Names : Nour Eldin Morsy, Nada Samir, Ahmed El Sayed, Ahmed Abdelhady, Adham Sakr, Amr Ashraf, Adam Shawky, Eslam Arafa, Ahmed Nasser

AAST-CAI DuckieTown Robot

Documentation

1. introduction

We propose the use of Duckietown, an open-source platform for autonomy education and research, as a hardware implementation for our lane-following task in reinforcement learning. Duckietown includes autonomous vehicles called "Duckiebots" that are equipped with powerful onboard computers, such as Raspberry Pi, and a variety of sensors, including cameras and odometry sensors. The Duckiebots are capable of performing complex single-robot and multi-robot behaviors, making them an ideal platform for autonomy education and research.

The platform is designed to provide a sliding scale of difficulty in perception, inference, and control tasks, making it suitable for a wide range of applications, from undergraduate-level education to research-level problems. The platform is also designed to be easily reproducible and inexpensive, with a cost of approximately $150 per vehicle and $2/m2 for the environment.

The Duckietown platform also provides a complex software architecture that includes components such as sensor calibration, configuration, low-level perception, object recognition, nonlinear relative estimation, global localization, high-level planning, and decentralized coordination. This architecture is comparable in complexity with full-scale implementations, yet still understandable by beginners.

we utilize the Duckietown platform to implement a Q-learning algorithm for lane following in the Duckietown environment. Q-learning is a model-free, off-policy algorithm that aims to find the optimal action-value function for a given problem. It uses a Q-table to store the action-value function, which is updated at each time step based on the observed rewards and the estimated value of the next state. Q-learning has been widely used in a variety of applications, including mobile robot navigation, and its effectivnes has been shown in solving problems with stochastic dynamics and partial observability.

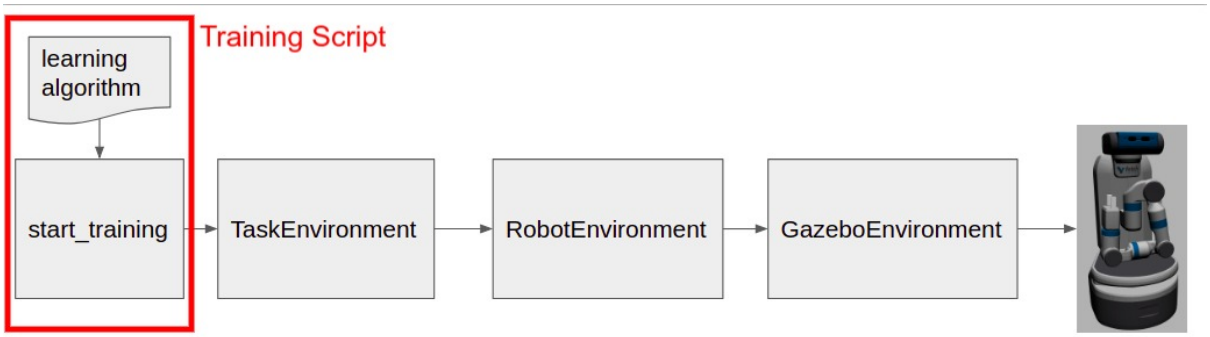
The implementation of Q-learning algorithm for lane following in Duckietown is done using the ROS framework. ROS (Robot Operating System) is a set of libraries and tools that

provide a common interface for creating and executing robot software. This allows for easy integration of different sensors and actuators, and facilitates the implementation of the Q-learning algorithm.

Overall, the use of the Duckietown platform for our lane following task in reinforcement learning presents a promising approach for autonomous mobile robots in complex environments. The platform allows for accurate perception of the environment and efficient decision-making and adaptation to changing conditions. This work serves as a foundation for further research in the field of autonomous mobile robots and reinforcement learning.

2. Model structure

Our implementation demonstrates the potential of the Duckietown platform for autonomous mobile robots in complex environments. The platform allows for accurate perception of the environment and efficient decision-making and adaptation to changing conditions. The use of



1. Gazebo environment

The Gazebo environment for Dukietown is a virtual 3D simulation platform designed to provide a realistic and interactive experience for students learning robotics. It allows students to program and control robots in a simulated environment, providing them with the opportunity to explore the fundamentals of robotics without the need for physical hardware.

The Gazebo environment features a variety of simulated objects, including buildings, trees, roads, and other obstacles. Additionally, it provides an intuitive graphical user interface that allows users to easily program and control their robots. With its realistic physics engine and detailed graphics, the Gazebo environment provides an immersive learning experience that helps students understand the complexities of robotics.

To build an environment we need two components:a **word file** to define simulation and it's models(i.e streets , traffics,etc) and an **URDF file** to define the robot components (i.e motors , camera,etc)

the Q-learning algorithm in this implementation presents a promising approach for solving lane following tasks in autonomous mobile robots. This work serves as a foundation for further research in the field of autonomous mobile robots and reinforcement learning.

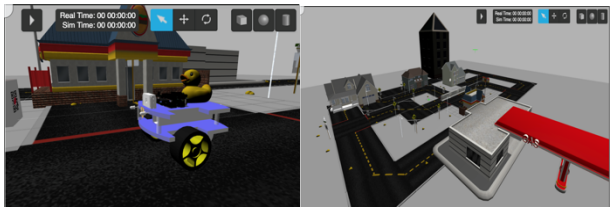


Fig. 1: The Duckietown

Fig. 2: the Duckiebot

2. Robot environment

The robot environment will then contain all the functions associated to the specific robot that you want to train in our case duckiebot. This means, it will contain all the ROS functionalities that your robot will need in order to be controlled.

we will build a class contain some points like how our robot works, how to access our robot's sensor(s), which topic(s) does our robot respond to (i.e camera , motors). Learning algorithm is necessary for the robot environment to learn from it.

3. Task Environment

The task environment class provides all the context for the task we want the robot to learn.

Task Environment specifies the following required for the training:

- ● **_init_env_variables:** function used to initialise any var that need to be set back to initial value on every episode.
- ● **_set_init_pose:** function used to initialise the robot position on every episode.
- ● **_set_action:** function used to apply the selected action to the robot. The robot can take three actions (Turn left, Turn right, Forward).
- ● **_get_obs:** function used to get the observations resulting from the action that in our case the observations of images from the camera.
- ● **_compute_reward:** function used to compute the reward. Our reward system will be based on how far or near the robot's front body with respect to the lane (e.g (centered) = 100 and (off-course) = -10).
- ● **_is_done:** function used to detect if the training for the current episode is finished.

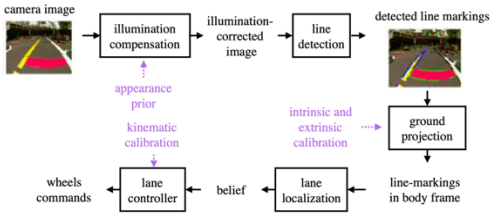
4. Lane Detection

The lane detection algorithm used in Duckietown is based on computer vision

techniques. It uses a combination of color segmentation, edge detection, and Hough transform to detect the lanes in an image.

First, the algorithm segments the image into different colors, then it detects the edges of the lanes using Canny edge detection. **Secondly,** it calculates the centroid

of the lane. **Thirdly,** it calculates the error between the car and lane. **Fourthly,** calculates kinematics equations and speed depending on error. **Finally,** the output of this algorithm is a set of points that represent the lane boundaries. These points can then be used by other algorithms to control the Duckiebot's movement.



3. Reinforcement model

The reinforcement learning model used in this project is the Q-learning algorithm. Q-learning is a model-free, off-policy algorithm that is used to estimate the optimal action-value function for a given Markov Decision Process (MDP) problem. The goal of Q-learning is to learn a policy that maximizes the expected cumulative reward over time.

The Q-learning algorithm is based on the Q-function, which is a value function that represents the expected cumulative reward of taking a specific action in a specific state and following the learned policy thereafter. The Q-function is defined as

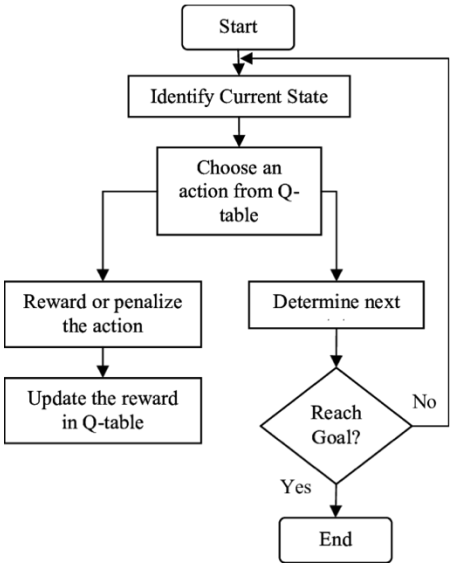
Q(s, a) = E[R(t) | S(t) = s, A(t) = a], where s is the current state, a is the current action, R(t) is

techniques. It uses a combination of color segmentation, edge detection, and Hough transform to detect the lanes in an image.

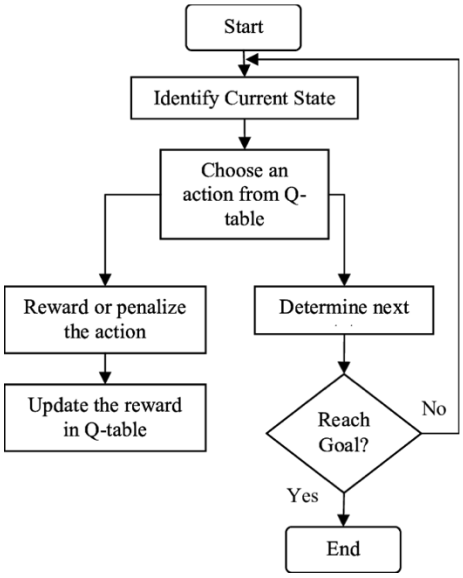
First, the algorithm segments the image into different colors, then it detects the edges of the lanes using Canny edge detection. **Secondly,** it calculates the centroid of the lane. **Thirdly,** it calculates the error between the car and lane. **Fourthly,** calculates kinematics equations and speed depending on error. **Finally,** the output of this algorithm is a set of points that represent the lane boundaries. These points can then be used by other algorithms to control the Duckiebot's movement.

the reward at time t and E[.] is the expectation operator.

The Q-learning algorithm uses the Bellman equation to update the Q-function. The Bellman equation is an iterative method that updates the Q-function using the Q-function of the next state and the reward received from taking the current action. The Bellman equation for Q-learning is Q(s, a) = Q(s, a)+α(r+γ*maxaQ(s',a)-Q(s, a)), where α is the learning rate, r is the reward received, γ is the discount factor, s' is the next state and maxa Q(s', a) is the maximum expected cumulative reward for the next state.



The Q-learning algorithm uses the epsilon-greedy exploration strategy to balance the exploitation of the learned policy and the exploration of new actions. The epsilon-greedy exploration strategy is a probabilistic method that selects the action with the highest Q-value with probability (1-ε) and selects a random action with probability ε. The value of ε is decreased during the training process to increase the exploitation of the learned policy.



In the context of this project, the Q-learning algorithm was applied to the problem of lane following for a duck-shaped robot in a simulated environment using the ROS and Gazebo frameworks. The state space consists of the observations of the duckbot's camera, and the action space consists of the linear and angular velocities of the duckbot's wheels. The Q-learning algorithm was implemented using the QLearn class in python and the parameters, such as the learning rate, discount factor and exploration rate were set through ROS parameters.

Overall, the Q-learning algorithm is a powerful and widely used technique for solving MDP problems in reinforcement learning. Its ability to learn an optimal policy without a model of the environment, and its ability to handle large and continuous state spaces make it a suitable choice for the lane following problem of the duckbot.

4. Source code implementation:

I. Detection of the lane:
   1. Take the image from the observation
   2. crop the image and leave the left up image this where the lane is there
   3. mask the image with yellow degree from the lower to the higher
   4. create a centroid in the image to calculate the error between the car and the lane

```
class LineFollower(object):

    def __init__(self):

        self.bridge_object = CvBridge()
        self.image_sub = rospy.Subscriber("/robot1/duckbot/camera2/image_raw",Image,self.camera_callback)

    def camera_callback(self,data):

        try:
            # We select bgr8 because its the OpneCV encoding by default
            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")
        except CvBridgeError as e:
            print(e)

        # We get image dimensions and crop the parts of the image we don't need
        # Bear in mind that because the first value of the image matrix is start and second value is down l
        # Select the limits so that it gets the line not too close and not too far, and the minimum portion
        # To make process faster.
        height, width, channels = cv_image.shape
        rows_to_watch = 100
        top_trunc = 1*height / 2 #get 3/4 of the height from the top section of the image
        bot_trunc = top_trunc + rows_to_watch #next set of rows to be used
        crop_img = cv_image[top_trunc:bot_trunc, 0:width]

        # Convert from RGB to HSV
        hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV).astype(np.float)

        # Define the Yellow Colour in HSV
```

```
    lower_yellow = np.array([20,100,100])
    upper_yellow = np.array([50,255,255])

    # Threshold the HSV image to get only yellow colors
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    # Calculate centroid of the blob of binary image using ImageMoments
    m = cv2.moments(mask, False)
    try:
        cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
    except ZeroDivisionError:
        cy, cx = height/2, width/2


    # Bitwise-AND mask and original image
    res = cv2.bitwise_and(crop_img,crop_img, mask= mask)

    # Draw the centroid in the resultut image
    # cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]])
    cv2.circle(res,(int(cx), int(cy)), 10,(0,0,255),-1)

    cv2.imshow("Original", cv_image)
    cv2.imshow("HSV", hsv)
    cv2.imshow("MASK", mask)
    cv2.imshow("RES", res)

    cv2.waitKey(1)


    error_x = cx - width / 2;
    angular_z = -error_x / 100;
    rospy.loginfo("ANGULAR VALUE SENT===>"+str(angular_z))

def clean_up(self):
    self.movekobuki_object.clean_class()
    cv2.destroyAllWindows()
```

II. Robot environment :

1. Define all functionality that defines the robot :
    a. Set all action , reward ,init position in the simulation , observation and where the robot is done or not
    b. Check all sensor is well (i.e topics and parameters)
    c. Init all the subscripters and publishers to the sensors

```python
    def _set_init_pose(self):
        """Sets the Robot in its init pose
        """
        raise NotImplementedError()

    def _init_env_variables(self):
        """Inits variables needed to be initialised each time we reset at the start
        of an episode.
        """
        raise NotImplementedError()

    def _compute_reward(self, observations, done):
        """Calculates the reward to give based on the observations given.
        """
        raise NotImplementedError()

    def _set_action(self, action):
        """Applies the given action to the simulation.
        """
        raise NotImplementedError()

    def _get_obs(self):
        raise NotImplementedError()

    def _is_done(self, observations):
        """Checks if episode done based on observations given.
        """
        raise NotImplementedError()
```

```python
def _check_all_sensors_ready(self):
    rospy.logdebug("START ALL SENSORS READY")
    self._check_odom_ready()
    self._check_camera_rgb_image_raw_ready()
    rospy.logdebug("ALL SENSORS READY")

def _check_odom_ready(self):
    self.odom = None
    rospy.logdebug("Waiting for "+ self.odom_topic +" to be READY...")
    while self.odom is None and not rospy.is_shutdown():
        try:
            self.odom = rospy.wait_for_message(self.odom_topic, Odometry, timeout=5.0)
            rospy.logdebug("Current "+ self.odom_topic + " READY=>")

        except:
            rospy.logerr("Current "+ self.odom_topic + " not ready yet, retrying to get odom")

    return self.odom


def _check_camera_rgb_image_raw_ready(self):
    self.camera_rgb_image_raw = None
    rospy.logdebug("Waiting for "+ self.camera_topic + " to be READY...")
    while self.camera_rgb_image_raw is None and not rospy.is_shutdown():
        try:
            self.camera_rgb_image_raw = rospy.wait_for_message(self.camera_topic, Image, timeout=5.0)
            rospy.logdebug("Current "+ self.camera_topic + " READY=>")

        except:
            rospy.logerr("Current "+ self.camera_topic + " not ready yet, retrying for getting camera_rgb_image_raw")
    return self.camera_rgb_image_raw
```

```python
def _odom_callback(self, data):
    self.odom = data

def _camera_rgb_image_raw_callback(self, data):
    self.camera_rgb_image_raw = data

def _camera2_rgb_image_raw_callback(self, data):
    self.camera2_rgb_image_raw = data

def _check_publishers_connection(self):
    """
    Checks that all the publishers are working
    :return:
    """
    rate = rospy.Rate(10)  # 10hz
    while self._cmd_vel_pub.get_num_connections() == 0 and not rospy.is_shutdown():
        rospy.logdebug("No susbribers to _cmd_vel_pub yet so we wait and try again")
        try:
            rate.sleep()
        except rospy.ROSInterruptException:
            # This is to avoid error when world is rested, time when backwards.
            pass
    rospy.logdebug("_cmd_vel_pub Publisher Connected")

    rospy.logdebug("All Publishers READY")
```

III. The task environment :

1. We defines all the task functionality (i.e. lane following)
    a. we define the actions(i.e. forward , turn left , turn right) with numbers and there velocities for the actuators

```python
def _set_action(self, action):
    """
    This set action will Set the linear and angular speed of the DuckBot
    based on the action number given.
    :param action: The action integer that set s what movement to do next.
    """

    rospy.logdebug("Start Set Action ==>"+str(action))
    self.action_taken = action
    # We convert the actions to speed movements to send to the parent class CubeSingleDiskEnv
    if action == 0: #FORWARD
        linear_speed = self.linear_forward_speed
        angular_speed = 0.0
        self.last_action = "FORWARD"
    elif action == 1: #LEFT
        linear_speed = self.linear_turn_speed
        angular_speed = self.angular_speed
        self.last_action = "TURN_LEFT"
    elif action == 2: #RIGHT
        linear_speed = self.linear_turn_speed
        angular_speed = -1*self.angular_speed
        self.last_action = "TURN_RIGHT"

    # We tell DuckBot the linear and angular speed to set to execute
    self.move_base( linear_speed,
                    angular_speed,
                    epsilon=0.05,
                    update_rate=10)
```

b. We get here the observation from the sensors and get the center lane

```
def _get_obs(self):
    """
    Here we define what sensor data defines our robots observations
    To know which Variables we have acces to, we need to read the
    DuckieTownEnv API DOCS
    :return:
    """
    rospy.logdebug("Start Get Observation ==>")

    duck_image_1 = self.get_camera_rgb_image_raw()
    duck_image_2 = self.get_camera2_rgb_image_raw()

    discretized_observations = self.discretize_observation(duck_image_1, duck_image_2)
    #rospy.logerr("Observations==>"+str(discretized_observations))

    rospy.logdebug("END Get Observation ==>")
    rospy.logerr(discretized_observations.shape)
    return discretized_observations
```

c. Check where the car get out the lane or not

```
def _is_done(self, observations):

    if self._episode_done:
        rospy.logerr("DuckBot veered OFF-COURSE ==>")
        return self._episode_done
    else:
        rospy.logerr("DuckBot is Ok ==>")

    return self._episode_done
```

d. Compute the reward and return the reward value

```
def _compute_reward(self, observations, done):
    """
    Our reward system will be based on how far or near the robot's forward with respect to the lane
    E.g 0(centered) = 100 and 1(off-course) = -10
    """

    if not done:
        if (self.action_taken != None):
            if(self.action_taken == 0):
                reward = self.follow_lane_reward #Favour going foward to turning left/right
            else:
                reward = self.left_right_reward
    else:
        reward = -1*self.follow_lane_reward


    rospy.logdebug("reward=" + str(reward))
    self.cumulated_reward += reward
    rospy.logdebug("Cumulated_reward=" + str(self.cumulated_reward))
    self.cumulated_steps += 1
    rospy.logdebug("Cumulated_steps=" + str(self.cumulated_steps))

    return reward
```

e. Get the parameter form the config file and extracted it

```
# Actions and Observations
self.linear_forward_speed = rospy.get_param('/duckbot/linear_forward_speed')
self.linear_turn_speed = rospy.get_param('/duckbot/linear_turn_speed')
self.angular_speed = rospy.get_param('/duckbot/angular_speed')
self.init_linear_forward_speed = rospy.get_param('/duckbot/init_linear_forward_speed')
self.init_linear_turn_speed = rospy.get_param('/duckbot/init_linear_turn_speed')

#Control Parameters
self.dMax = rospy.get_param('/duckbot/max_lane_offset')
self.dSafe = rospy.get_param('/duckbot/safe_lane_offset')
self.look_ahead_distance = rospy.get_param('/duckbot/look_ahead_distance')

# Rewards
self.follow_lane_reward = rospy.get_param("/duckbot/follow_lane_reward")
self.left_right_reward = rospy.get_param("/duckbot/left_right_reward")
self.veer_off_reward = rospy.get_param("/duckbot/veer_off_reward")
self.end_episode_points = rospy.get_param("/duckbot/end_episode_points")
```

IV. Qlearning algorithm:

a. We define some attributes (i.e. q value , epsilon , alpha , gamma , action)

b. We fine the q equation in learnq function

```
class QLearn:
    def __init__(self, actions, epsilon, alpha, gamma):
        self.q = {}
        self.epsilon = epsilon  # exploration constant
        self.alpha = alpha      # discount constant
        self.gamma = gamma      # discount factor
        self.actions = actions

    def getQ(self, state, action):
        return self.q.get((state, action), 0.0)

    def learnQ(self, state, action, reward, value):
        '''
        Q-learning:
            Q(s, a) += alpha * (reward(s,a) + max(Q(s') - Q(s,a))
        '''
        oldv = self.q.get((state, action), None)
        if oldv is None:
            self.q[(state, action)] = reward
        else:
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)
```

c. we define how we choose the best action by the q value and random of the epsilon exploration

```
def chooseAction(self, state, return_q=False):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)

    if random.random() < self.epsilon:
        minQ = min(q); mag = max(abs(minQ), abs(maxQ))
        # add random values to all the actions, recalculate maxQ
        q = [q[i] + random.random() * mag - .5 * mag for i in range(len(self.actions))]
        maxQ = max(q)

    count = q.count(maxQ)
    # In case there're several state-action max values
    # we select a random one among them
    if count > 1:
        best = [i for i in range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]
    if return_q: # if they want it, give it!
        return action, q
    return action

def learn(self, state1, action1, reward, state2):
    maxqnew = max([self.getQ(state2, a) for a in self.actions])
    self.learnQ(state1, action1, reward, reward + self.gamma*maxqnew)
```

V. Starting algorithm:

a. Here we starting the loop by the number of episodes and number of steps

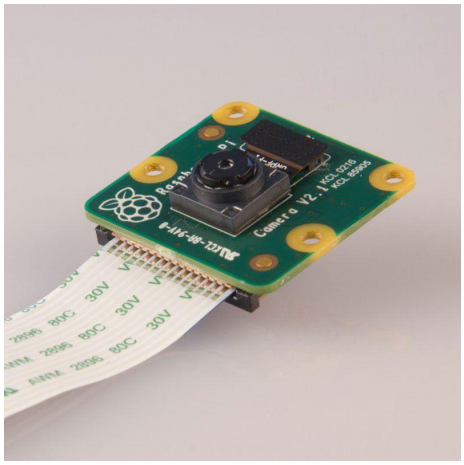
b. Get the action and the q value and compute the reward and get the best action from the max q value

```
for x in range(nepisodes):
    rospy.logdebug("################ START EPISODE=>" + str(x))

    cumulated_reward = 0
    done = False
    if qlearn.epsilon > 0.05:
        qlearn.epsilon *= epsilon_discount

    # Initialize the environment and get first state of the robot
    observation = env.reset()
    state = ''.join(map(str, observation))

    # Show on screen the actual situation of the robot
    # env.render()
    # for each episode, we test the robot for nsteps
    for i in range(nsteps):
        rospy.logwarn("############### Start Step=>" + str(i))
        # Pick an action based on the current state
        action = qlearn.chooseAction(state)
        rospy.logwarn("Next action is:%d", action)
        # Execute the action in the environment and get feedback
        observation, reward, done, info = env.step(action)

        #rospy.logwarn(str(observation) + " " + str(reward))
        cumulated_reward += reward
        if highest_reward < cumulated_reward:
            highest_reward = cumulated_reward

        nextState = ''.join(map(str, observation))

        # Make the algorithm learn based on the results
        #rospy.logwarn("# state we were=>" + str(state))
        rospy.logwarn("# action that we took=>" + str(action))
        rospy.logwarn("# reward that action gave=>" + str(reward))
        rospy.logwarn("# episode cumulated_reward=>" + str(cumulated_reward))
        #rospy.logwarn("# State in which we will start next step=>" + str(nextState))
        qlearn.learn(state, action, reward, nextState)
```

5. Hardware implementation

In the hardware implementation of a Q-learning based lane following robot, several key components are used to make the system work.

Camera: A camera is used to capture images of the lane and send them to the Raspberry Pi for processing. This camera should have a high resolution and a wide field of view in order to capture as much of the lane as possible. This will help the robot to detect the edges of the lane more accurately.



Raspberry Pi 4 model B: The Raspberry Pi 4 model B is a powerful single-board computer with 64-bit quad-core processor, 4GB of RAM. Used to process the images captured by the camera. It runs the Q-learning algorithm, which is responsible for making decisions about which actions the robot should take. The Raspberry Pi 4 also communicates with the other components of the system, such as the motors and encoders.

Encoders: are used to measure the rotational position of the motors that drive the wheels of the robot using pulses per revolution, it is a measure of the number of pulses per full revolution, with a full revolution being 360 degrees, so we connect the encoder to the raspberry pi through the motor driver and rotate the encoder manually one full rotation then read

the number of pulses generated in one revolution. To calculate the RPM we use the following equation:

This information is used to determine the robot's position and speed. The encoders send this information to the Raspberry Pi, which uses it to make decisions about how to control the robot's movement.

Motors: We used two DC 12V 80rpm encoder motors. These Motors are used to drive the wheels of the robot. They are controlled by the Raspberry Pi, which sends commands to them based on the decisions made by the Q-learning algorithm. The motors should be powerful enough to move the robot at a reasonable speed and should have a high level of precision to make fine adjustments to the robot's movement.

$$\text{RPM} = \frac{(\text{Pulse frequency in pulses /sec}) \times (60 \text{ sec/min})}{(\text{Pulses/revolution})} = \frac{\text{Revolutions}}{\text{Minute}}$$

H-bridge motor controller: is a type of circuit that is used to control the speed and direction of a DC motor. The circuit is called an H-bridge because it is shaped like the letter "H" when viewed from above.

The H-bridge circuit is made up of four transistors, two of which are used to control the direction of current flow through the motor, and the other two are used to control the speed of the motor using a pulse width module (PWM). The H-bridge circuit is controlled by the raspberry pi, which can be used to control the speed and direction of the motor in response to various inputs.

Battery: A battery is used to power the robot. It should be powerful enough to run all of the robot's components for a reasonable amount of time and should be easy to replace or recharge.

Chassis: A chassis is the physical structure that holds all of the robot's components together. It should be strong and lightweight, and should be designed to protect the robot's components from damage.

Wheels: wheels are the main contact between the robot and the environment. They should be well-designed and well-matched with the robot's motors and encoders. They should have a good grip on the surface and be able to move smoothly.

By having all of these components working together, the duckierobot is able to navigate and follow lanes. The camera captures images of the lane, the Raspberry Pi processes these images, the encoders measure the robot's position and speed, the motors drive the robot's wheels, the motor controller controls the motors, the battery powers the robot and the chassis holds everything together. The wheels are the main contact between the robot and the environment.