Documentation of Git Repository ReinforcementMatrix

June 18, 2020

Contents

1 Introduction

1.0.0.1 This documentation shows how to set up the *three_pi* Robot simulation using ROS and Gazebo, explains the reinforcement learning algorithm using q-learning and how to run and test the code.

2 Introduction to Gazebo and ROS

2.1 Set up for Linux Ubuntu 18.04.4 Bionic Beaver System

2.1.1 Requirements

Please make sure that you have installed the required software.

1. Ubuntu 18.04.4 LTS Bionic Beaver or higher
   (a) link
2. ROS melodic or higher
   (a) Installation guide for ROS melodic (`http://wiki.ros.org/melodic/Installation/Ubuntu`)

2.1.2 Creating a Catkin Workspace

Follow this (`http://wiki.ros.org/catkin/Tutorials/create_a_workspace`) tutorial or do the following:

1. Run the following commands in your command shell:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

   You can create your workspace anywhere on your harddrive by replacing "~"by the path to your local disc space, but notice that in the following it will be assumed that you used the displayed path for your workspace.

2. Source commands to your .bashrc
   (a) In your command shell run

```
$ gedit ~/.bashrc
```

   (b) Paste the following lines to the end of the newly opened file and save it

```
source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

2.1.3 Clone Git Repository

1. Move into the folder *~/catkin_ws/src*
2. Run the following command in your command shell

```
$ git clone https://github.com/Lizzylizard/
  ReinforcementMatrix.git
```

3. Run the following commands in your command shell or follow this (https://automaticaddison.com/how-to-launch-the-turtlebot3-simulation-with-

```
$ git clone https://github.com/ROBOTIS-GIT/
  turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.
  git
$ git clone https://github.com/ROBOTIS-GIT/
  turtlebot3_simulations.git
```

2.1.4 Build the necessary plugins

1. Message Type *vel_msg*
   (a) Paste the following line to the end of your .bashrc (see Section **??**):

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/
  catkin_ws/src/ReinforcementMatrix/my_msgs/build
```

   (b) Navigate into the folder *~/catkin_ws/src/ReinforcementMatrix/my_msgs*
   (c) Create a folder named *build*

```
$ mkdir build
```

   (d) Move into this folder and run the following commands:

```
$ cmake ../
$ make
```

2. Controls plugin
   (a) Paste the following line to the end of your .bashrc (see Section **??**):

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/
  catkin_ws/src/ReinforcementMatrix/plugins/
  vel_joint_motors/build/
```

```
16  find_package(gazebo REQUIRED)
17
18  #ADDED
19  add_message_files(
20    FILES
21    VelJoint.msg
22  )
```

Figure 1: CMakeLists.txt

(b) Move into the folder *~/catkin_ws/src/ReinforcementMatrix/plugins/vel_joint_motors*

(c) Open up the file *CMakeLists.txt* with a text editor

(d) Add the following line between line 19 and 20 (see Figure **??**):

```
1  DIRECTORY /home/YOURNAME/catkin_ws/src/
     ReinforcementMatrix/my_msgs/msg
```

i. Replace *YOURNAME* by the username of your linux system

(e) Redo the steps **??** to **??** inside the folder *~/catkin_ws/src/ReinforcementMatrix/plugins/vel_joint_motors*

3. Rebuild the catkin workspace

(a) Navigate to *~/catkin_ws*

(b) Run

```
1  $ catkin_make
```

2.1.5 Start simulation

1. In a new shell window run:

```
1  $ roscore
```

2. In a second shell window run:

```
1  $ roslaunch three_pi_description three_pi_race_city.
     launch
```

3. In yet another shell window run:

```
1  $ rosrun drive_three_pi reinf_matrix_4.py
```

(a) As an alternative you can add this last command to the launch file of the robot having the effect of not having to run the command **??** in the future anymore

i. Move to *~/catkin_ws/ReinforcementMatrix/descriptions/three_pi_description/launch*

ii. Add the following line

```
<node pkg="drive_three_pi" type="reinf_matrix_4.py" name="reinf_matrix_4" />
```

This will result in the robot starting to learn how to properly follow the line on the ground of the racetrack. It will do so in a finite number of steps which can be adjusted as we wish. We will discuss this later in Section **??**. It will also produce a lot of output. It might be helpful to redirect the output into an external text file to have a better look at what happens during the learning algorithm. This can be done by using the following command when starting the node:

```
$ rosrun drive_three_pi reinf_matrix_4.py >> Schreibtisch/Output.txt
```

2.2 Quick overview of ROS

2.2.1 What is ROS

ROS is short for Robot Operating System.

2.2.2 Commands

We are mainly going to use three ROS commands: *roscore*, *roslaunch* and *rosrun*.

roscore starts ??

roslaunch is used to start a specified simulation. It takes two arguments, the first one being the ?? and the second one being the launch-file connecting the robot to the world-file. Blabla

rosrun starts a given node. It also takes two arguments, the first one being the name of the node (name of the ROS package) and the second one being the script inside the node which we wish to execute.

Apart from that there are a lot of other ROS commands which can be very helpful. A few are specified in the following:

1. *rostopic*
2. *other command*
3. *other command*

3 Reinforcement Learning

3.1 General information

3.2 Q-Learning

3.2.1 The Bellman equation

3.3 Deep-Q-Learning

4 The code explained

Now we are going to take a closer look at the code.

4.1 The main program

Let's walk through the code step by step. We will start with the main process, continue with the image handling and finally have a look at choosing rewards and actions.

4.1.1 Overview of *reinf_main_4.py*

You can find the corresponding code here:

`../ReinforcementMatrix/node/drive_three_pi/src/Q_Matrix/Code/reinf_matrix_4.py`

4.1.1.1 Introduction *reinf_main_4.py* contains the main routine of the learning algorithm. It also takes care of the communication with ROS, which means it starts a *learning node*, publishes on a topic to actually make the robot drive and subscribes to a topic to receive the camera images of the robot.

4.1.1.2 Constructor: Connection to ROS The constructor holds important class variables and establishes the ROS node.

```
1 # publisher to publish on topic /cmd_vel
2 self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist,
    queue_size=100)
3
4 # initializing ROS-node
5 rospy.init_node('reinf_matrix_driving', anonymous=True)
6 # subscribe to topic '/camera/image_raw' using rospy.
    Subscriber
7 # class
8 self.sub = rospy.Subscriber('/camera/image_raw', Image, self.
    cam_im_raw_callback)
```

Listing 1: ROS

In this case, the node is called *reinf_matrix_driving* (line **??**).

It subscribes to a topic called */camera/image_raw* which publishes image files with a frequency of 60 Hz at the most. If an image is received, the callback method *self.cam_im_raw_callback* is invoked, which turns the ROS

image to an *OpenCV* image, segments it into foreground and background as described in section **??** and saves it as a global class variable (line **??**).

It publishes on a topic called */cmd_vel*. The message type that has to be published is called *Twist* (for further information go to [**?**]) and consists of six floating point values. We are only going to use the first two values, which will represent the speed of the robot's right and left wheel.

4.1.1.3 Important class methods

First let's have a look at the messages being published:

```
  # sets fields of Twist variable so robot drives sharp left
  def sharp_left(self):
    vel_msg = Twist()
    vel_msg.linear.x = self.speed + self.sharp
    vel_msg.linear.y = self.speed - self.sharp
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0
    # print("SHARP LEFT")
    return vel_msg
```

Listing 2: Twist message

4.1.2 State and reward handling

```
while not rospy.is_shutdown():
    # q-learning
    if(episode_counter <= self.max_episodes):
        # wait for next image
        img = self.get_image()
        # save last state, get new state and calculate reward
        self.last_state = self.curr_state
        print("Last state: " + \
          str(self.state_strings.get(self.last_state)))
        self.curr_state = self.bot.get_state(img)
        print("Current state: " + \
          str(self.state_strings.get(self.curr_state)))
        reward = self.bot.calculate_reward(self.curr_state)
        print("Reward: " + str(reward))
```

Listing 3: Main Program part 1

The main process of learning is dominated by a cycle of trying, getting feedback and, based on it, starting over again, which is represented by the while-loop in line **??** in Listing **??**. This is also the main-loop for the ROS

node. As long as the ROS node is running (as long as the user did not interrupt, e.g. pressing ctrl+c), the while loop will be executed.

Therefore an abort criterion is needed to stop the robot from learning if it passed through enough episodes. This is done in line **??**. The variable *episode_counter* stores the amount of already passed episodes, *self.max_episodes* stores the total number of episodes. As long as the first one is smaller than the second one, the robot will continue learning, otherwise it will drive based on the established q-matrix.

To start off q-learning, we need to know what the robot "sees", meaning that the current camera image is needed (line **??**). After that, the current state of the robot (how big is the distance from the robot's position to the middle of the line, represented as a positive integer between 0 and 7) is saved in a variable named *self.last_state* (line **??**). Initially *self.curr_state* is set to -1.

Next, the current state has to be computed (line **??**) based on the new image, which was received at the beginning of the while-loop. We will discuss the computing of the state in section **??** in more detail.

Dependent on that state, a reward is given (line **??**). The highest reward is given for the best state, which means, a reward of 0 is given for the line being exactly in the middle of the image, while the smallest reward (-1000) is given if the robot lost the line completely (state 7). For more information see section **??**.

4.1.3 Updating Q-Matrix

```
1 # save reward for last state and current action in q-matrix
2   self.bot.update_q_table(self.last_state, self.curr_action,
     alpha, reward, gamma, self.curr_state)
```

Listing 4: Main Program part 2

In the next step, the q-matrix is updated. This is done by calculating the Bellman-equation (see Sections **??**, **??** and source [**?**]), saving the result in the matrix at the position for the action just executed and the state of the robot before the execution of the action.

4.1.4 End of an episode

```
1 # begin a new episode if robot lost the line
2 if (self.curr_state == self.lost_line):
```

```
3     # stop robot
4     self.stopRobot()
5     # set robot back to starting position
6     self.reset_environment()
7     # episode is done => increase counter
8     episode_counter += 1
9     print("NEW EPISODE: ", episode_counter)
10    # print current q-matrix to see what's
11    # going on
12    self.bot.printMatrix(time.time())
13    print("-" * 100)
14    # skip the next steps and start a new loop
15    continue
```

Listing 5: Main Program part 3

After that we have to check whether or not the robot lost the line after the action executed. *self.lost_line* is an integer variable as well and just contains the number of the state for the lost line (7). Of course during the very first step of the first episode, there are no actions done yet, which means that the robot's current state will not be equal to having lost the line.

If the line is lost, the robot has to stop driving (line **??**) and has to be put back to a position on the line, in this case the starting position (line **??**). This also means, that the current episode is done, so the *episode_counter* is increased. To see what is going on a few print statements are done. At last the upcoming steps (Listings **??** and ...) are skipped by calling *continue* and the while-loop in Listing **??** starts over.

4.1.5 Executing actions

```
1  # get the next action
2     # if exploring: choose random action
3     if (self.epsilon_greedy(exploration_prob)):
4       print("Exploring")
5       action = self.bot.explore(img)
6       print("Action: " + self.action_strings.get(action))
7       self.execute_action(action)
8       self.curr_action = action
9     # if exploiting: choose best action
10    else:
11      print("Exploiting")
12      action = self.bot.exploit(img, self.curr_state)
13      print("Action: " + self.action_strings.get(action))
14      self.execute_action(action)
15      self.curr_action = action
```

Listing 6: Main Program part 4

4.1.5.1 Possible actions Based on the robot's current state a decision has to be made on which action to take next. There are only a few actions to choose from, in this case:

1. *sharp left*: robot turns far to the left
2. *left*: robot turns to the left
3. *slightly left*: robot turns a little to the left
4. *forward*: robot does not turn (drives forward)
5. *slightly right*: robot turns a little to the right
6. *right*: robot turns to the right
7. *sharp right*: robot turns far to the right

Taking one of those actions will result in the robot turning into the chosen direction and driving towards it, which will then produce a new state the robot is in.

For example: Let the current state of the robot be *3 = in the middle of the line*. Then the action *7 = sharp right* is taken. The robot now turns far to the right and drives for a couple of milliseconds. The expected new state would usually be *6 = far right* or even *7 = lost*.

Since nothing is learned yet in the first couple of episodes, there is no way to choose an action other than choosing it randomly (line **??**). This is called *exploring*.

After some time of learning, the q-matrix will be filled with some values. The goal is to achieve the maximum possible reward, so the next action to be taken will be the one with the highest value in the q-matrix at the position for the robot's current state (line **??**). This is called *exploiting*.

Sometimes it may be better to *explore* even though there are enough values to *exploit*, since we can never know, if the best option was already learned for a specific state or if there might be a better one. The decision of when to *explore* and when to *exploit* is made in line **??** and will be discussed in Section **??**.

In both cases, the chosen action will be executed (line **??** and **??**) and afterwards saved for later use (line **??** and **??**).

4.1.6 Review of main process

Now all steps are taken to start the while-loop all over again.

Let's quickly resume:

- The robot started off in a current state (it was somewhere on the line)
- Based on that state a reward was given. A high one if the line was close to the middle and a small one if it was not.
- The reward was saved in the q-matrix
- The next action was chosen an executed.
- The robot is now in a second state, which means that the next reward can be computed, so it can be started over.
- If the robot loses the line completely, the smallest reward will be given and it has to start from the starting position again.

4.2 The image processing class

4.2.1 Overview of *MyImage_4.py*

The image processing class can be found at *../ReinforcementMatrix/node/drive_three_pi/src/Q_Matrix/Code/MyImage_4.py*

This class receives an image as a *numpy array* from the main program after it was converted from the image that Gazebo sends via ROS into an *Open CV* image.

It segments the image into background (not the line) and foreground (the line) pixels and then computes the state of the robot depending on the line's position in the image.

We will have a closer look at it now.

4.2.2 Image segmentation

```
# black = (  0,   0,   0)
# white = (255, 255, 255)
def segmentation(self, img):
  # set color range
  light_black = (0, 0, 0)
  dark_black = (50, 50, 50)

  # black and white image (2D Array): 255 (>50)
```

```
9   # => NOT in color range,
10  # 0 to 50 => IN color range
11  mask = cv.inRange(img, light_black, dark_black)
12  return mask
```

Listing 7: MyImage.py Segmentation

Like in [?], at first, the accepted color range is defined. In here it reaches from completely black = (0, 0, 0) to a dark grey = (50, 50, 50) (lines **??** and **??**). Everything *inside* the color range will be set to 1 (white = (255, 255, 255)) and the rest will be set to 0 (black = (0, 0, 0)).

Since the line in the received image is *black* it will be set to *white*, while the background is *grey = (74, 74, 74)*, therefore does not belong into the color range and will be set to *black* (line **??**).

The resulting image is resturned (line **??**).

This method is actually called everytime a new image is received in the callback function in the main program.

4.2.3 State computing

```
1  def get_line_state(self, img):
2    # get left edge of line
3    left = self.count_pxl(img)
4    # flip image vertically (pixel on the right will be on the
       left, pixel on top stays on top)
5    reversed_img = np.flip(img, 1)
6    # get right edge of line (start counting from the right)
7    right = self.count_pxl(reversed_img)
8
9    # get width of image (should be 50)
10   width = np.size(img[0])
11
12   # get right edge of line (start counting from the left)
13   absolute_right = width - right
14   # middle is between left and right edge
15   middle = float(left + absolute_right) / 2.0
16
17   if (left >= (width * (99.0 / 100.0)) or right >=
18   (width * (99.0 / 100.0))):
19     # line is lost
20     state = 7
21   elif (middle >= (width * (0.0 / 100.0)) and middle <=
22   (width * (2.5 / 100.0))):
23     # line is far left
24     state = 0
25   # ...
26   # states 1 to 6
```

```
27    # ...
28    else:
29      # line is lost
30      state = 7
31
32    return state
```

Listing 8: MyImage_4.py State

The method *def get_line_state* takes the already segmented image as a parameter. It receives it from the main program via a helper method in the *Bot_4.py*-Class. Note that Gazebo sends an image which only consists of *one line* of 50 pixels, due to a setting that was made in the Gazebo-simulation files.

At first the left and right edge of the line have to be located on the picture. Line **??** simply counts the amount of black pixels (black pixels = background pixels) until the first white pixel (foreground pixel) is found. That gives us the distance from the left side of the image to the left edge of the line.

Line **??** does the same for the right edge of the line. But because the method *self.count_pxl(img)* will always start counting on the left side, the image has to be flipped vertically beforehand (line **??**).

Now the middle of the line can be computed. It lies exactly between the left edge of the line and the right edge (line **??**).

All the information needed to compute the state is now given. There are eight different options for the robot to be in a state:

0:	*far left*:	the middle of the line is very far to the left of the image
1:	*left*:	the middle of the line is to the left of the image
2:	*slightly left*:	the middle of the line is slightly to the left of the image
3:	*middle*:	the middle of the line is in the middle of the image
4:	*slightly right*:	the middle of the line is slightly to the right of the image
5:	*right*:	the middle of the line is to the right of the image
6:	*far right*:	the middle of the line is very far to the right of the image
7:	*lost*:	the line is not in the image anymore

The image is equally divided into sections for the states:

0%	to	2.5%	=	state 0
2.5%	to	21.5%	=	state 1
21.5%	to	40.5%	=	state 2
40.5%	to	59.5%	=	state 3
59.5%	to	78.5%	=	state 4
78.5%	to	97.5%	=	state 5
97.5%	to	100%	=	state 6

If the middle of the line is located somewhere inside one section, the corresponding state is returned as an integer (for example in line **??**).

If not, the line is lost and state 7 is returned (line **??**).

The line is also lost, if there are (almost) no foreground pixels at all (line **??**).

4.3 The robot class: Reward and Q-Matrix

4.3.1 Overview of *Bot_4.py*

The robot class stores values in the q-matrix, calculates the reward for a given state and chooses the next action for the robot to take. In short: the main steps of learning take place in this class.

The code can be found at *../ReinforcementMatrix/node/drive_three_pi/src/Q_Matrix/Code/Bot_4.py*

4.3.2 Storing values: The Bellman equation

```
# fill q-matrix -> Bellman equation
def update_q_table(self, curr_state, action, alpha, reward,
    gamma, next_state):
  # update q-matrix
  self.Q[curr_state, action] = (1 - alpha) * self.Q[
    curr_state, action] + alpha * (reward + gamma * np.max(
    self.Q[next_state, :]))
```

Listing 9: Bot_4.py Bellman

The method *def update_q_table(self, ...)* takes 6 parameters excluding the reference to the calling object:

1. *curr_state:* the state that the robot was in before taking an action
2. *action:* the action that the robot took
3. *alpha:* a floating point number between 0 and 1 that is set globally in the main program
4. *reward:* the reward the robot got after taking an action
5. *gamma:* a floating point number between 0 and 1 that is set globally in the main program
6. *next_state:* the new state that the robot is in after taking an action

The method stores the result of the Bellman equation (as described in [?] and in Section **??**) in a class variable *self.Q*, which is a two dimensional array.

The rows of this matrix are the possible actions that can be chosen, while the columns of the matrix represent the possible states.

The method's behaviour can be summarized as follows:

- The reward is discounted depending on *alpha, gamma* and the highest reward possible in the next state
- the discounted reward is stored in the q-matrix at the position [curr_state, action]

- Note that the reward *always* has to be stored for the *last* state the robot was in
- it *must not* be stored for the new state resulting from the taken action
- Otherwise the state-action pair would not match (a reward would be given for an action that was taken at a different time step) and the q-matrix's values would not be useful in the end

Compare to Sections ?? and ??.

4.3.3 Reward calculation

```
# returns the reward for a given state
def calculate_reward(self, curr_state):
  if (curr_state == 3):
    # best case: middle
    reward = 0
  elif (curr_state == 2):
    # second best case: slightly left
    reward = -1
  elif (curr_state == 1):
    # bad case: left
    reward = -2
  elif (curr_state == 0):
    # worse case: far left
    reward = -3
  elif (curr_state == 4):
    # second best case: slightly right
    reward = -1
  elif (curr_state == 5):
    # bad case: right
    reward = -2
  elif (curr_state == 6):
    # worse case: far right
    reward = -3
  else:
    # worst case: line is lost
    reward = (-1000)

  return reward
```

Listing 10: Bot_4.py Reward

The reward calculation is both simple and straight forward. It maps the current state of the robot, which is an integer number between 0 and 7 to a reward, which is also an integer number, as follows:

state 3	=	highest reward	(0)	(line **??**)
states 2 and 4	=	second reward	(-1)	(line **??** and **??**)
states 1 and 5	=	third reward	(-2)	and so on ...
states 0 and 6	=	fourth reward	(-3)	...
state 7	=	worst reward	(-1000)	...

That means, that the robot will get the highest reward if the line is exactly in the middle underneath it. The farther it gets away from the middle, the lower the reward will be. If it completely left the line, it will get a punishment of -1000. The punishment has to be a lot smaller than the rewards to really make a difference later on in the q-matrix (source [**?**]).

4.3.4 Choosing the next action

```
# explore by choosing a random action
  def explore(self, img):
    # choose one random action
    action_arr = np.random.choice(self.actions, 1)
    action = action_arr[0]
    return action

  # choose best action by getting max value out of the q-
   matrix
  def exploit(self, img, state):
    action = np.argmax(self.Q[state, :])
    return action
```

Listing 11: Bot_4.py Action

As discussed in sections **??** and **??**, the next action can either be chosen randomly (*exploring*) or by selecting the best possible option at the current state of learning (*exploiting*).

The action itself is represented by an integer number between 0 and 7 as follows:

0	=	*sharp left*
1	=	*left*
2	=	*slightly left*
3	=	*forward*
4	=	*slightly right*
5	=	*right*
6	=	*sharp right*
7	=	*stop*

Those are equivalent to the actions described in section **??**. They are stored in a one dimensional class array called *self.actions*.

If the robot is exploring, method *def explore(self, img)* will be called (line **??**). It simply chooses exactly one element out of the *self.actions-array* randomly and returns it.

If the robot is exploiting, the best possible actions will be chosen by the method *def exploit(self, img, state* (line **??**).

The best possible action is the action, which has the highest value in the q-matrix for the given state. Since the states are the *lines* of the q-matrix, the method returns the highest value in *self.Q* at index *state* (line **??**).

The decision of exploring or exploiting is made in the main routine of the program at line **??** in section **??** and is explained more detailed in section **??**.

5 Running the code

6 Results

7 Summary