and above formula for the current state. Next state will then be set to current state and repeat the same process until agent reach the goal state. However, if the agent does not reach the goal state within 100 steps then we still conclude the episode to start a new one. Testing set contains 20 episodes (similar to training episode, testing episode also contain random generated starting location). Unlike the training process, however, the agent will not be doing any exploration but instead only using the best action provided by Q-table and we also not update the Q-value during testing phase.

For our particular UAV problem, we try to keep the target in the middle of the camera to prevent the UAV from straying off the path it's suppose to follow until reaching the goal. With the reward system mentioned above in approach section, the optimal action will ultimately lead agent to the goal state as the goal reward has a propagation effect throughout state space (ex: state which is closer to goal has higher Q-value than the one which is further away). Since, reward is -1 for all other state except the goal state, in order to maximize its total reward, our agent will tries to minimize the number of steps it takes to reach the goal, therefore not only it will be able to maintain the object in middle by a series action but also does that in the most efficient way possible (least amount steps) through training.

Result

After training, our agent was able to execute a series of action in order to maintain its destination in the center of the camera in least of amount of steps side defined in the graph as observed in the simulation.

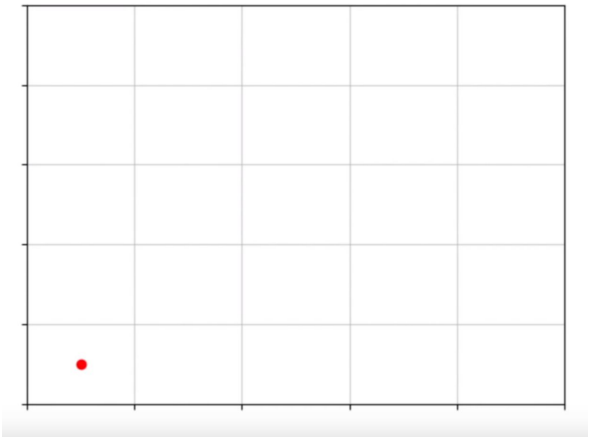

Figure 2c - Starting State (denote: red)

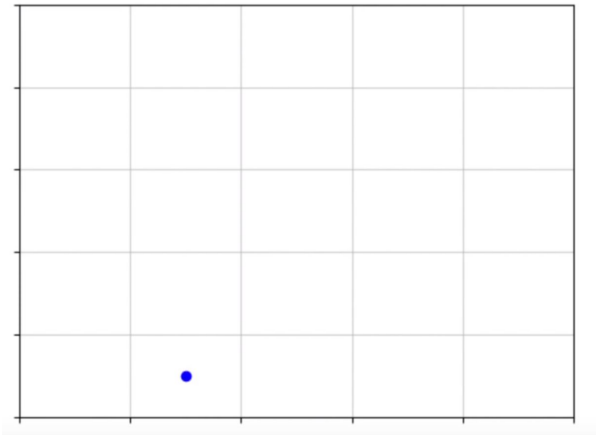

Figure 2d - Exploring State Space (denote: blue)

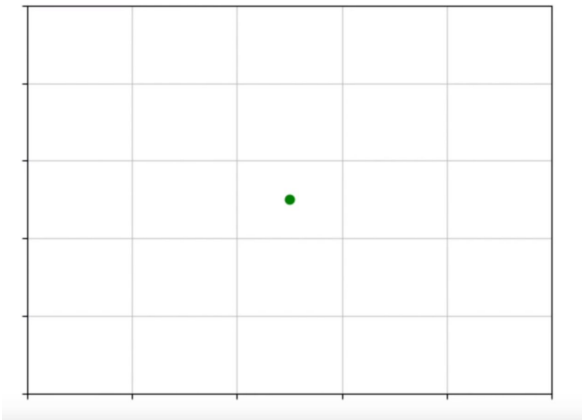

Figure 2e - Reaching Goal State (denote: green)

Testing log:

****

EPISODE 0

Starting position: [0, 4]

List of actions: ['L', 'U', 'L', 'U']

Episode rewards: 2

****

EPISODE 1

Starting position: [3, 4]

List of actions: ['D', 'L', 'L']

Episode rewards: 3

…...

****

EPISODE 19

Starting position: [0, 3]

List of actions: ['U', 'L', 'U']

Episode rewards: 3

****

Score over time: 3.4

For full log detail, link to the simulation and source code please refer to the appendix.

Simulation and Integration

ROS

Robot Operating System (ROS) is a powerful tool with robust libraries and open source code that creates an opportunity to create various robotics applications [6]. In this project, ROS was used in simulating and integrating the different parts of the project.

Simulation

Before programming the drone, simulation is used to simplify tests and avoid damaging the drone in the process. The simulation used in this project is RotorS Simulator. The simulator closely mimics the actual drone hardware to make the transfer to the actual drone trivial. Thus, resulting in an effortless transfer to hardware. The simulator has various drone models, worlds and sensors. It is robust and well documented to interface with the sensor modules and real-world physics and noise. It also documents how to create a completely new model, new sensors, and new worlds. Finally, it easily interfaces with path planning algorithms.

Approach

After following the installation process for RotorS simulator. The following code was run on the terminal:

Roslaunch rotors_gazebo mav_hovering_example.launch mav_name:=ardrone world:=basic

This launches an empty world with a Micro Aerial Vehicle (MAV) or drone with hovering capabilities. Besides setting up the world, the launch file already sets the physics in place, and begins hovering the drone 1 unit above the ground ([x, y, z] := [0, 0, 1]). The drone model chosen was ardrone, which is the provided drone that can be used during the hardware implementation. Due to memory limitations and the desire of a no-obstacles environment, most of the tests and simulations were done on the basic world.

Messages

RotorS Simulator provides messages that allow direct control over the motors, this is simply explained in the github repository as an example. For the purpose of this project, the trajectory mapping was more important. The code to move the drone through a trajectory is as follow:

Rosrun rotors_gazebo waypoint_publisher <x> <y> <z> <yaw> <delay> __ns:=ardrone

The x, y, and z are all axes locations in the simulator and the yaw rotates the drone. The above code is a ROS node that creates the trajectory points given these locations and publishes it as a trajectory message to the drone. This code is mainly used in integrating Q-Learning and the RotorS Simulator.

Integration

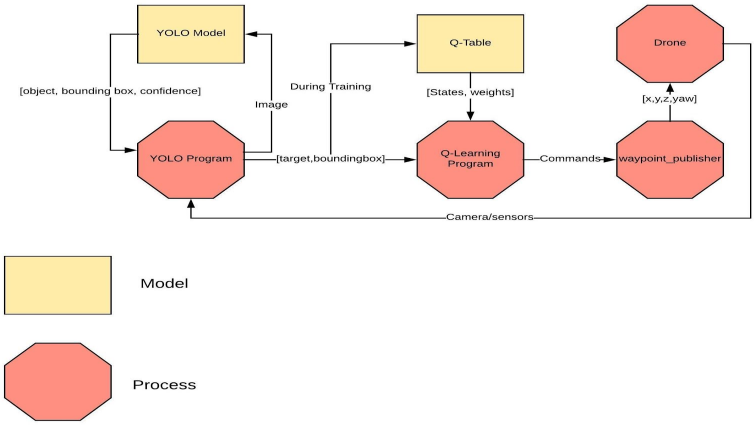

Figure 3a. Integrated task diagram of the entire system.

The figure above shows a complete integration of the project. A complete integration would include importing the YOLO model and formatting the data of the target object and its bounding box as a simplified input to the Q-Learning program.

The Q-Learning program creates its Q-Table from multiple training datasets and possibilities using real bounding box data from the YOLO program. The Q-Table would be stored in a file. During the actual execution of the program, the Q-Learning program finds the optimal policy given the YOLO data and publishes the appropriate coordinate commands to a ROS node similar to the waypoint_publisher program above. Additionally, the ardrone camera would feed images to the YOLO program. The drone thread above is actually a more complicated structure but was abstracted by RotorS simulator. The figure below shows what the drone process actually looks like .

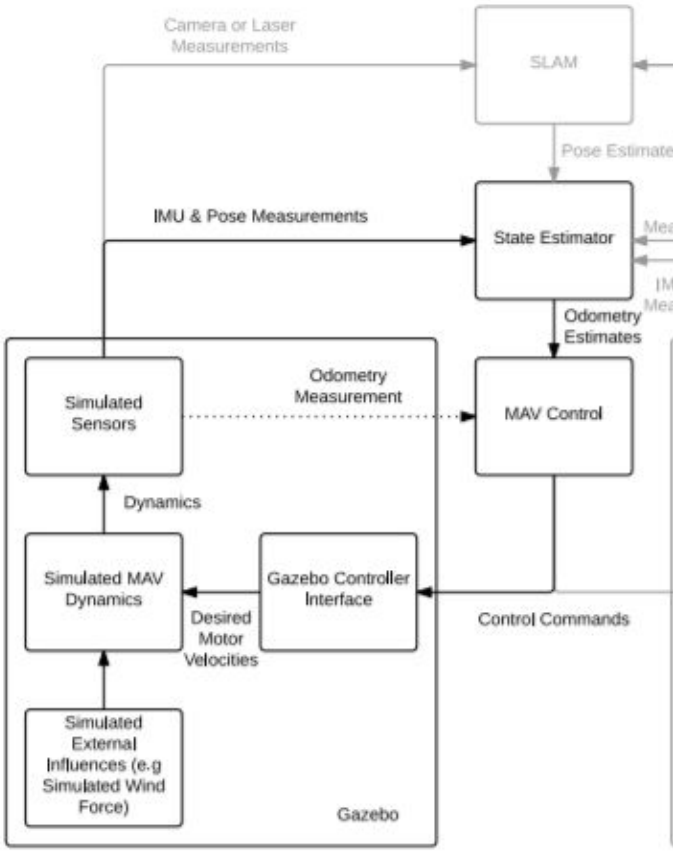

Figure 3b. Simulation Modeling a Real Drone. This is the also the Drone Process

Due to the time constraint, a complete integration was not achievable. We were only able to integrate Q-Learning and the waypoint_publisher tasks. The integration was done by modifying the Q-Learning code to output immediate waypoints instead of commands.

This was achieved by creating a [x, y, z, yaw] data structure within the execution part of the Q-Learning code. This data structure was initially [0, 0, 1, 0].

To create a functioning test, the Q-Learning program randomized the [x,y] location of the bounding box in the camera view. The Q-Learning program then returns the list of 1 step actions needed to keep the target in the middle of the camera view. This required translating it to a global waypoint action for the drone to execute. Thus a command of [‘U’, ‘L’, ‘R’, ‘D’] actually means [‘D’, ‘R’, ‘L’, ‘U’] in the actual drone actions. Moreover, the camera view does not represent the actual drone location, so limiting actions such as ‘D’ when z <= 0 or ‘U’ when z >= 2.5 were not allowed.

Because depth is not taken into consideration during this integration, it is assumed that the target is moving one step forward at a stride length of 0.75 m at each iteration. To further simplify integrating Q-Learning with waypoint publisher, Q-Learning instead executed

Rosrun rotors_gazebo waypoint_publisher <x> <y> <z> <yaw> <delay> __ns:=ardrone

To immediately communicate with the drone.

Results

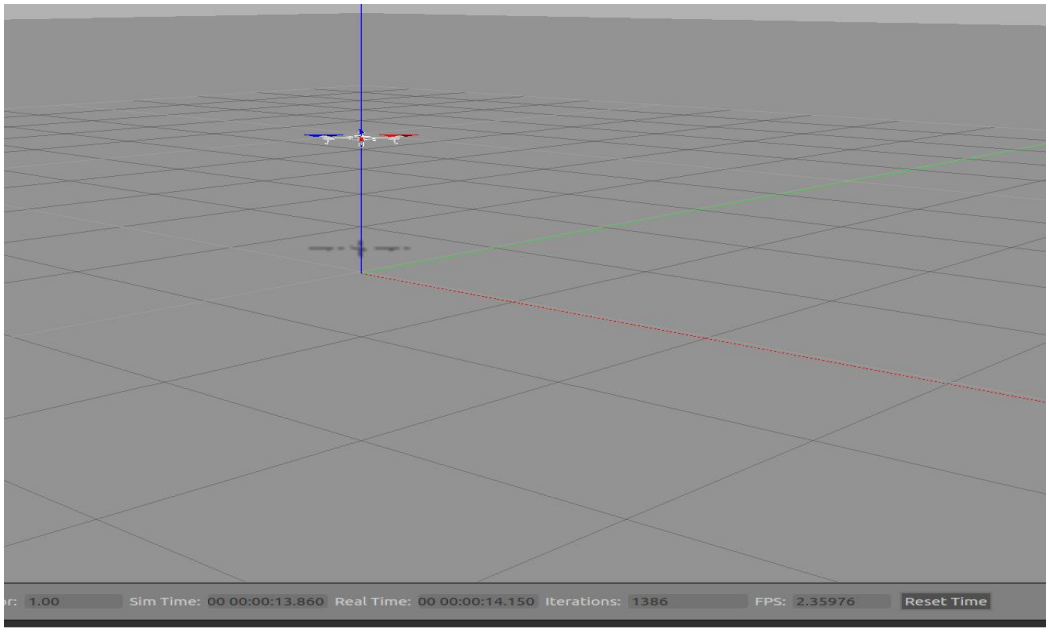

Figure 3c - UAV Simulation in ROS

In the figure above, the ardrone model is hovering 1 unit above ground in an empty world. Even with minimizing the memory consumed, the amount of FPS ranges between 2-3. Although it has a real-time factor of about 1, the simulator is very choppy and slow.

The integrated section of the entire system is capable of simulating the ardrone model to theoretically follow a target moving randomly in the y and z axis, but only forward in the x axis direction. The figure below shows a sample run simulation.

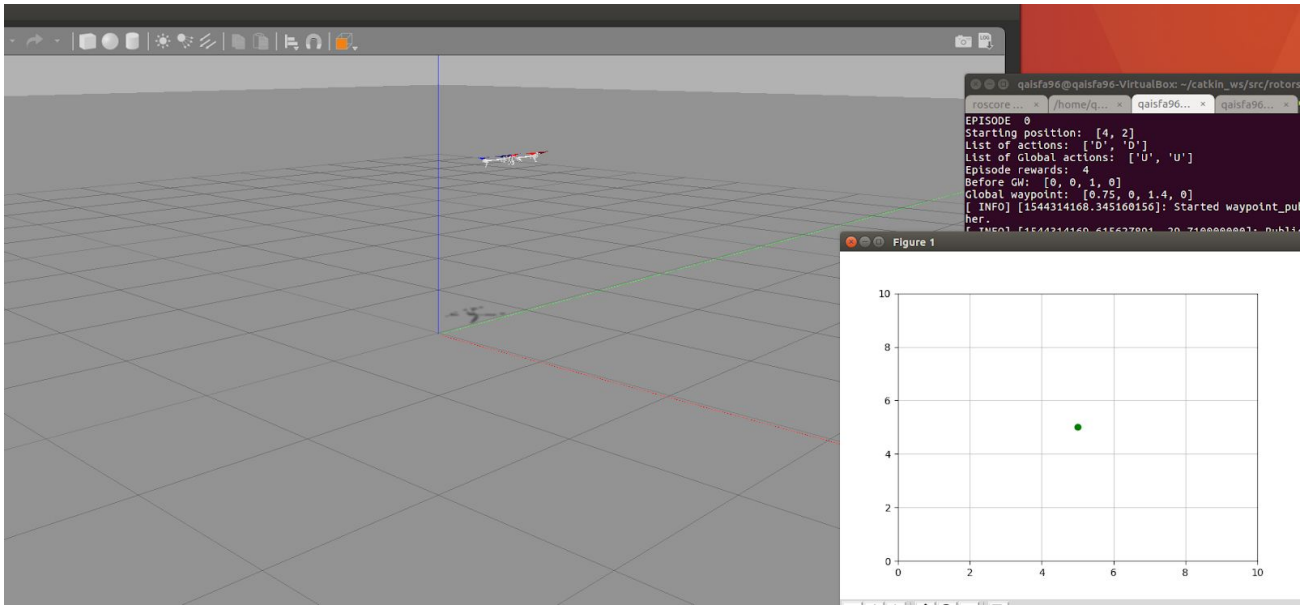

Figure 4d. Simulation in which drone is executing Q-Learning Process Commands. The grid represents the Camera View of the drone

Simulation Results

The video linked below shows a test run spanning 20 steps in which the target is randomly appearing in the camera view FoV [8].

Simulation and Integration result:

https://drive.google.com/drive/u/1/folders/0ABCF3FOGCg-AUk9PVA

Afterthought

Our design method and process aim toward simplicity by dividing the problem in to sublevel modules therefore increasing the efficiency (less prone to errors, clean design) of the design and reducing the time spend on design problem. Also knowing new modeling technique like continuous assignment contribute a great deal on productivity of design process.

With the two design elements mentioned above, we will also be able to perform more general and larger scale implementation.

Consideration

During the design process, we take several things into consideration:

- ➢ Simplicity was one of our main goal when modeling the agent.
- ➢ Several errors happened during the set up of Q-table data structure at the early stage as our initial thought was that Q-Table only keeping the best action and discarded the other. This learning experience taught us to be more careful and also get a better understanding of the correct concept when implementing a more complicated model.

Conclusion

Through this project, we have became more familiar with robot path planning and machine learning in general as well as successfully applying knowledge that we have learned from class. We also got to know more about variety of path finding techniques that widely used for industrial robot application which will help us significantly in the future.

Appendix

Research paper

[1] Research paper: https://www.researchgate.net/publication/322537496

Q-Learning

[2]Simulation video: https://www.youtube.com/watch?v=B4oPgqC9yeo&feature=youtu.be

[3] Source code: https://github.com/glacegon/Machine-Learning-Q-Learning

[4] (2019, May 04). An introduction to Q-Learning: Reinforcement learning. Available: https://medium.freecodecamp.org/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc

[5] Oppermann, A. (2019, May 04). Self Learning AI-Agents Part II: Deep Q-Learning – Towards Data Science. Available: https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

Simulation and Integration

[6] ROS: http://www.ros.org/about-ros/

[7] Robot Operating System (ROS); The Complete Reference (Volume 1), Edition: 1st ed., Publisher: Springer International Publishing, Editors: Anis Koubaa, 2016 [Online]. Available: https://www.researchgate.net/publication/309291237_RotorS_-_A_Modular_Gazebo_MAV_Simulator_Framework

[8] Simulation Results: https://drive.google.com/drive/u/1/folders/0ABCF3FOGCg-AUk9PVA