Deep reinforcement learning(SLAM)

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Rajendra Singh
(111601017)

under the guidance of

Dr. Chandra Shekar

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

<u>CERTIFICATE</u>

*This is to certify that the work contained in this thesis entitled "**Deep reinforcement learning(SLAM)**" is a bonafide work of **Rajendra Singh** (**Roll No. 111601017**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Chandra Shekar

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to express my special thanks of gratitude to my mentor **Dr. Chandra Shekar** as well as our project coordinator **Albert sunny** who gave me the golden opportunity to do this wonderful project on the topic **Deep reinforcement learning(SLAM)**, which also helped me in doing a lot of Research and I came to know about so many new things I am really thankful to them. Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Contents

List of Figures

Chapter 1

Introduction

During my summer internship at UST Global, I studied various slam algorithm. Now my attempt is to write reinforcement learning based slam algorithm for robot control, path planning, mapping and localization.

1.1 SLAM

Simultaneous localization and mapping is a problem where a moving object needs to build a map of an unknown environment, while simultaneously calculating its position within this map. There are several areas which could benefit from having autonomous vehicles with SLAM algorithms implemented. Examples would be the mining industry, underwater exploration, and planetary exploration.

The SLAM problem, in general, can be formulated using a probability density function denoted

p(xt, m—z1: t, u1:t)

where,

xt - position of the vehicle at time t

m - map

z1:t - vector of all measurements(Observations)

u1:t - vector of the control signals of the vehicle(control commands or odometry)

1.2 Goal for this semester

For this semester my main focus is to write the efficient reinforcement learning based algorithm for slam, test and analysis them with simulated robot using ros. I will be learning to design the custom environment for simulations. By the end this semester I start implementing this algorithm on the real hardware.

1.3 Organization of The Report

Code written for slam package is large and is in form of ros packages, hence this report contain just few main codes and their documentation and formulation.

In first chapter, we're discussing about the slam algorithm, goal for this semester and organisation of content in this report.

In second chapter, we'll discuss about the PID controller algorithm to control the drone, and its implementation using the ROS and VREP.

In third chapter, we'll discuss about the Q-learning and it implementation on mountain car problem using the openAI gym.

In four chapter, we'll discuss about the ROS based openAI implementation for controlling the drone using the Q learning.

In the end, fifth chapter contain the conclusion and future work.

Chapter 2

PID controller

My first sub-task of BTP was to train a drone in simulation to go from point A to point B without much deviations, jerks and oscillations. It can be achieve in two ways, one by pid controller and other by training a drone by reinforcement learning. Its important to understand PID controller approach first, later will discuss the other approach.

2.1 Introduction

PID stands for Proportional, Integral, Derivative, it's part of a flight controller software that reads the data from sensors and calculates how fast the motors should spin in order to retain the desired rotation speed of the aircraft.

The goal of the PID controller is to correct the "error", the difference between a measured value (gyro sensor measurement), and a desired set-point (the desired rotation speed). The "error" can be minimized by adjusting the control inputs in every loop, which is the speed of the motors.

There are 3 values in a PID controller, they are the P term, I term, and D term:

"P" looks at present error – the further it is from the set-point, the harder it pushes

"D" is a prediction of future errors – it looks at how fast you are approaching a set-point and counteracts P when it is getting close to minimize overshoot

”I” is the accumulation of past errors, it looks at forces that happen over time; for example if a quad constantly drifts away from a set-point due to wind, it will spool up motors to counteract it

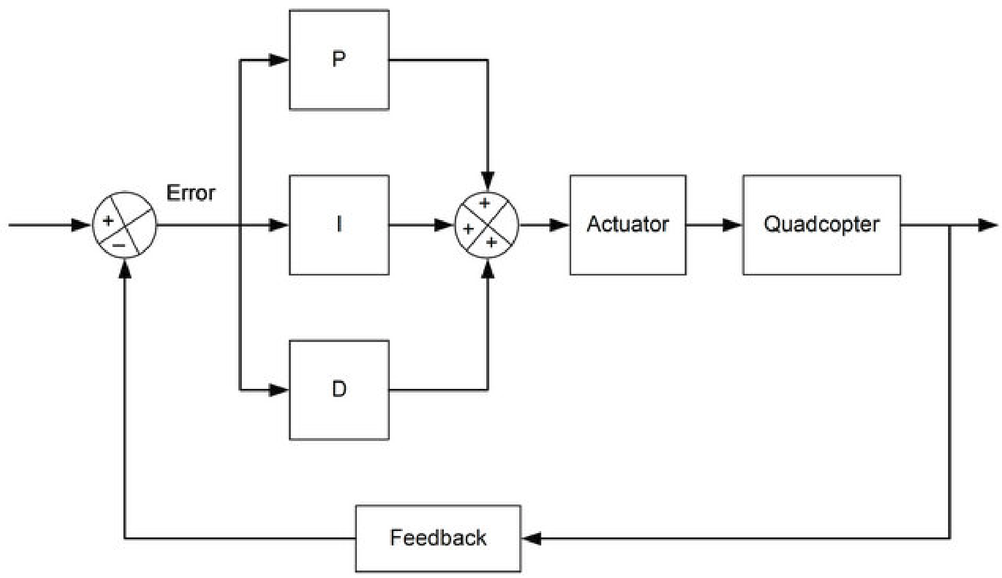

Figure 2.1 PID Controller Diagram

2.2 Implementation

Here we will implement the position hold algorithm for drone using ROS in vrep software.

2.2.1 ROS

Robot Operating System(ROS) is a meta-operating system for a robot. It provides services that one would expect from an operating system, including hardware abstraction, device drivers, libraries, visualizers, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. It is named as a meta-operating system because it is something between an operating system and middleware. It provides not only standard operating system services (like hardware abstraction) but also high-level functionalities like asynchronous and synchronous calls, a centralized database, a robot configuration system, etc. ROS can be interpreted also as a software framework for robot software development, providing the operating system.ROS is based on a Unix-like philosophy of building many small tools that are designed to work together. ROS grows out of a collaboration between industry and academia and is a novel blend of professional software development practices and the latest research results.

2.2.2 Pluto drone

Drone used for this experiment is pluto drone, this is light weight drone.

Figure 2.2 Pluto drone

2.2.3 V-REP

V-REP is simulation software which can be used with ROS to simulated various robots.

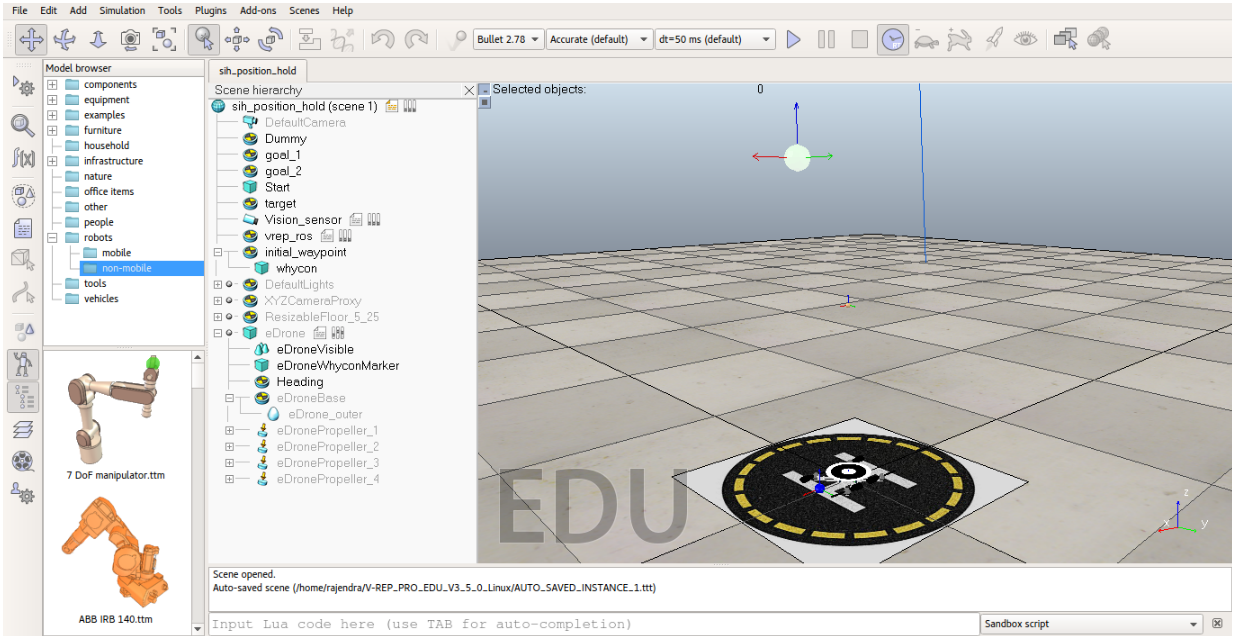

Figure 2.3 V-Rep simulation of pluto drone

2.2.4 Algorithm

Lets write the **positionHoldWhycon.py** roscode as below. It uses the position of drone we get by detecting the whycon marker on the drone and publish the drone command to hold to the given position.

```
#!/usr/bin/env python


'''


This python file runs a ROS-node of name drone_control which holds the
position of e-Drone on the given dummy.
This node publishes and subsribes the following topics:


    PUBLICATIONS        SUBSCRIPTIONS
    /drone_command        /whycon/poses
    /alt_error          /pid_tuning_altitude
```

```
    /pitch_error      /pid_tuning_pitch
    /roll_error        /pid_tuning_roll
    /yaw_error        /pid_tuning_yaw
                /drone_yaw
'''
# ========== Importing the required libraries =============#
from plutodrone.msg import *
from geometry_msgs.msg import PoseArray
from std_msgs.msg import Int16
from std_msgs.msg import Int64
from std_msgs.msg import Float64
from pid_tune.msg import PidTune
import rospy
import time


class Edrone():
  """docstring for Edrone"""
  def __init__(self):
    rospy.init_node('drone_control')  # initializing ros node with name
    drone_control
    # This corresponds to your current position of drone. This value must
    be updated each time in your whycon callback
    # [x,y,z,yaw_value]
    self.drone_position = [0.0,0.0,0.0,0.0]
    # [x_setpoint, y_setpoint, z_setpoint, yaw_value_setpoint]
```

```
self.setpoint = [-8.39, 4.98, 27.92, 0.0] # whycon marker at the
position of the dummy given in the scene. Make the whycon marker
associated with position_to_hold dummy renderable and make changes
accordingly
#Declaring a cmd of message type PlutoMsg and initializing values
self.cmd = PlutoMsg()
self.cmd.rcRoll = 1500
self.cmd.rcPitch = 1500
self.cmd.rcYaw = 1500
self.cmd.rcThrottle = 1500
self.cmd.rcAUX1 = 1500
self.cmd.rcAUX2 = 1500
self.cmd.rcAUX3 = 1500
self.cmd.rcAUX4 = 1500
# self.cmd.plutoIndex = 0
#initial setting of Kp, Kd and ki for [pitch, roll, throttle, yaw].
eg: self.Kp[2] corresponds to Kp value in throttle axis
#after tuning and computing corresponding PID parameters, change the
parameters
self.Kp = [5.13, 12, 23.1, 9]
self.Ki = [0.8 , 0, 0, 0.1496]
self.Kd = [13.365, 58.5, 342.9, 112.83]
#------Add other required variables for pid here-----------
self.prev_values = [0,0,0,0]
self.max_values = [1700,1700,1800,1800]
self.min_values = [1300,1300,1200,1200]
self.sum_of_error = [0.0, 0.0, 0.0, 0.0]
```

```
self.output = [0.0, 0.0, 0.0, 0.0]
self.iterm = [0,0,0,0]
#-------------------------------------------------------------
# This is the sample time in which run pid.
self.sample_time = 0.10 # in seconds
self.error_pub = [0.0, 0.0, 0.0, 0.0]
self.zero_line = 0
# Publishing /drone_command, /alt_error, /pitch_error, /roll_error,
/yaw_error
self.command_pub = rospy.Publisher('/drone_command', PlutoMsg,
queue_size=1)
#---------------Add other ROS Publishers here---------------
self.error_pub[0] = rospy.Publisher('/pitch_error', Float64,
queue_size=1)
self.error_pub[1] = rospy.Publisher('/roll_error', Float64,
queue_size=1)
self.error_pub[2] = rospy.Publisher('/alt_error', Float64,
queue_size=1)
self.error_pub[3] = rospy.Publisher('/yaw_error', Float64,
queue_size=1)
self.zero_line = rospy.Publisher('/zero_line', Float64, queue_size=1)
#-------------------------------------------------------------
# Subscribing to /whycon/poses, /drone_yaw, /pid_tuning_altitude,
/pid_tuning_pitch, pid_tuning_roll
rospy.Subscriber('whycon/poses', PoseArray, self.whycon_callback)
rospy.Subscriber('/pid_tuning_altitude',PidTune,self.altitude_set_pid)
#----------Add other ROS Subscribers here--------------
```

```
    rospy.Subscriber('/pid_tuning_pitch',PidTune,self.pitch_set_pid)
    rospy.Subscriber('/pid_tuning_roll',PidTune,self.roll_set_pid)
    rospy.Subscriber('/pid_tuning_yaw',PidTune,self.yaw_set_pid)
    rospy.Subscriber('/drone_yaw',Float64, self.droneYaw)
    #------------------------------------------------------------
    self.arm() #ARMING THE DRONE

  # Disarming condition of the drone
  def disarm(self):
    self.cmd.rcAUX4 = 1100
    self.command_pub.publish(self.cmd)
    rospy.sleep(1)

  # Arming condition of the drone : Best practise is to disarm and then
  arm the drone.
  def arm(self):
    self.disarm()
    self.cmd.rcRoll = 1500
    self.cmd.rcYaw = 1500
    self.cmd.rcPitch = 1500
    self.cmd.rcThrottle = 1000
    self.cmd.rcAUX4 = 1500
    self.command_pub.publish(self.cmd)  # Publishing /drone_command
    rospy.sleep(1)

  # Whycon callback function
```

```
# The function gets executed each time when /whycon node publishes
/whycon/poses
def whycon_callback(self,msg):
  self.drone_position[0] = msg.poses[0].position.x
  #-----Set the remaining co-ordinates of the drone from msg--
  self.drone_position[1] = msg.poses[0].position.y
  self.drone_position[2] = msg.poses[0].position.z
  #---------------------------------------------------------

# Callback function for /pid_tuning_altitude
# This function gets executed each time when /tune_pid publishes
/pid_tuning_altitude
def altitude_set_pid(self,alt):
  self.Kp[2] = alt.Kp * 0.06 # This is just for an example. You can
  change the fraction value accordingly
  self.Ki[2] = alt.Ki * 0.008
  self.Kd[2] = alt.Kd * 0.3

#----Define callback function like altitide_set_pid to tune pitch, roll
and yaw as well-------
def pitch_set_pid(self, pit):
  self.Kp[0] = pit.Kp * 0.006
  self.Ki[0] = pit.Ki * 0.008
  self.Kd[0] = pit.Kd * 0.003

def roll_set_pid(self, rl):
  self.Kp[1] = rl.Kp * 0.06
```

```
  self.Ki[1] = rl.Ki * 0.008
  self.Kd[1] = rl.Kd * 0.0375

def yaw_set_pid(self, yw):
  self.Kp[3] = yw.Kp * 0.006
  self.Ki[3] = yw.Ki * 0.0008
  self.Kd[3] = yw.Kd * 0.03

def droneYaw(self, yw):
  self.drone_position[3] = yw.data
#-------------------------------------------------------------
def setRange(self):
  for i in range (0, 4):
    if self.output[i] > self.max_values[i] :
      self.output[i] = self.max_values[i]
    elif self.output[i] < self.min_values[i] :
      self.output[i] = self.min_values[i]

  if self.cmd.rcPitch > self.max_values[0]:
     self.cmd.rcPitch = self.max_values[0]
  elif self.cmd.rcPitch < self.min_values[0]:
     self.cmd.rcPitch = self.min_values[0]

  if self.cmd.rcRoll > self.max_values[1]:
     self.cmd.rcRoll = self.max_values[1]
  elif self.cmd.rcRoll < self.min_values[1]:
     self.cmd.rcRoll = self.min_values[1]
```

```
    if self.cmd.rcThrottle > self.max_values[2]:
        self.cmd.rcThrottle = self.max_values[2]
    elif self.cmd.rcThrottle < self.min_values[2]:
        self.cmd.rcThrottle = self.min_values[2]

    if self.cmd.rcYaw > self.max_values[3]:
        self.cmd.rcYaw = self.max_values[3]
    elif self.cmd.rcYaw < self.min_values[3]:
        self.cmd.rcYaw = self.min_values[3]

def pid(self):
#------------- PID algorithm here-------------------------
# Steps:
#   1. Compute error in each axis. eg: error[0] =
self.drone_position[0] - self.setpoint[0] ,where error[0] corresponds
to error in x...
#  2. Compute the error (for proportional), change in error (for
derivative) and sum of errors (for integral) in each axis.
#  3. Calculate the pid output required for each axis. For eg: calcuate
self.out_roll, self.out_pitch, etc.
#  4. Reduce or add this computed output value on the avg value ie
1500.EXPERIMENT AND FIND THE CORRECT SIGN
#  5. Don't run the pid continously. Run the pid only at the a sample
time. self.sampletime defined above is for this purpose.
#  6. Limit the output value and the final command value between the
maximum(1800) and minimum(1200)range before publishing.
```

```
  #  7. Update previous errors.eg: self.prev_error[1] = error[1] where
  index 1 corresponds to that of pitch
  #  8. Add error_sum

    error = [0.0, 0.0, 0.0, 0.0]
    change_in_error = [0.0, 0.0, 0.0, 0.0]
    for i in range (0, 4):
      error[i] = self.setpoint[i] - self.drone_position[i]
      self.error_pub[i].publish(error[i])
      change_in_error[i] = error[i] - self.prev_values[i]
      self.sum_of_error[i] = self.sum_of_error[i] + error[i]
      self.iterm[i] = self.iterm[i] + (error[i] * self.Ki[i])
      self.output[i] = (self.Kp[i] * error[i]) + self.iterm[i] +
      (self.Kd[i] * change_in_error[i])
      self.prev_values[i] = error[i]

    self.cmd.rcPitch = 1500 - self.output[0]
    self.cmd.rcRoll = 1500 - self.output[1]
    self.cmd.rcThrottle = 1500 - self.output[2]
    self.cmd.rcYaw = 1500 + self.output[3]

    self.setRange()
    self.command_pub.publish(self.cmd)
    self.zero_line.publish(0.0)
    rospy.sleep(self.sample_time)
  #-----------------------------------------------------------------#
if __name__ == '__main__':
```

```
e_drone = Edrone()
while not rospy.is_shutdown():
  e_drone.pid()
```

2.3 Conclusion

I was able to hold the drone to given position but it required a lot and very precise pid tuning to get there. Hence, PID control based algorithm is easy and intuitive to understand but hard to improve and tune.

Chapter 3

Reinforcement learning

There are various RL algorithms. Here firstly we'll focus on Q learning.

3.1 Q-learning

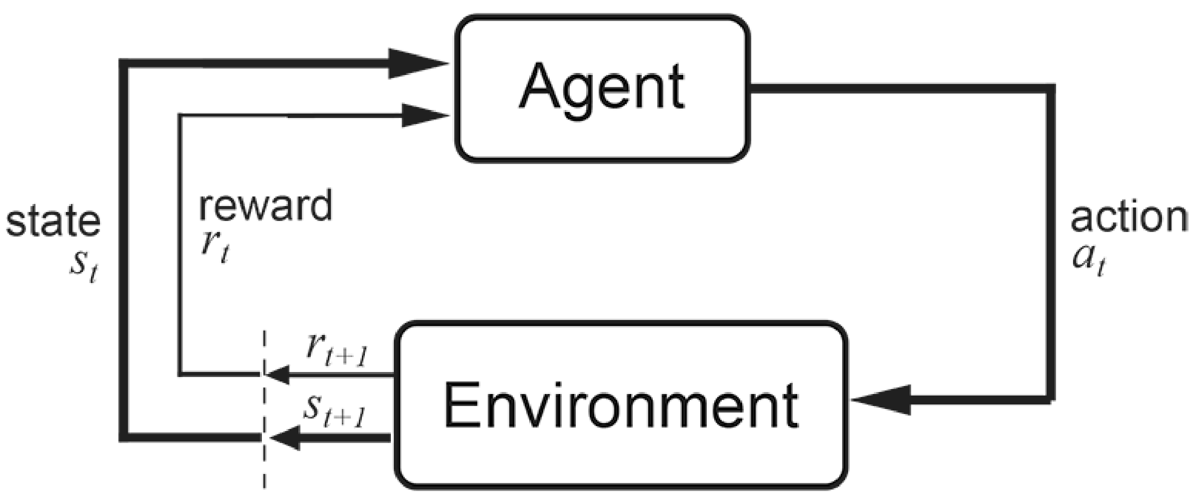

Figure 3.1 Agent and environment

Pseudo code

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

3.2 OpenAI gym

Openai gym is open source library for implementing the RL algorithm. Let train the mountain car using the q learning as

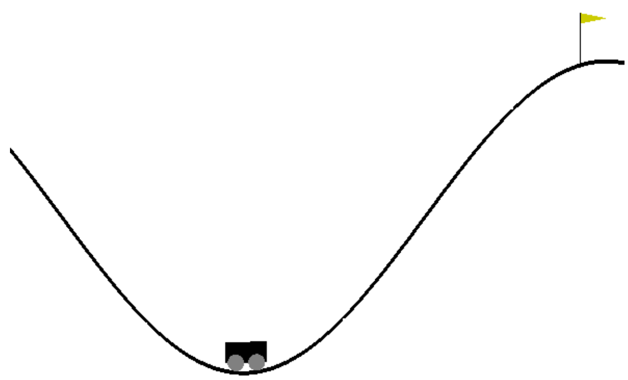

Figure 3.2 Mountain Car

Code

```
'''
    Here variable are abreviated as,
    s - state
    sd - discretised state
    s_ - next state
    s_d - discretised next state
```

```
    ns - number of state
    a - action
    na - number of action
    Q - Q table
    rlist - list of reward
    avgrlist -  - list average of reward
    R - current reward
    ep - epsilon(exploration probability)
    mxep - max epsilon
    mnep - min epsilon
    epd - epsilon decay
    gamma - discount_rate
    lr - learning rate
    ne - number of episode
'''
#============== Import necessary libraries=================#
import numpy as np
import gym
import matplotlib.pyplot as plt
#========================== create env ============================#
env = gym.make('MountainCar-v0')
env.reset()
#============== discretise state space=================#
def discret(temp):
    tempd = (temp- env.observation_space.low)*np.array([10, 100])
    tempd = np.round(tempd, 0).astype(int)
    return tempd
```

```
#============== value iteration =================#
def Viteration(env, lr, gamma, ep, mnep, ne):
    #=====ns=======#        !!! .n wont work
    ns = (env.observation_space.high - env.observation_space.low) *
    np.array([10, 100])#10 horzontal and 100 verticle
    ns = np.round(ns, 0).astype(int) + 1
    #=====qtable=======#
    Q = np.random.uniform(low = -1, high = 1, size = (ns[0], ns[1],
    env.action_space.n))
    #=====rsum, rlist, avgrlist=======#
    rlist = []
    avgrlist = []
    epd = (ep-mnep)/ne# Calculate episodic epd in ep
    #========== for ne episodes ============#
    for i in range(ne):
        done = False
        rsum,R = 0,0
        s = env.reset() #return (x,y)
        sd = discret(s)# Discretize
        #========== while not done ============#
        while done != True:
            if i >= (e - 20):# Render environment for last five e
                env.render()
            #====== explore vs exploit =====#
            if np.random.random() < 1 - ep:#epsilon greedy exr vs expt
                a = np.argmax(Q[sd[0], sd[1]])
            else:
```

```
            a = np.random.randint(0, env.action_space.n)
        s_, R, done, info = env.step(a)
        s_d = discret(s_)# Discretize
        #====== update q =====#
        if done and s_[0] >= 0.5:#Allow for terminal states
            Q[sd[0], sd[1], a] = R
        else:# Adjust Q value for current s
            Q[sd[0], sd[1],a] =(1-lr)*Q[sd[0],sd[1],a]  +   lr*(R +
            gamma*np.max(Q[s_d[0], s_d[1]]))
        #====== variable update =====#
        rsum += R# Update variables
        sd = s_d
    #====== epsilion decay =====#
    if ep > mnep:# Decay ep
        ep -= epd
    rlist.append(rsum)# Track rewards
    #====== avg reward per 100 episodes =====#
    if (i+1) % 100 == 0:
        avgr = np.mean(rlist)
        avgrlist.append(avgr)
        rlist = []
        print('Episode {} Average Reward: {}'.format(i+1, avgr))
  env.close()
  return avgrlist
#===================== calling ================================#
lr=0.2
gamma=0.9
```

```
ep=0.8
ne=5000
mnep = 0
avgrlist = Viteration(env, lr, gamma, ep, mnep, e)
#========================= plot and save =========================#
plt.plot(100*(np.arange(len(avgrlist)) + 1), avgrlist)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.title('Average Reward vs Episodes')
plt.savefig('rewards.jpg')
plt.show()
plt.close()
```

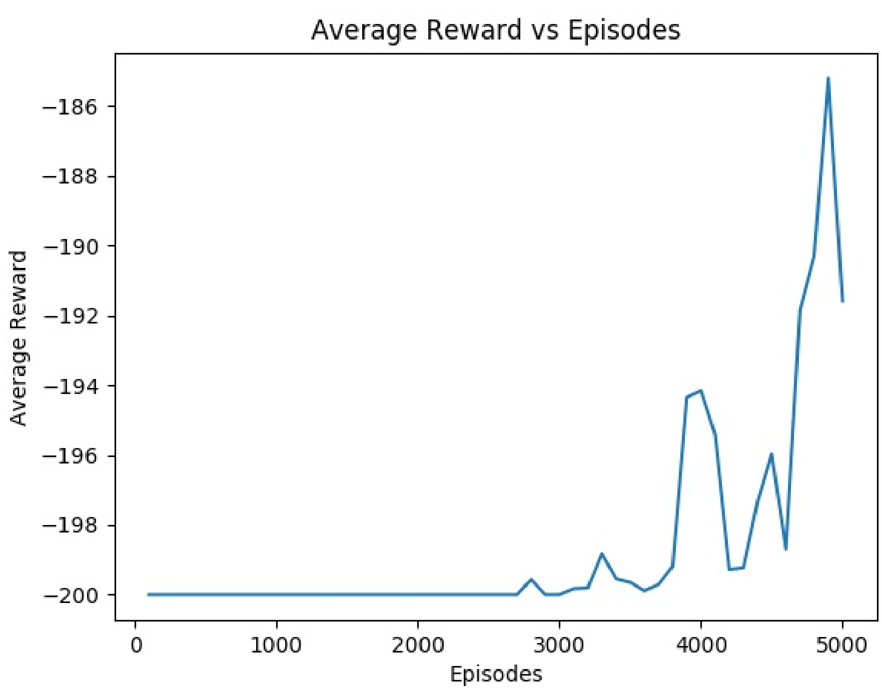

Figure 3.3 Result - average reward for 100 episodes

3.3 Conclusion

In this chapter, we studied the Q learning and implemented it on the mountain car problem using the openai gym library .

Chapter 4

ROS OpenAI Gym

Here we will try to implement the q learning based controller for drone in ros. Lets try to send drone from point A to B.

4.1 ARdrone

ARdrone most popular drone often used with the simulation using the ros.

Figure 4.1 Ardrone

4.2 ROS development studio

I used the online ros development studio by **theconstruct** for simulating RL algorithm on the drone as my local system was not powerful enough to take heavy load.

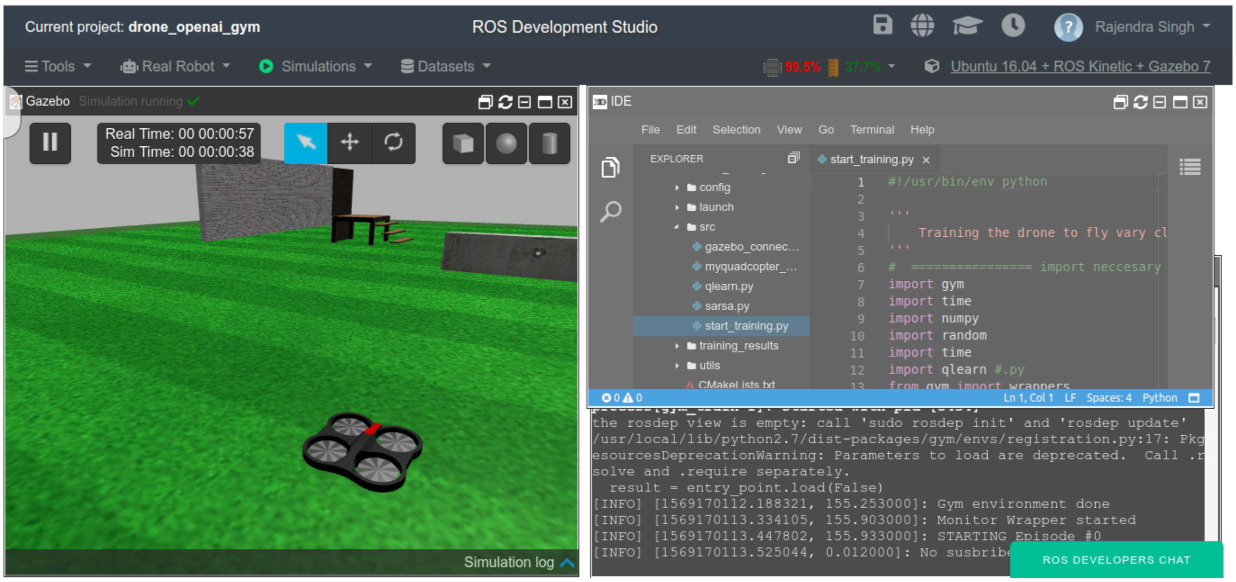

Figure 4.2 Simulation of ARdrone on ROS Development studio

4.3 Algorithm

This code is inspired by another rosject at theconstruct, let's try to understand it.

Code

```
#!/usr/bin/env python

'''
    Training the drone to fly from point A to point B.
'''
#  ================ import neccesary for gym ============= #
import gym
```

```python
import time
import numpy
import random
import time
import qlearn #.py
from gym import wrappers
# =============== required ROS libraries ================#
import rospy
import rospkg
# ========== import our training environment ============#
import myquadcopter_env #.py


if __name__ == '__main__':
    rospy.init_node('drone_gym', anonymous=True)
    # ======= Create the Gym environment ===#
    env = gym.make('QuadcopterLiveShow-v0')
    rospy.loginfo ( "Gym environment done")
    # ============== Set the logging system =================#
    rospack = rospkg.RosPack()
    pkg_path = rospack.get_path('drone_training')
    outdir = pkg_path + '/training_results'
    env = wrappers.Monitor(env, outdir, force=True)
    rospy.loginfo ( "Monitor Wrapper started")
    last_time_steps = numpy.ndarray(0)
    # load param form yaml file
    Alpha = rospy.get_param("/alpha")
    Epsilon = rospy.get_param("/epsilon")
```

```
Gamma = rospy.get_param("/gamma")
epsilon_discount = rospy.get_param("/epsilon_discount")
nepisodes = rospy.get_param("/nepisodes")
nsteps = rospy.get_param("/nsteps")
# Initialises(class) the algorithm that we are going to use for
learning
qlearn = qlearn.QLearn(actions=range(env.action_space.n), alpha=Alpha,
gamma=Gamma, epsilon=Epsilon)
initial_epsilon = qlearn.epsilon
start_time = time.time()
highest_reward = 0

# for nepisodes
for x in range(nepisodes):
    rospy.loginfo ("STARTING Episode #"+str(x))
    cumulated_reward = 0
    done = False
    if qlearn.epsilon > 0.05: #epsilon delay
        qlearn.epsilon *= epsilon_discount
    observation = env.reset()# Initialize the environment and get
    first state of the robot
    state = ''.join(map(str, observation))
    #env.render()   # Show on screen the actual situation of the
    robot
    # ==== while nsteps or not done ====#
    for i in range(nsteps):
```

```
        action = qlearn.chooseAction(state)# Pick an action based on
        the current state
        observation, reward, done, info = env.step(action)# Execute
        the action in the environment and get feedback
        cumulated_reward += reward
        if highest_reward < cumulated_reward:#update the
        highest_reward
            highest_reward = cumulated_reward
        nextState = ''.join(map(str, observation))#next state
        qlearn.learn(state, action, reward, nextState)# Make the
        algorithm learn based on the results
        if not(done):
            state = nextState
        else:
            rospy.loginfo ("DONE")
            last_time_steps = numpy.append(last_time_steps, [int(i +
            1)])
            break
    #getting hours, minutes and secounds
    m, s = divmod(int(time.time() - start_time), 60)
    h, m = divmod(m, 60)
    rospy.loginfo ( ("EP: "+str(x+1)+" - [alpha:
    "+str(round(qlearn.alpha,2))+" - gamma:
    "+str(round(qlearn.gamma,2))+" - epsilon:
    "+str(round(qlearn.epsilon,2))+"] - Reward:
    "+str(cumulated_reward)+"     Time: %d:%02d:%02d" % (h, m, s)))
```

```
rospy.loginfo (
("\n|"+str(nepisodes)+"|"+str(qlearn.alpha)+"|"+str(qlearn.gamma)+"|"+str(initial_epsi
PICTURE |"))
l = last_time_steps.tolist()
l.sort()
#print("Parameters: a="+str)
rospy.loginfo("Overall score: {:0.2f}".format(last_time_steps.mean()))
rospy.loginfo("Best 100 score: {:0.2f}".format(reduce(lambda x, y: x +
y, l[-100:]) / len(l[-100:])))
env.close() #close env
```

Note:

1.) In above code we import the myquadcopter-env.py, this is a openai gym environment made using ARdrone gazebo simulation and It is not written by me.

4.4 Conclusion

In this chapter, we proposed another algorithm to control the drone, which is based on the Q learning. As this task not just required the good understanding of Q learning but also required very good understanding of ROS and gazebo, ARdrone simulation and ros message and topics. Hence, It was not possible for me to write whole code myself and therefore I used libraries for some part, e.g. myquadcopter-env.py as mentioned above. In future, I'll learn and try to write the env also myself.

Chapter 5

Conclusion and Future Work

In this report, I presented the my worked starting form writing pid controller for a drone, then I implement the Q learning on simple game of mountain car and at end I simulated the ARdrone using ROS and openAI gym.

This was a just a start, there is a lot of ground to cover. In future I will working on improving the Q learning algorithm specific to the task and also will be learning to build the custom environment to test the algorithms.

I will working on the different variant of the above problem for example, training the drone to chase another drone, to navigated in restricted environments etc.

References

1. OpenAI gym, https://gym.openai.com/
2. ROS, http://wiki.ros.org/
3. The Construct(ROS development studio), https://rds.theconstructsim.com/
4. My Rosject, http://www.rosject.io/l/ca1685c/
5. My github repo, https://github.com/iamrajee/Slam_and_RL_BTP
6. ARdrone simulation, https://github.com/AutonomyLab/ardrone_autonomy
7. Pluto drone ros, http://wiki.ros.org/pluto_drone
8. PID controller, https://en.wikipedia.org/wiki/PID_controller