Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Aprendizaje Automático en robots móviles

Machine learning in mobile robots

Andrés Zeus Hernández Impini

La Laguna, 13 de septiembre de 2022

D. **Jonay Tomas Toledo Carrillo**, con N.I.F. 78698554Y profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

Dña. **Bibiana Fariña Jeronimo**, con N.I.F. 54063772H Investigadora adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutora

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Aprendizaje Automático en robots móviles*”

ha sido realizada bajo su dirección por D. **Andrés Zeus Hernández Impini**, con N.I.F. 45.863.381-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos, firman la presente en La Laguna a 13 de septiembre de 2022

Agradecimientos

Agradezco a todas las personas que me han acompañado durante todo este proceso tan importante en mi vida como profesional, que es la carrera de Ingeniería Informática. Poniendo especial atención en aquellas personas que siempre, pero siempre, me han apoyado sin pensarlo y de manera incondicional, como es mi madre, Norma, que desde pequeño ha impulsado cada uno de mis sueños y proyectos, apoyándome y motivándome cada día desde que tengo uso de razón, ayudándome en lo que podía, por más insignificante que pareciera para mi todo eso era más que valioso, a ti te dedico esta memoria.

También en este proceso, me ha ayudado mucho, una persona que para mi vale millones, y posiblemente sea la persona más bondadosa e inteligente que conozca, mi pareja, Jacqueline, que siempre me ha ayudado en mis momentos más difíciles, siempre me ha levantado cuando no encontraba la luz y siempre me ha ayudado desde que le conocí, y como es tan importante en mi vida también lo es en todos mis proyectos así que, a ti también te agradezco por acompañarme durante todo este tiempo y te dedico esta memoria.

No puedo olvidarme de las maravillosas amistades que encontré en la carrera y espero seguir manteniendo, porque no solo en ellos encontré amistad, sino también un refugio, un lugar donde sentirme seguro dentro de lo desconocido de las asignaturas y que al ayudarnos mutuamente hemos podido superar todas nuestras dificultades.

Finalmente, a las amistades de toda la vida que me han ayudado siempre en lo que podían (a su manera)y a cerrar esta etapa en mi vida, les agradezco y les dedico esta memoria, ya que me han hecho todo más ameno y siempre les tendré un hueco en mi corazón.

No olvidarme de mis tutores, que me han ayudado en el desarrollo de este TFG, acompañándome y tutorizándome en cada un a de mis dudas, también tienen mi agradecimiento.

Licencia

Resumen

Este proyecto, en líneas generales, trata sobre dos temas importantes hoy en día, la robótica y el aprendizaje automático, basadas en agentes con cierto nivel de inteligencia artificial, particularmente nos adentraremos en el Q-Learning y algunas de sus variantes, aplicaremos esta técnica sobre un robot (nuestro agente) para evitar que colisione contra las paredes de un laberinto, y aprenda dinámicamente, mediante una técnica llamada aprendizaje por refuerzo, a moverse por el mismo, es decir si aprende a moverse por uno, debería aprender a moverse por todos.

Palabras clave: Q-Learning, Robótica Móvil, Inteligencia Artificial, Aprendizaje por refuerzo.

Abstract

This project, in general terms, deals with two important topics nowadays, robotics and automatic learning, based on agents with a certain level of artificial intelligence, particularly we will go into Q-Learning and some of its variants, we will apply this technique on a robot (our agent) to prevent it from colliding against the walls of a maze, and dynamically learn, through a technique called reinforcement learning, to move through it, i.e. if it learns to move through one, it should learn to move through all of them.

Keywords: Q-Learning, Mobile Robotics, Artificial Intelligence, Reinforcement Learning.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación

"Nos fascinan los robots porque son un reflejo de nosotros mismos."

– Ken Goldberg

La informática, desde que tengo uso de razón, siempre me ha gustado, y con ello cada una de sus ramas, y es aquí cuando llegamos a la robótica, esta división de la ingeniería increíblemente complicada, pero a la vez elegante, que quizás puede resultar hasta bonita, una rama multidisciplinar que se puede aplicar a casi todo, desde la medicina con sus robots quirúrgicos, hasta la fabricación de coches con sus robots produciendo en la línea de montaje. Debido a ello, considero que es una buena forma de cerrar esta etapa de mi vida, aportar mi pequeño grano de arena a lo que es el futuro venidero, que todos sabemos que llegará, tarde o temprano, la inteligencia artificial en cada ámbito de nuestra vida, y con ello en los robots, que serán capaces de hacer cada tarea, desde las más simples y triviales, hasta las más peligrosas y complejas, probablemente algún día nos ayudarán a resolver todos esos problemas que los humanos no somos capaces de entender... pero por algún lugar hay que empezar.

1.2. Antecedentes

Hasta hace medio siglo, la automatización era casi sinónimo de mecanización. El desarrollo de maquinaria y otros dispositivos técnicos sustituyó al trabajo manual. Los inconvenientes son los grandes costes y que los equipos son muy rígidos. Antes de producir un nuevo producto, había que reconstruir toda la línea de producción. Esto hizo que la mecanización se aplicara principalmente en las industrias con producción en masa, como la del automóvil[4].

La llegada de los semiconductores y con ellos de ordenadores y algoritmos más potentes hicieron que la robótica avanzara rápidamente entre los 80’ y los 90’[5]. Coincidió curiosamente con el invierno de la inteligencia artificial[6], haciendo que los algoritmos de navegación de esa época se basaran en el procesamiento bruto de los datos, a este tipo de algoritmos se los conoce como algoritmos deliberativos. En resumen, se basan, en crear o cargar un mapa del entorno en el cual se van a desarrollar y mediante algoritmos de búsqueda de caminos como el A estrella, logran llegar a su destino o realizar las tareas.

Si volvemos a nuestra época, nos encontramos con multitudes de algoritmos de todo tipo y con un gran avance en la inteligencia artificial, con distintas ramas de la misma, algoritmos de aprendizaje por refuerzo, algoritmos genéticos, redes neuronales, etc. Si nos centramos en el aprendizaje por refuerzo, o Q-Learning[7] con una simple búsqueda, nos daremos cuenta de que este algoritmo está resuelto desde hace años, pero aún se sigue utilizando en multitud de aplicaciones, desde entrenamiento de coches autónomos[8] hasta el uso de robots aspiradores en la vida diaria[9].

El ejemplo claro es el robot aspirador, sobre todo, los más modernos utilizan un sensor para evitar colisionar con las paredes, personas, mascotas, este tipo de robots son robots reactivos, que reaccionan al entorno mediante la información que les llega de su entorno en tiempo real y en nuestro caso lo harán de una forma dinámica.

1.3. Problemática

Como hemos visto en las secciones anteriores, podemos diferenciar dos ramas marcadas en la robótica móvil, la deliberativa, y la reactiva. Nosotros nos centraremos únicamente en la reactiva, el objetivo final es simular y estudiar el comportamiento de un robot reactivo, que tiene como algoritmo uno de aprendizaje por refuerzo. Para ello usaremos distintas herramientas muy utilizadas tanto para modelar como para simular robots.

1.3.1. Robotic Operating System

Robotic Operating System, se basa en la abstracción de un OS, de tal forma que se basa en el paso por mensajes de una manera estándar entre distintos tipos de computador[10], es decir, proporciona un soporte en las comunicaciones entre ordenadores muy distintos, por ejemplo, entre un PC normal y un robot, lo cual lo hace bastante potente además de que se trata de código abierto y podemos encontrar mucha documentación en la web. Para la realización de este proyecto debemos entender, aunque sea de manera básica, el funcionamiento de ROS.

1.3.2. Modelaje del Robot, nuestro agente

Para el robot utilizaremos un robot ampliamente usado para el desarrollo en ROS, el **Turtlebot**, es un robot prototipo, muy sencillo y fácil de construir. El TurtleBot consta de una base móvil, un sensor de distancia 2D/3D (Kinect de Microsoft), un SBC (Single Board Computer) (Raspberry Pi), microcontroladores (Arduino UNO), sensores (ultrasónico y acelerómetro- giroscopio) y componentes para permitir la movilidad.[11]

1.3.3. Gazebo, el mundo 'real'

Además, tenemos que añadir el entorno donde se va a poner el robot a aprender, es decir, los obstáculos, las paredes, etc. Para ello debemos modelar las paredes con el suficiente espacio para que el robot pueda moverse con cierta libertad para aprender.

Para ello utilizamos Gazebo, el mismo está diseñado para reproducir con precisión los entornos dinámicos que puede encontrar un robot. Todos los objetos simulados tienen masa, velocidad, fricción y muchos otros atributos que les permiten comportarse de forma realista cuando son empujados, tirados, derribados o transportados. Estas acciones pueden utilizarse como partes integrantes de un experimento, como la construcción o la búsqueda de objetos.

También lo utilizamos porque es completamente de código abierto y de libre acceso (una gran ventaja sobre la mayoría de los paquetes comerciales). Por ello, Gazebo cuenta con una base activa de colaboradores que hacen evolucionar rápidamente el paquete para satisfacer cualquier tipo de problema.[12].

1.3.4. Algoritmos de Aprendizaje y OpenAI, el cerebro del agente

Finalmente, para terminar de entender el funcionamiento del proyecto, es necesario saber que herramientas de aprendizaje por refuerzo poseemos. En nuestro caso hemos decidió utilizar en un principio para el aprendizaje básico, el Q-Learning.

Esto lo hacemos mediante la librería de OpenAI para ROS, específicamente la librería de OpenAI Gym. Este se centra en el escenario de episodios de aprendizaje por refuerzo, con el objetivo de maximizar la expectativa de recompensa total cada episodio y conseguir un nivel de rendimiento aceptable lo más rápido posible. El objetivo de este kit de herramientas es integrar la API de Gym con el hardware robótico, validando los algoritmos de aprendizaje por refuerzo en entornos reales[13].

2

Robotic Operating System

En el capítulo anterior se ha introducido ROS, un sistema para el paso de mensaje entre robots, este "Sistema Operativo", tiene ciertas peculiaridades que exploraremos a lo largo de este capítulo, su arquitectura y como funciona en líneas generales.

2.1. Estructura general

La estructura general de ROS se basa en nodos, mensajes, tópicos y servicios que explicaremos a continuación:

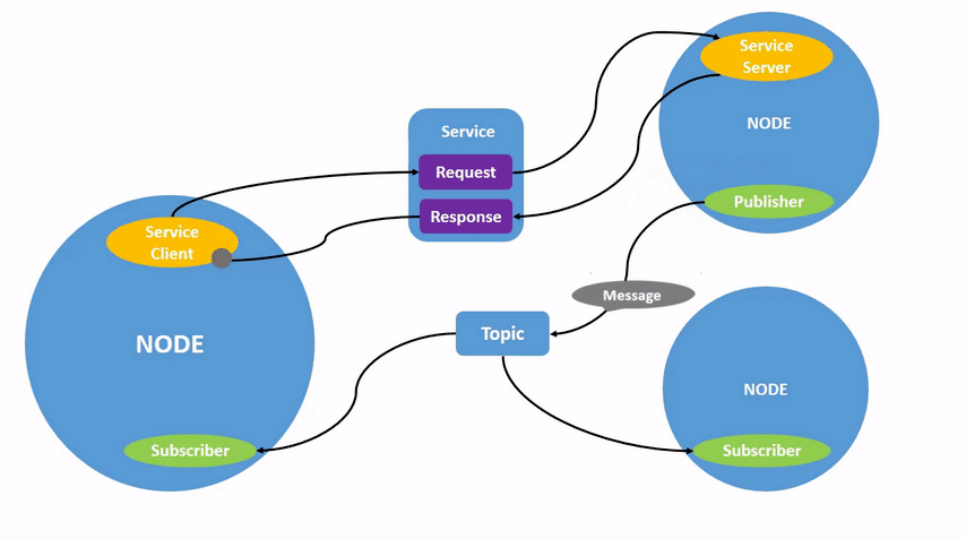


Figura 2.1: Diagrama general funcionamiento ROS[1]

2.1.1. Nodos

Cada nodo de ROS debe ser responsable de un único módulo (por ejemplo, un nodo para controlar los motores de las ruedas, un nodo para controlar un telémetro láser, etc.). Cada nodo puede enviar y recibir datos a otros nodos mediante temas, servicios, acciones o parámetros[14].

En nuestro proyecto los utilizaremos como punto de lectura desde nuestro algoritmo y como punto de emisión, desde el robot, transmitiendo los datos del láser, posición, etc.

2.1.2. Tópicos

Los tópicos son un elemento vital del grafo ROS que actúan como un bus para que los nodos intercambien mensajes. Un nodo puede publicar datos en cualquier número de tópicos y simultáneamente tener suscripciones a cualquier número de tópicos.

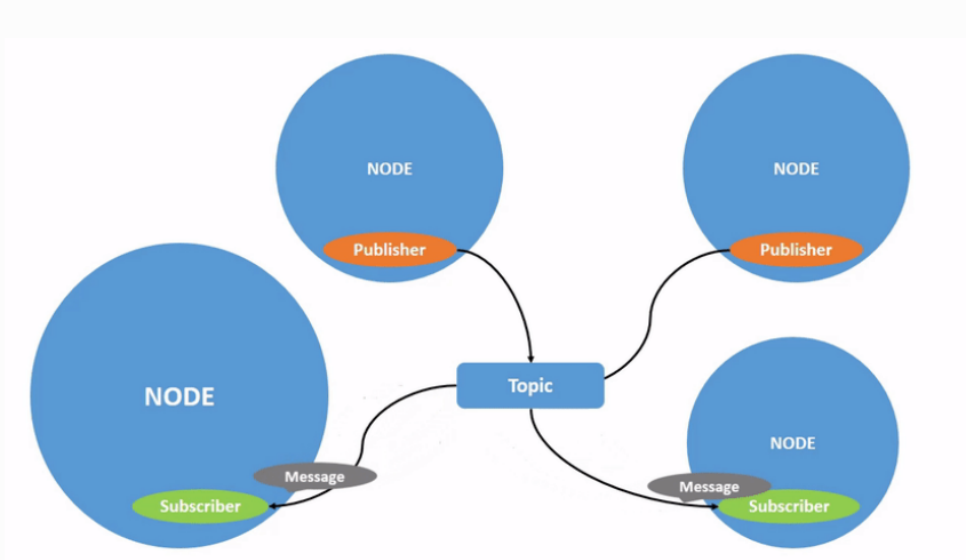


Figura 2.2: Funcionamiento Tópicos[2]

En la figura podemos ver como los nodos que tengan un ***publisher*** publican sus mensajes en los tópicos y los nodos que tienen un ***suscriber*** reciben estos mensajes[15].

En nuestro caso lo usaremos como canal para la trasmisión de mensajes entre el robot y nuestro algoritmo de aprendizaje.

2.1.3. Servicios

Los servicios son otro método de comunicación para los nodos del grafo ROS. Los servicios se basan en un modelo de llamada y respuesta, frente al modelo de editor-suscriptor de los temas. Mientras que los temas permiten a los nodos suscribirse a las secuencias de datos y obtener actualizaciones continuas, los servicios solo proporcionan datos cuando son llamados específicamente por un cliente[16]. Si bien no utilizamos esto directamente, lo utiliza la extensión de OpenAI para ir registrando el progreso para luego poder imprimirlo correctamente.

2.1.4. Comandos básicos

En cuanto a comandos básicos, tenemos los siguientes, lo destaco más que nada para tener una especie de base o línea generales para trabajar, y así acelerar el proceso del proyecto sin buscar en la pagina oficial de ROS.

Comando	Descripción
$catkin_make	Se utiliza para crear el directorio de trabajo catkin. Crear y modificar paquetes de catkin.
$source	Se utiliza para hacer switch entre los distintos directorios de trabajo de ROS
$roslaunch	Lanza un archivo .ros
$rostopic echo	Muestra la salida de un tópico en específico.
$rostopic list	Muestra todos los tópicos actualmente activos.

Tabla 2.1: Resumen Comandos ROS

Estos comandos los usaremos para crear el paquete en el cual vamos a trabajar, además de añadir librerías necesarias como el GYM de OpenAI, o para tareas de depurado como ver los tópicos para ver que nuestros algoritmos funciona bien, etc.

3

El Robot, modelaje y características

Como ya expusimos en la introducción, nosotros usaremos una versión de un robot que se utiliza ampliamente en ROS, ya que ha sido creada para ser usada en conjunto con el mismo ROS, usaremos el TurtleBot.

Tenemos un robot, “low-cost“ de no más de 36 cm de ancho y largo, con dos pequeños motores, que hacen posible el movimiento, hacia delante, girar derecha e izquierda y moverse hacia atrás. Además, se le puede debido al espacio disponible en la altura, se le puede adjuntar un pequeño portátil capaz de hacer las operaciones necesarias para lo que nosotros queramos.

Finalmente, la última pieza para que el robot este completo sería el sensor, podríamos utilizar un sensor Kinect o uno de marca independiente que funcione de manera similar, si tomamos el Kinect, tiene un campo de vision de aproximadamente 58º horizontalmente, con una distancia efectiva de uso de entre 0.8 m y 4 metros[17]. La tecnología de detección de la profundidad y el movimiento que se encuentra en el núcleo de Kinect es posible gracias a su sensor de profundidad. El Kinect original para Xbox 360 utilizaba luz estructurada para ello: la unidad utilizaba un patrón de luz casi infrarroja que se proyectaba en el espacio delante del Kinect, mientras que un sensor de infrarrojos captaba el patrón de luz reflejado. El patrón de luz se deforma en función de la profundidad relativa de los objetos que tiene delante, y se pueden utilizar las matemáticas para estimar esa profundidad en función de varios factores relacionados con la disposición del hardware del Kinect [18].

Es importante destacar que además posee un sensor de giroscopio y de “choque“. Esto, como veremos más adelanto, lo utilizaremos como un láser de “barrido“ tomando muestras de 10 en 10, codificados en 5 sectores. Simplificaremos la información del sensor más adelante en la configuración del robot mediante código.

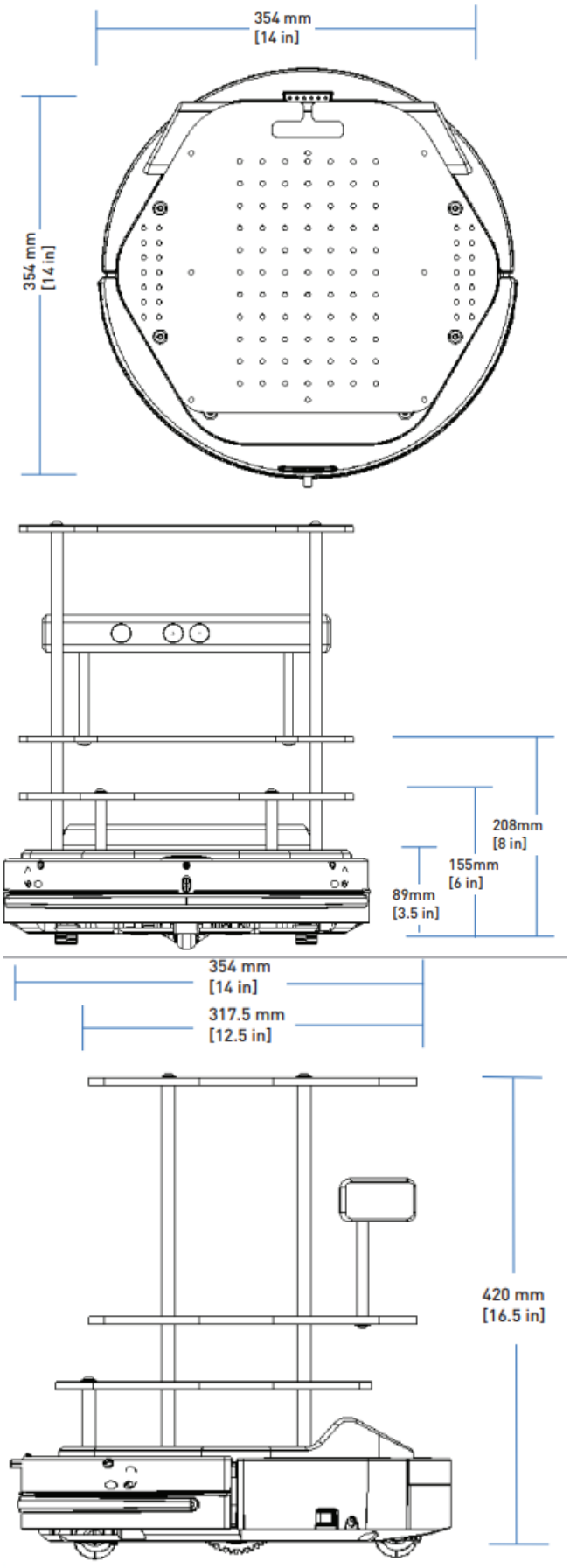


Figura 3.1: Planta, alzado y perfil del Turtlebot[3]

4

Gazebo, el mundo “real“

Para representar el mundo donde se moverá el robot, tenemos que utilizar Gazebo, el mismo es el que coloca el robot donde nosotros queramos, con la configuración que nosotros queramos, sensores, velocidad, obstáculos, etc. En nuestro caso utilizaremos los mapas típicos y sencillos que vienen incluidos con ROS, además que ya tienen configurados los servicios de tópicos, para poder enviar los datos del sensor, velocidad y para recibir la información proporcionada por los algoritmos de OpenAI, por tanto, nos ahorramos trabajo a la hora de implementar estos tópicos.

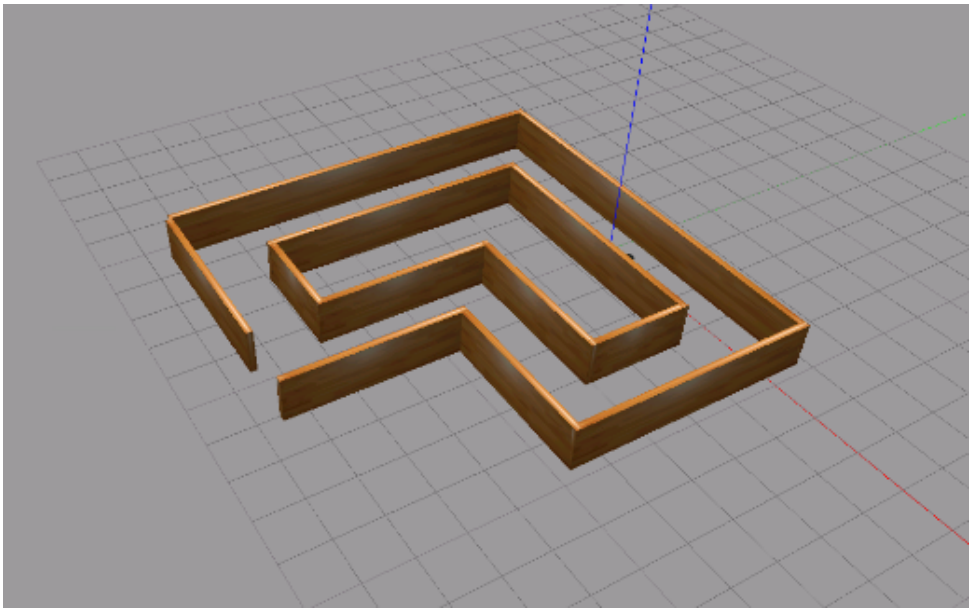


Figura 4.1: Mapa de entrenamiento 1

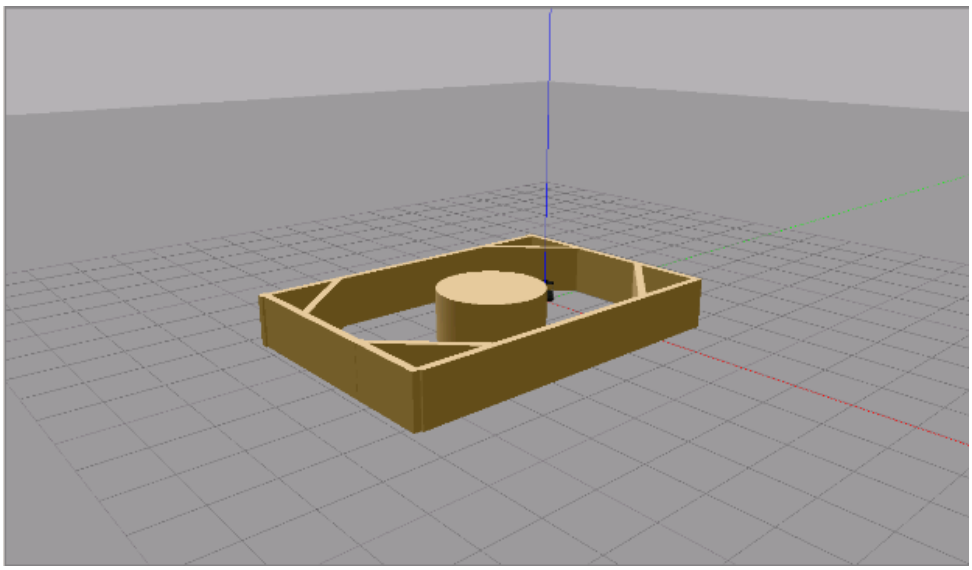


Figura 4.2: Mapa de entrenamiento 2

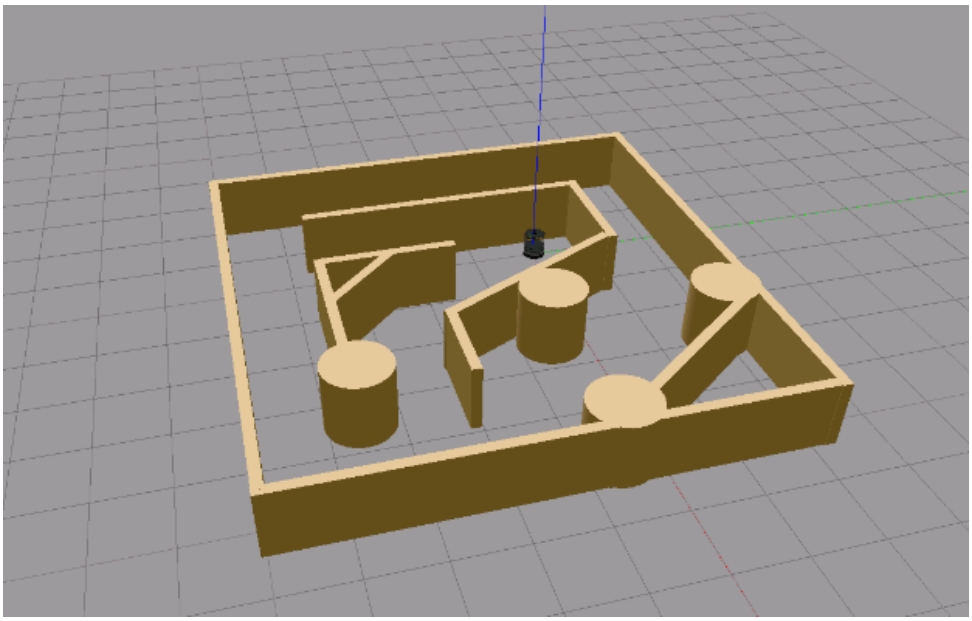


Figura 4.3: Mapa de entrenamiento 3

También cabe destacar que los mapas de entrenamiento son intercambiables para realizar un entrenamiento en distintas situaciones, ejemplo giros cerrados o giros más amplios, o incluso giros en una sola dirección.

5

OpenAI y Algoritmos de aprendizaje

El aprendizaje por refuerzo es un área del aprendizaje automático que se ocupa de cómo los agentes inteligentes deben realizar acciones en un entorno para maximizar la noción de recompensa acumulada. El aprendizaje por refuerzo es uno de los tres paradigmas básicos del aprendizaje automático en Inteligencia Artificial, junto con el aprendizaje supervisado y el aprendizaje no supervisado.

El aprendizaje por refuerzo se diferencia del aprendizaje supervisado en que no necesita que se presenten pares de entrada/salida con etiquetas y en que no necesita que se corrijan explícitamente las acciones subóptimas. En su lugar, se trata de encontrar un equilibrio entre la exploración (del territorio inexplorado) y la explotación (del conocimiento actual).[19]

5.1. Q-Learning

El aprendizaje Q es un algoritmo de aprendizaje por refuerzo sin modelo para aprender el valor de una acción en un estado concreto. No requiere un modelo del entorno (de ahí lo de "sin modelo"), y puede manejar problemas con transiciones y recompensas estocásticas sin requerir adaptaciones.

Para cualquier proceso de decisión de Markov finito (FMDP), el aprendizaje Q encuentra una política óptima en el sentido de maximizar el valor esperado de la recompensa total en todos y cada uno de los pasos sucesivos, empezando por el estado actual. El aprendizaje Q puede identificar una política óptima de selección de acciones para cualquier FMDP dado, con un tiempo de exploración infinito y una política parcialmente aleatoria.[20]

En nuestro caso lo utilizaremos, debido a que la definición del algoritmo, acepta entradas discretas y es particularmente rápido y eficiente, además de su sencillez a la hora de codificarlo.

5.2. OpenAI

Aquí es donde entra en juego la librería de OpenAI, nos da la calidad de crear un gym, o un enviroment donde se encontrara nuestro robot, y él se encargara de ir aplicando el algoritmo que nosotros deseemos (en nuestro caso el Qlearn).

5.2.1. Gym

El agente realiza algunas acciones en el entorno (normalmente pasando algunas entradas de control al entorno, por ejemplo, entradas de torque de los motores) y observa cómo cambia el estado del entorno. Un intercambio de acción-observación de este tipo se denomina timestep.

El objetivo en la RL es manipular el entorno de alguna manera específica. Por ejemplo, queremos que el agente navegue con un robot hasta un punto específico del espacio. Si lo consigue (o avanza hacia ese objetivo), recibirá una recompensa positiva junto con la observación de ese paso de tiempo. La recompensa también puede ser negativa o 0, si el agente no ha tenido éxito (o no ha hecho ningún progreso). El agente será entonces entrenado para maximizar la recompensa que acumule a lo largo de muchos timesteps.

Después de algunos pasos de tiempo, el entorno puede entrar en un estado terminal. Por ejemplo, el robot puede haberse estrellado. En ese caso, queremos restablecer el entorno a un nuevo estado inicial. El entorno emite una señal de hecho al agente si entra en ese estado terminal. No todas las señales de "hecho"deben ser provocadas por un "fallo catastrófico": A veces también queremos emitir una señal de terminado después de un número fijo de pasos de tiempo, o si el agente ha logrado completar alguna tarea en el entorno[21].

Para gestionar toda esta información, mensajes, acciones, estados terminales, etc. Se necesita un gym, que ejecuta o interacciona con una serie de parámetros necesarios para el correcto funcionamiento:

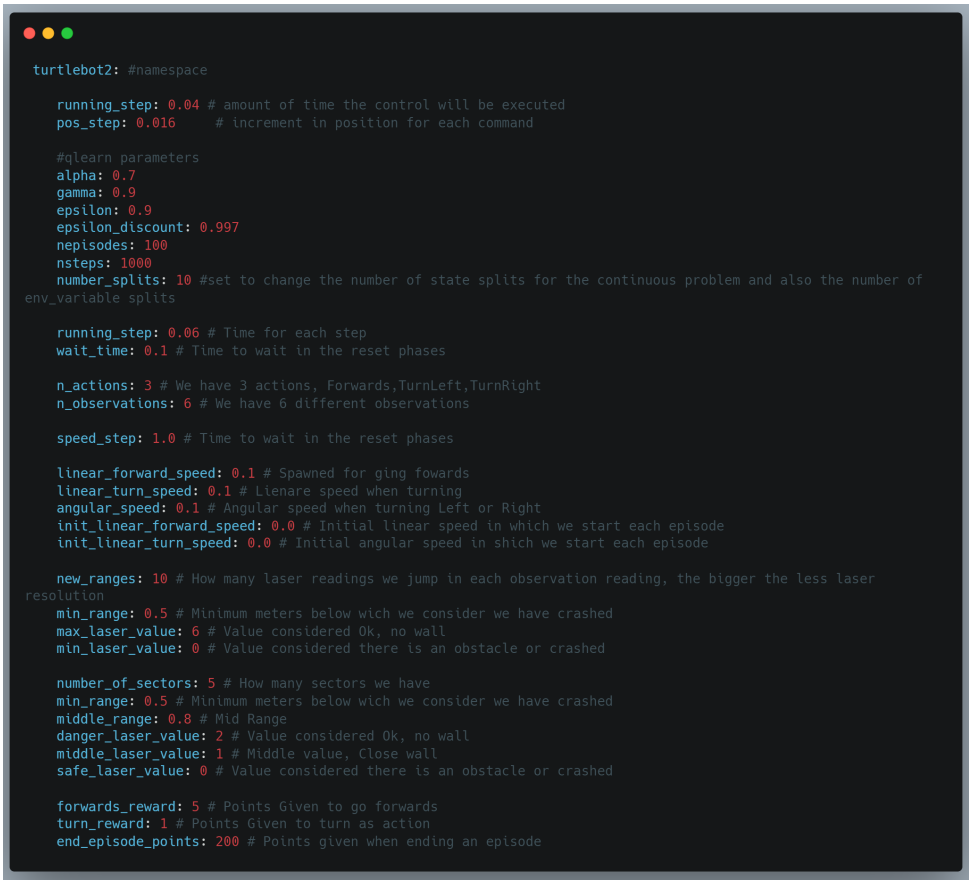
Parámetro	Descripción
Action Space	Es el conjunto de acciones del agente, en el caso del robot solo utilizaremos 3, delante, izquierda y derecha
Observation Space	Es el conjunto de observaciones del agente, lo configuraremos para obtener 5 sectores, con datos discretos con un rango desde el 0 al 2, 0 choca, 1 a punto de chocar, 2 sin peligro
Rango de recompensas	Es el conjunto de recompensas por acciones del agente, solo dará recompensas en el caso de avanzar, 5 hacia delante para beneficiar el movimiento hacia delante, 1 punto si se puede mover hacia la derecha o izquierda y -200 en caso de chocarse, para penalizar gravemente y así evitar los choques
Step o paso	Que código o que acciones se ejecutaran cada vez que sé de una interacción del algoritmo.
Reset	Que código o que acciones se ejecutaran cada vez que finalice una interacción del algoritmo.
Render	Como se va a mostrar el agente, la librería de OpenAI para Ros lo hace por nosotros y nos ahorra el trabajo de representarlo en Gazebo.

Tabla 5.1: Acciones requeridas para ejecutar un Gym de OpenAI

Configuración

A continuación, mostraremos como se traduce toda esta información a código:

En un primer lugar, tenemos un archivo de configuración donde podemos determinar diversos parámetros como el número de acciones, de sectores y lecturas del sensor, recompensas, etc.



```
turtlebot2: #namespace

    running_step: 0.04 # amount of time the control will be executed
    pos_step: 0.016     # increment in position for each command

    #qlearn parameters
    alpha: 0.7
    gamma: 0.9
    epsilon: 0.9
    epsilon_discount: 0.997
    nepisodes: 100
    nsteps: 1000
    number_splits: 10 #set to change the number of state splits for the continuous problem and also the number of
env_variable splits

    running_step: 0.06 # Time for each step
    wait_time: 0.1 # Time to wait in the reset phases

    n_actions: 3 # We have 3 actions, Forwards,TurnLeft,TurnRight
    n_observations: 6 # We have 6 different observations

    speed_step: 1.0 # Time to wait in the reset phases

    linear_forward_speed: 0.1 # Spawned for ging fowards
    linear_turn_speed: 0.1 # Lienare speed when turning
    angular_speed: 0.1 # Angular speed when turning Left or Right
    init_linear_forward_speed: 0.0 # Initial linear speed in which we start each episode
    init_linear_turn_speed: 0.0 # Initial angular speed in shich we start each episode

    new_ranges: 10 # How many laser readings we jump in each observation reading, the bigger the less laser
resolution
    min_range: 0.5 # Minimum meters below wich we consider we have crashed
    max_laser_value: 6 # Value considered Ok, no wall
    min_laser_value: 0 # Value considered there is an obstacle or crashed

    number_of_sectors: 5 # How many sectors we have
    min_range: 0.5 # Minimum meters below wich we consider we have crashed
    middle_range: 0.8 # Mid Range
    danger_laser_value: 2 # Value considered Ok, no wall
    middle_laser_value: 1 # Middle value, Close wall
    safe_laser_value: 0 # Value considered there is an obstacle or crashed

    forwards_reward: 5 # Points Given to go forwards
    turn_reward: 1 # Points Given to turn as action
    end_episode_points: 200 # Points given when ending an episode
```

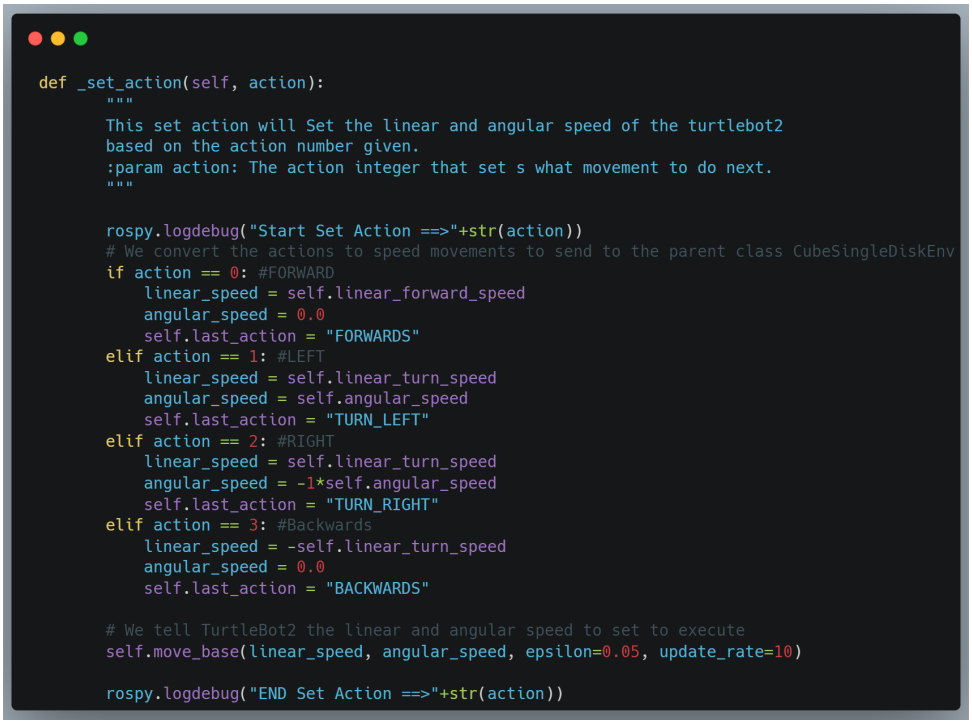
Figura 5.1: Archivo de configuración del gym

Lo más interesante, a destacar, es como mencionamos anteriormente, el número de acciones, en nuestro caso 3, aunque podríamos tener más, lo he decidido limitar para tener un entrenamiento más eficiente.

También podemos ver como definimos el número de barridos que hace el láser, en este caso 10 con un rango de 5 sectores, es decir que de 10 lecturas las simplifica en 5, además también se determina los valores que va a tener el láser a medida que se acerca o no a las paredes, por ejemplo el ***danger_laser_value*** nos dice cuando no hay nada delante y es seguro avanzar, luego el ***mid_range_value*** nos dice en caso de que exista una pared, pero aún se puede avanzar y el ***safe_range*** nos dice cuando el robot se “choca“ o en este caso pasa el mínimo determinado en el ***min_range***.

Espacio de acciones

Visto el archivo de configuración y que solo tenemos 3 acciones, podemos visualizar y explicar el archivo para el espacio de acciones:



```
def _set_action(self, action):
        """
        This set action will Set the linear and angular speed of the turtlebot2
        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))
        # 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 = "FORWARDS"
        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"
        elif action == 3: #Backwards
            linear_speed = -self.linear_turn_speed
            angular_speed = 0.0
            self.last_action = "BACKWARDS"

        # We tell TurtleBot2 the linear and angular speed to set to execute
        self.move_base(linear_speed, angular_speed, epsilon=0.05, update_rate=10)

        rospy.logdebug("END Set Action ==>"+str(action))
```

Figura 5.2: Codificación del espacio de acciones

Si bien aparece una cuarta acción, debido a que en el fichero de configuración está limitado a 3, pues nunca se dará la otra opción, el funcionamiento del espacio de acciones, es que cuando le llega la acción determinada por la función Q en el step, esta le dice al robot como tiene que funcionar, modificando la velocidad angular y la linear, luego lo aplicamos y el robot realizara la acción.

Espacio de las observaciones

Aquí es donde entra el juego el número de sectores para discretizar la salida de los sensores, esto se debe a que la entrada del Q-Learning necesita números enteros para funcionar correctamente. Se reduce las 10 tomas de datos a 5 sectores discretizados y lo hace mediante el siguiente código:

```
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
        TurtleBot2Env API DOCS
        :return:
        """
        rospy.logdebug("Start Get Observation ==>")
        # We get the laser scan data
        laser_scan = self.get_laser_scan()

        discretized_observations = self.discretize_observation( laser_scan,
                                                                self.number_of_sectors
                                                                )


        rospy.logdebug("Observations==>"+str(discretized_observations))
        rospy.logdebug("END Get Observation ==>")
        return discretized_observations
```

Figura 5.3: Codificación del espacio de observaciones

En la figura 5.4, podemos ver como se discretiza el sensor, primero se reducen todas las capas dentro del rango, en este caso 5, luego se va mirando si los datos están por encima o por debajo de ciertos mínimos y rangos, luego se le asigna el valor discretizado, en este caso 1 u 0.

```
def discretize_observation(self,laser_data,number_of_sectors):
        """
        Discards all the laser readings that are not multiple in index of number_of_sectors
        value.
        """

        base = len(laser_data.ranges)/number_of_sectors
        current_sector = -1

        sector_readings = [self.safe_laser_value]*number_of_sectors

        for i, item in enumerate(laser_data.ranges):
            #rospy.logwarn("#### S ###")
            #rospy.logwarn(str(i))
            #rospy.logwarn(str(item))
            rest_is_zero = (i%base==0)

            if rest_is_zero:
                rospy.logwarn("CHANGE SECTOR="+str(rest_is_zero))
                current_sector += 1
            else:
                rospy.loginfo("NO CHANGE SECTOR="+str(rest_is_zero))


            if numpy.isnan(item):
                rospy.logerr(">>>>>>>>>>>>>NAN VALUE=>>>"+str(item))

            elif (self.min_range >= item ):
                sector_readings[current_sector] = self.danger_laser_value

            elif (self.middle_range >= item > self.min_range):
                sector_readings[current_sector] = self.middle_laser_value


        return sector_readings
```

Figura 5.4: Codificación de la discretización del Sensor

Rango de recompensas

Para el rango de recompensas, en la figura 5.1 podemos ver que estaban definidas 3 recompensas, el ***forwards_reward*** que determina la recompensa en ir para delante, en nuestro caso 5, el ***turn_reward*** que determina la recompensa por girar hacia cualquiera de los lados, en nuestro caso 1, y finalmente el ***end_episode_reward*** que determina la recompensa por terminar un episodio, en nuestro caso 200.

```
def _compute_reward(self, observations, done):

        if not done:
            if self.last_action == "FORWARDS":
                reward = self.forwards_reward
            elif self.last_action == "BACKWARDS":
                reward = -1*self.forwards_reward
            else:
                reward = self.turn_reward
        else:
            reward = -1*self.end_episode_points


        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
```

Figura 5.5: Codificación del rango de recompensas

Podemos observar que es un código sencillo que simplemente mediante *ifs* y *elses* determina las recompensas, es interesante determinar que si terminamos el episodio le quitamos todo lo que consiguió poniéndole un menos delante al ***end_episode_reward***.

Step

Si queremos determinar el código que se ejecutara en cada paso de las iteraciones del algoritmo, lo tenemos que hacer por fuera del código intrínseco al gym, es decir, tenemos que combinar todo el código que fuimos viendo anteriormente.

```
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)

            if not (done):
                rospy.logwarn("NOT DONE")
                state = nextState
            else:
                rospy.logwarn("DONE")
                last_time_steps = numpy.append(last_time_steps, [int(i + 1)])
                break
            rospy.logwarn("############### END Step=>" + str(i))
```

Figura 5.6: Codificación de cada paso del algoritmo

Debido a que el gym ya nos proporciona una capa de abstracción simplmente tenemos que hacer ***env.step(action)*** con la acción que determino la función Q.

Reset y Render

Para el reset y el render, no hay que destacar mucho, simplemente el gym env, proporciona los comandos para poder ejecutar ambas acciones, el reset simplemente coloca al robot en su pose inicial. Y el render se complementa junto al gazebo para mostrar al robot y lo hace por dentro de la API.

```
observation = env.reset()
state = ''.join(map(str, observation))
```

Figura 5.7: Codificación del Reset

6

Problemática y posibles soluciones

Debido a que trabajamos con una tecnología, relativamente compleja, como es ROS y sus librerías, que requiere cierta práctica y experiencia es inevitable encontrarse con problemas que debemos solventar, como OpenAI, que si bien es sencillo cuando lo entiendes, requiere cierto tiempo de reflexión y aprendizaje, nunca mejor dicho, y es verdad que a medida que vas profundizando en el proyecto, te vas encontrando con más y más problemas, que es completamente normal y a mi parecer es parte de la experiencia que te ayuda a seguir adelante y aprender, en este capítulo, destacaré los más importantes con la solución (o posible solución) que le he dado.

6.1. Encontrando la configuración óptima

Este es el problema más sencillo que te puedes encontrar, que por una mala configuración se detecten falsos choques u obstáculos, lo cual hace inviable el entrenamiento, incluso que por una falla en la configuración del descuento del algoritmo, en alguna iteración aleatoria del algoritmo, el rendimiento de la recompensa caiga en picada, tiene una solución sencilla pero algo tediosa, que es mediante prueba y error ajustando poco a poco todos los parámetros, investigando por internet, porque cae en picada el rendimiento, viendo que no eres el único, y que ya se ha encontrado una solución, determinar un rango mínimo para que el sensor no detecte como el mismo robot como si fuera un obstáculo o que el robot atraviese paredes como si fuera un fantasma. Probablemente, todo esto se deba a la inexperiencia en el campo del aprendizaje por refuerzo y más en este entorno desconocido para mí.

Respecto a este problema, también me he encontrado un problema a la larga y casi terminando el proyecto, me he dado cuenta de que los entrenamientos no hacían que el robot fuera más "inteligente", me di cuenta, investigando [22], de que el valor épsilon, que determina si el robot va a explotar la solución actual o seguir mejorándola, siguiendo el método actual de experimentación, cada 2000 iteraciones volvía a ser greedy, intentando mejorar la solución a toda costa, por ello cuando evaluaba varios conjuntos de entrenamiento, tenía que también guardar la configuración épsilon.

Finalmente, todo esto sirve de experiencia, para evitar que te pase en futuros proyectos, y saber, al menos, por donde empezar.

6.2. Tiempos de Entrenamiento

En los tiempos de entrenamiento, los cuales son muy altos, para mil iteraciones, tarda aproximadamente 2 horas, he intentado reducirlo de diversas maneras, sin éxito alguno, por suerte, a las 250 iteraciones se puede ver cierta progresión y eficacia a la hora de girar en la primera curva del laberinto, aun así los entrenamientos están siendo divididos, para poder hacerlos cuando me resulte más conveniente, debido a que el entorno que estoy utilizando tiene un time out de 15 minutos, tengo que estar al pendiente para evitar que me eche de la sesión por haber agotado el tiempo de ejecución, desarrolle un código para ir guardando los resultados y los entrenamientos (Que resuelve también el siguiente problema).

Como posible solución a este problema es añadir potencia de cómputo, realizando múltiples entrenamientos al mismo tiempo, pero que aporten al mismo vector Q, también realizando programación en sistemas distribuidos para aumentar potencialmente la potencia de cómputo, en conjunto a sistemas en la nube, como el servicio Azure[23], podrían entrenar 24 horas posiblemente obteniendo resultados muy buenos en un par de días. Aparte esto es algo que ya se ha utilizado varias veces [24] [25], pero que no he implementado por falta de tiempo y conocimiento en este tema.

6.3. Guardar el entrenamiento

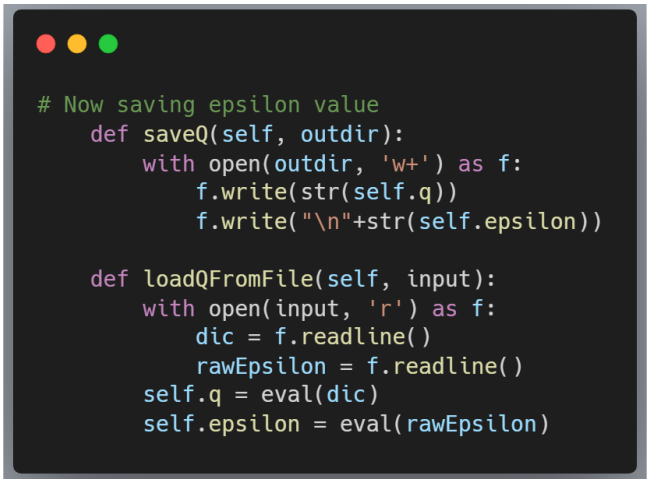
En este tipo de algoritmos es fundamental poder guardar el progreso, a medida que avanza el algoritmo, para ello he tenido que modificar el algoritmo de Q-Learn[26] que ya tenía (Apéndice A.1), no me he dado cuenta hasta ya avanzado en el desarrollo de este Trabajo de Fin De Grado. Añadiendo una simple parte en el código, podemos exportar e importar las veces que queramos el entrenamiento o mejor dicho el vector Q.

```
def saveQ(self, outdir):
        with open(outdir, 'w+') as f:
            f.write(str(self.q))

def loadQFromFile(self, input):
    with open(input, 'r') as f:
        for i in f.readlines():
            dic = i  # string
        self.q = eval(dic)
```

Figura 6.1: Pequeño trozo de código para guardar el vector Q

Más adelante, siguiendo con el punto explicado en 6.1, me di cuenta de que a este código tenía que añadirle la capacidad de guardar y cargar el factor épsilon.



```
# Now saving epsilon value
    def saveQ(self, outdir):
        with open(outdir, 'w+') as f:
            f.write(str(self.q))
            f.write("\n"+str(self.epsilon))

    def loadQFromFile(self, input):
        with open(input, 'r') as f:
            dic = f.readline()
            rawEpsilon = f.readline()
        self.q = eval(dic)
        self.epsilon = eval(rawEpsilon)
```

Figura 6.2: Código mejorado para poder salvar y cargar el valor épsilon.

7

Experimentación y Resultados

El robot consta de un entrenamiento que se reinicia cada vez que sufre un accidente

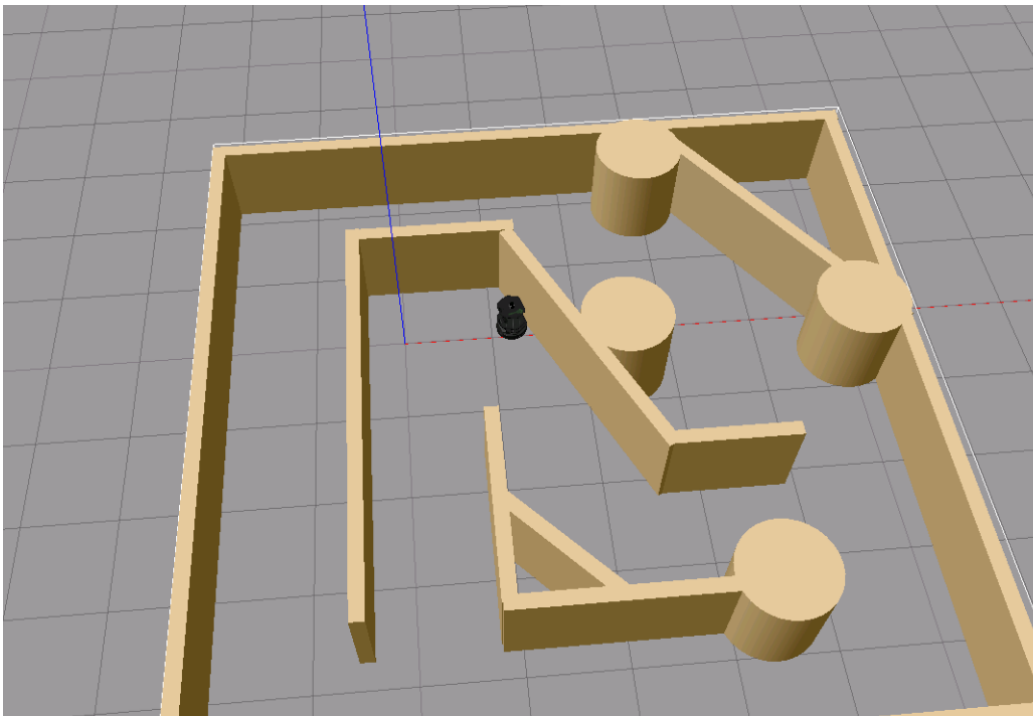


Figura 7.1: Choque al inicio del circuito

Al chocar al inicio no se aprende mucho, y no se cambia tanto el vector Q en consecuencia, tiene su utilidad en las primeras iteraciones, ya que esta aprendiendo el "nuevo mundo".

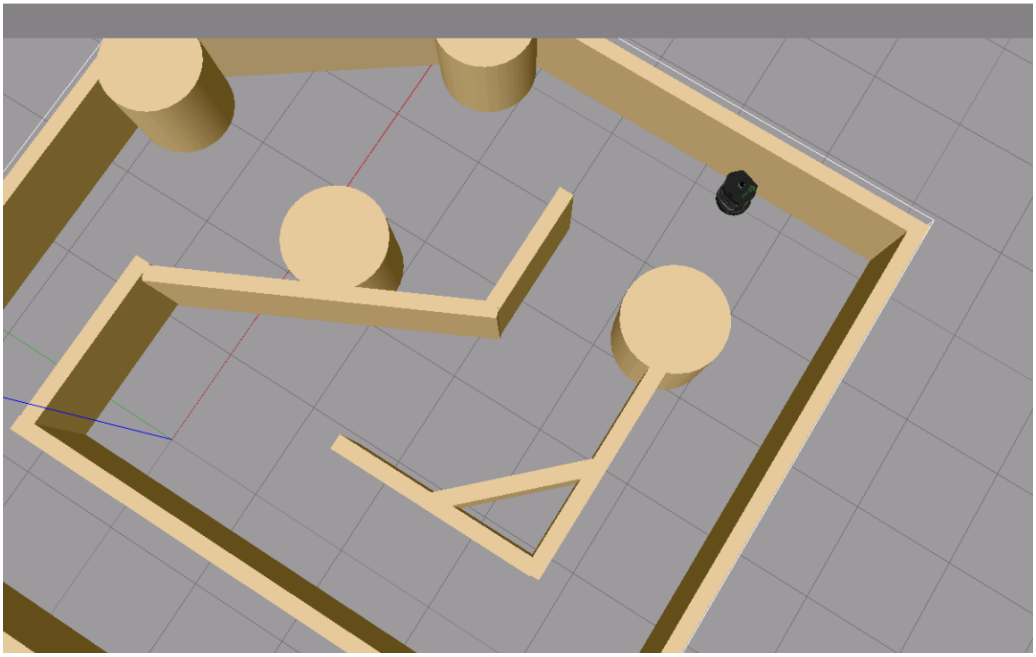


Figura 7.2: Choque al intermedio del circuito

Al chocar más adelante, por ejemplo aprende poco a poco que hacer en una situación donde hay una bifurcación, al final de manera aleatoria se determinara hacia un lado...

7.1. Cálculo épsilon y salida del valor épsilon

Como ya hemos comentado en el capítulo 6 al ajustar hemos tenido problemas con el valor épsilon y por un error de diseño en el algoritmo, tuvimos una salida del valor épsilon de la siguiente manera:

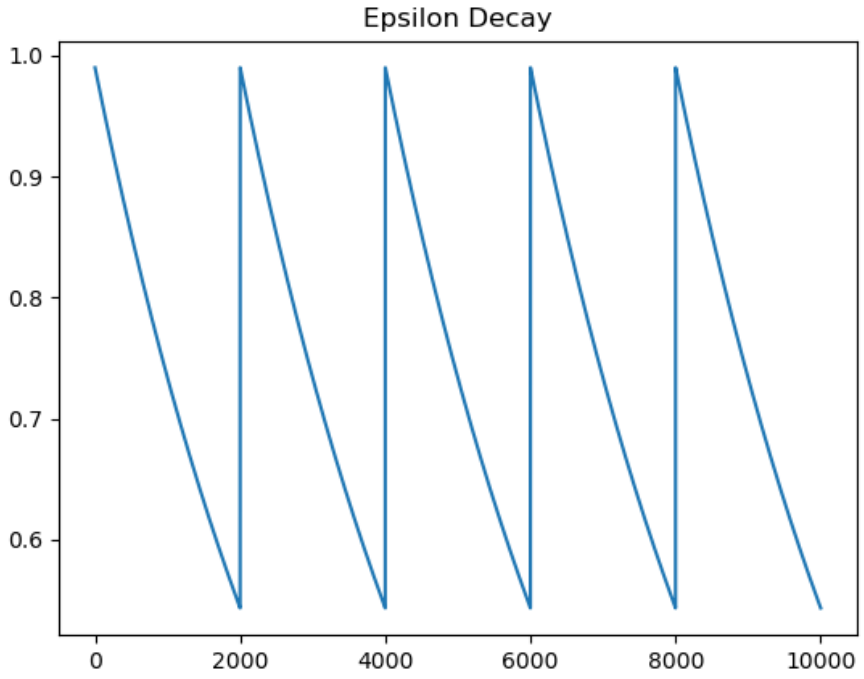


Figura 7.3: Gráfico del valor épsilon durante las primeras 10 mil iteraciones del algoritmo

Esto hacía que la capacidad de explotar la solución al robot que había encontrado y hacía que siempre cada 2 mil iteraciones estuviera reiniciando su entrenamiento buscando nuevas soluciones, y no mantenerse en una línea que era lo que más le favorecía, luego de solucionar el error nos queda el gráfico de la siguiente manera:

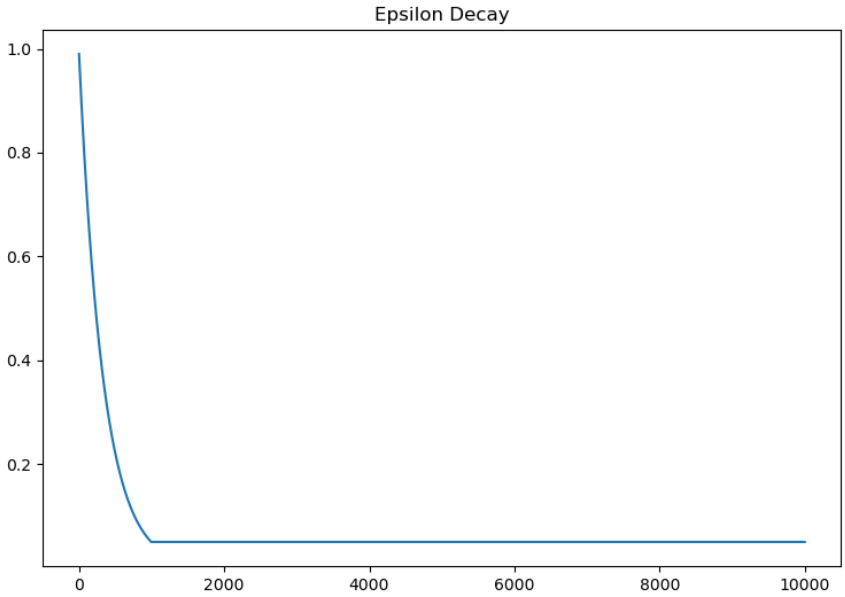


Figura 7.4: Gráfico del valor épsilon durante las últimas 10 mil iteraciones del algoritmo

De esta forma, el robot es capaz de explotar la solución y mantenerse en una misma línea para buscar la solución durante más tiempo antes de explorar nuevo tipo de soluciones[22].

7.2. Salida de la recompensa final

Mirando los datos de la recompensa final, nos damos cuenta de que realmente son bastante malos, ya que también disminuyen con el tiempo, pero esto sospecho que se debe al reinicio del valor de épsilon comentada en la sección 7.1, en total se han invertido más de 16 horas de entrenamiento intercambiando entre distintos entornos donde se mueve el robot, lo que da aproximadamente 20 mil iteraciones del Q-learning.

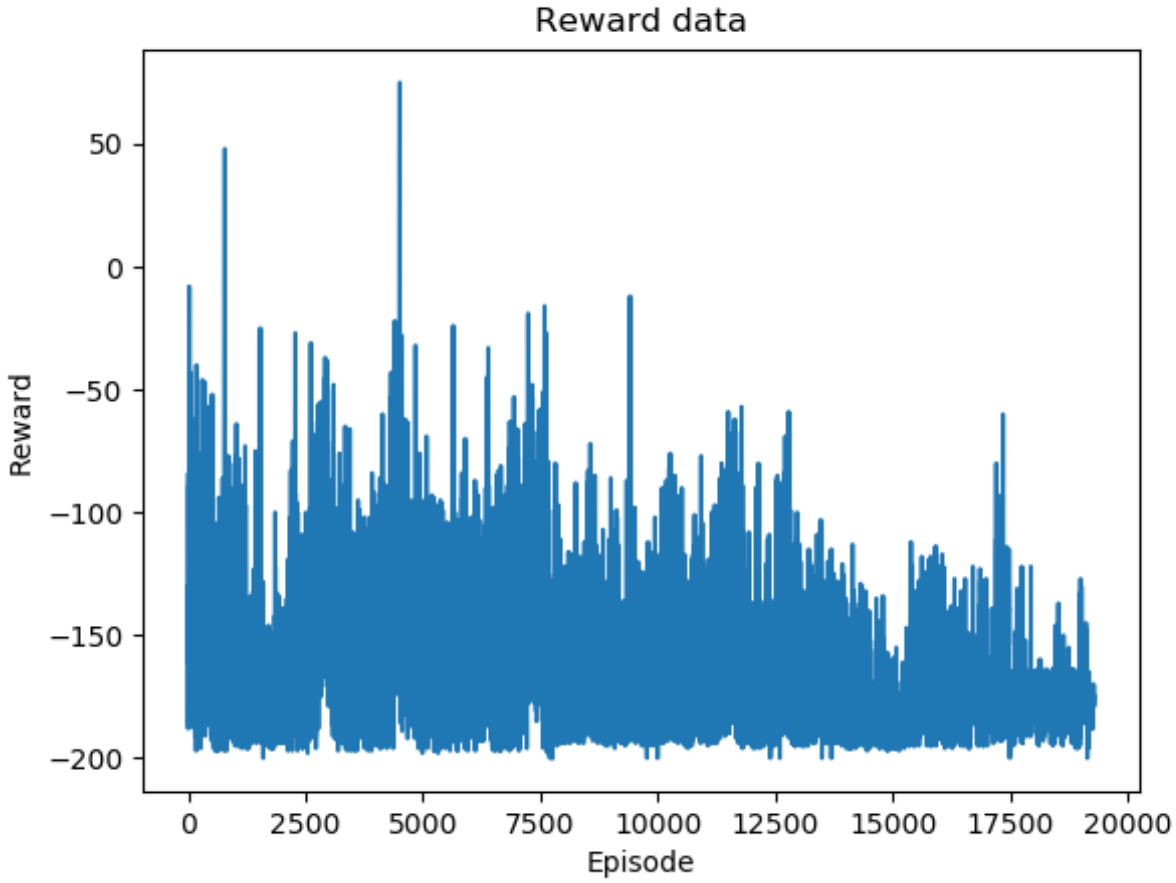


Figura 7.5: Gráfico del entrenamiento total del robot combinando todos los entrenamientos(Épsilon corregidos y no corregido)

Si solo cogemos la información de donde se corrigen las iteraciones de épsilon, menos de 5 mil iteraciones en el mapa de entrenamiento número 3, también tiene resultados algo malos, pero esto se debe a que no tuvo el tiempo suficiente para entrenar.

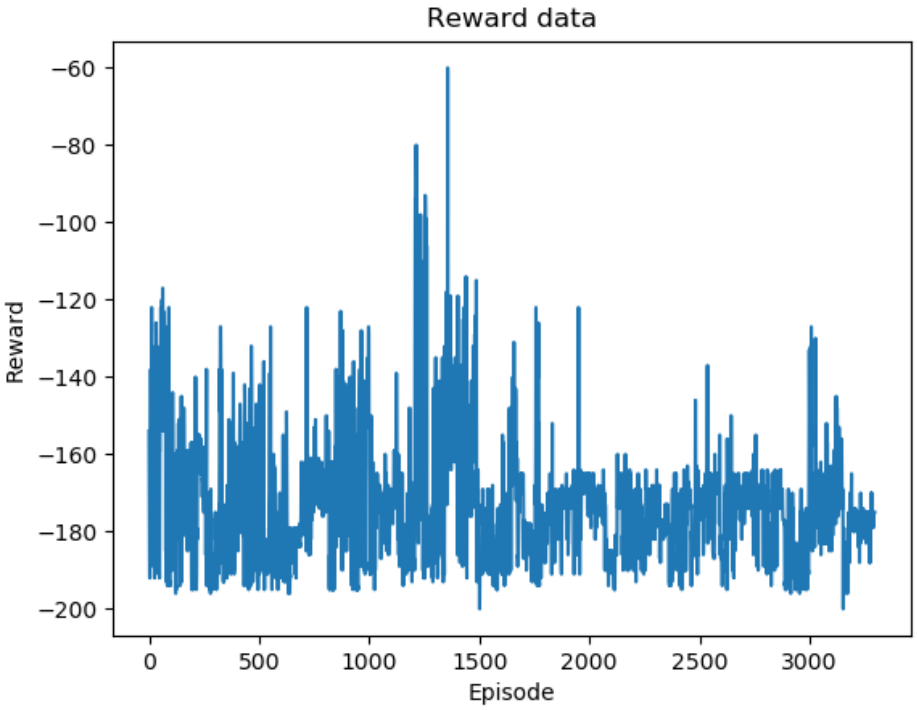


Figura 7.6: Gráfico de entrenamiento con el valor épsilon corregido

8

Conclusiones y líneas futuras

En este capítulo se recogerán las conclusiones basadas en lo aprendido de ROS, Q-Learning y los resultados del entrenamiento.

8.1. Conclusiones

- ROS es un programa complejo y tiene una línea de aprendizaje muy alta.
- ROS está algo abandonado para el turtleBot2, podríamos decir que está discontinuado.
- Se ha conseguido desarrollar un algoritmo Q-Learning que se ajusta a los requisitos del proyecto.
- Es un modelo de Q-Learning estándar, por lo que podría utilizarse en otros robots, o incluso en otras aplicaciones aparte del TurtleBot2.
- La librería de OpenAI simplifica bastante el desarrollo del aprendizaje por refuerzo, ya que facilita los entornos, los tópicos, y todo lo necesario para poder empezar lo antes posible.
- Hemos obtenido resultados bastante malos en cuanto a la recompensa, que al menos por mi parte esperaba en un principio fuera positiva, aun así esto nos hizo darnos cuenta de muchos errores de desarrollo, tener la capacidad de ponerles fin y reentrenar parcialmente el modelo implementado estos "fixes".
- Hemos descubierto un mundo nuevo en cuanto algoritmos de aprendizaje por refuerzo, ya que hay multitud de ellos, e investigando te puedes encontrar miles, incluyendo variantes y mejoras de los originales, como por ejemplo el SARSA, una mejora del Q-Learning.

8.2. Líneas futuras

En esta sección recogeremos aspectos destacables que servirán para la continuidad del proyecto.

- Implementar el SARSA algorithm y compararlo con el Q-Learning, sería la vía natural de esta investigación acerca de los algoritmos de aprendizaje por refuerzo.
- Implementar el modelo sobre otros robots y entornos, para diversificar y normalizar aún más el modelo.
- Probar más configuraciones del algoritmo.
- Dedicarle más tiempo y potencia de entrenamiento, con entrenamiento en paralelo.
- Llevarlo al mundo real y probar la capacidad de moverse en un entorno completamente nuevo y real.
- Implementar una versión más amigable con el desarrollador, llevándolo a entornos más sencillos de programar y con documentación al día, como puede ser Unity ML Agents[27], o similares.

9

Summary and Conclusions

This chapter will draw conclusions based on the learning from ROS, Q-Learning and the results of the training.

9.1. Conclusions

- ROS is a complex program and has a very high learning curve.
- ROS is somewhat abandoned for turtleBot2, we could say it is discontinued.
- A Q-Learning algorithm has been developed that fits the requirements of the project.
- It is a standard Q-Learning model, so it could be used in other robots, or even in other applications apart from TurtleBot2.
- The OpenAI library makes the development of reinforcement learning quite simple, as it provides the environments, topics, and everything needed to get started as quickly as possible.
- We have obtained quite bad results in terms of reward, which at least for my part I expected at the beginning to be positive, even so this made us realize many development errors, have the ability to put an end to them and partially retrain the model by implementing these "fixes".
- We have discovered a whole new world of reinforcement learning algorithms, as there are a multitude of them, and researching you can find thousands, including variants and improvements of the original ones, such as SARSA, an improvement of Q-Learning.

9.2. Future Improvements

- Implementing the SARSA algorithm and comparing it with the Q-Learning would be the natural route for this research on reinforcement learning algorithms.
- Implement the model on other robots and environments to further diversify and standardize the model.
- Test more configurations of the algorithm.
- Dedicate more time and training power to it, with parallel training.
- Take it to the real world and test the ability to move in a completely new and real environment.
- Implement a more developer-friendly version, taking it to simpler programming environments with up-to-date documentation, such as Unity ML Agents [27], or similar.

10

Presupuesto

Este capítulo estimará los costes de realización del proyecto y su desglose en distintas tablas, recursos humanos y materiales, para tener en una visión general del coste total del proyecto

10.1. Presupuesto recursos materiales

Presupuesto necesario relacionado con los componentes materiales de la realización del proyecto.

Componente	**Precio**
Ordenador personal con conexión a internet	700 €
Librerías	0 €
Visual Studio Code	0 €
Total:	700€

Tabla 10.1: Presupuesto recursos materiales

10.2. Presupuesto recursos humanos

Esta sección sirve para estimar los costes de recursos humanos, en definitiva, el número de horas en investigación y desarrollo necesarias para poner a hacer funcionar el proyecto.

Tareas	Número Horas	Coste Bruto por Hora	Coste Bruto Total
Búsqueda y valoración de información relacionada con ROS y OpenAI	100	20 €	2000 €
Búsqueda y valoración de información relacionada con Q-Learn	50	20 €	1000 €
Implementación del algoritmo de Q-Learning	50	20 €	1000 €
Puesta a punto de los entornos y lanzamiento del robot	5	20 €	100 €
Entrenamiento del robot y del modelo, supervisado	90	20 €	1800 €
Documentación y elaboración de la memoria	5	20 €	100€
Total:	300	120 €	6000 €

Tabla 10.2: Presupuesto recursos humanos

10.3. Presupuesto final

Sumando las anteriores tablas y sus totales, nos quedaría el siguiente presupuesto final:

Elemento	Precio
Presupuesto recursos materiales	700 €
Presupuesto recursos humanos	6000 €
Total:	6700€

Tabla 10.3: Presupuesto final

Apéndice A

Algoritmos necesarios para el funcionamiento del robot

A.1. Algoritmo QLearn

```
'''
Enfoque de Q-learning para diferentes problemas de RL
como parte de la serie básica sobre aprendizaje por refuerzo
@https://github.com/vmayoral/basic_reinforcement_learning

Inspired by https://gym.openai.com/evaluations/eval_kWknKOkPQ7izrixdhriurA

        @author: Andrés Zeus Hernández Impini <alu0101207957@ull.edu.es>
'''
import random
import json
import numpy as np


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)

    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)

# save-load code
def saveQ(self, outdir):
    with open(outdir, 'w+') as f:
        f.write(str(self.q))
        f.write("\n"+str(self.epsilon))

def loadQFromFile(self, input):
    with open(input, 'r') as f:
        dic = f.readline()
        rawEpsilon = f.readline()
    self.q = eval(dic)
    self.epsilon = eval(rawEpsilon)
```

A.2. Algoritmo StartTraing.py

```
/*********************************************************************************
 *
 * my_start_traing_maze_qlearn.py
 *
 *********************************************************************************
 *
 * Andrés Zeus Hernández Impini
 *
 * 07/03/2022
 *
 * Este fichero sirve para cargar openai y nuestro algoritmo
 *
 *
 *********************************************************************************/

 #!/usr/bin/env python

import sys
import gym
import numpy
import time
import qlearn
from gym import wrappers
# ROS packages required
import rospy
import rospkg
# import our training environment
import my_turtlebot2_maze


if __name__ == '__main__':

    outdirTraining = str(raw_input("Output dir: "))

    m, s = divmod(int(time.time()), 60)
    h, m = divmod(m, 60)
    outdirTraining = outdirTraining + "_" + \
        str(h) + ":" + str(m) + ":" + str(s)

    rospy.init_node('example_turtlebot2_maze_qlearn',
                    anonymous=True, log_level=rospy.WARN)

    # Create the Gym environment
    env = gym.make('MyTurtleBot2Maze-v0')
    rospy.loginfo("Gym environment done")

    # Set the logging system
    rospack = rospkg.RosPack()
    pkg_path = rospack.get_path('turtle2_openai_ros_example')
    outdir = pkg_path + '/training_results'
    env = wrappers.Monitor(env, outdir, force=True)
    rospy.loginfo("Monitor Wrapper started")

    outdirTraining = pkg_path + '/' + outdirTraining
```

```
print("Archivo de salida: " + str(outdirTraining))

loadTraining = False

loadTrainingInput = str(
    raw_input("Quiere cargar un archivo de entrenamiento? [S/N]: "))
if loadTrainingInput == "S":
    loadTraining = True
    loadTrainingInput = pkg_path + '/' + \
        str(raw_input("Introduzca el archivo de entrenamiento: "))

last_time_steps = numpy.ndarray(0)

# Loads parameters from the ROS param server
# Parameters are stored in a yaml file inside the config directory
# They are loaded at runtime by the launch file
Alpha = rospy.get_param("/turtlebot2/alpha")
Epsilon = rospy.get_param("/turtlebot2/epsilon")
Gamma = rospy.get_param("/turtlebot2/gamma")
epsilon_discount = rospy.get_param("/turtlebot2/epsilon_discount")
nepisodes = rospy.get_param("/turtlebot2/nepisodes")
nsteps = rospy.get_param("/turtlebot2/nsteps")

running_step = rospy.get_param("/turtlebot2/running_step")

# Initialises 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

if loadTraining:
    qlearn.loadQFromFile(loadTrainingInput)

start_time = time.time()
highest_reward = 0

# Starts the main training loop: the one about the episodes to do
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)

        if not (done):
            rospy.logwarn("NOT DONE")
            state = nextState
        else:
            rospy.logwarn("DONE")
            last_time_steps = numpy.append(last_time_steps, [int(i + 1)])
            break
        rospy.logwarn("############### END Step=>" + str(i))
        #raw_input("Next Step...PRESS KEY")
        # rospy.sleep(2.0)
    m, s = divmod(int(time.time() - start_time), 60)
    h, m = divmod(m, 60)
    rospy.logerr(("EP: " + str(x + 1) + " - [alpha: " + str(round(qlearn.alpha, 2)) + " - gamma:
        round(qlearn.gamma, 2)) + " - epsilon: " + str(round(qlearn.epsilon, 2)) + "] - Reward:
        cumulated_reward) + "     Time: %d:%02d:%02d" % (h, m, s)))

    qlearn.saveQ(outdirTraining)
    with open(pkg_path + '/plot_info.txt', "a") as f:
        f.write(str(cumulated_reward)+"\n")

rospy.loginfo(("\n|" + str(nepisodes) + "|" + str(qlearn.alpha) + "|" + str(qlearn.gamma) + "|"
    initial_epsilon) + "*" + str(epsilon_discount) + "|" + str(highest_reward) + "| 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()
```

A.3. Algoritmo plot data

```
#####################################################################
#
# plot-data.py
#
#####################################################################
# Andrés Zeus Hernández Impini
#
# 2022
#
#  This file prints the data from the file data.txt and plot it
#
#####################################################################

## import plt
from array import array
import matplotlib.pyplot as plt
import numpy as np

## import sys


## Load a file from path
def load_file(path):
    with open(path, 'r') as f:
        return f.read()

## get content line by line and return a list of numbers
def get_content(content):
    return [int(x) for x in content.splitlines()]

## plot the data and export it to a png file
def plot_data(data):
    ## calculate  regression
    x = np.array(range(len(data)))
    y = np.array(data)
    z = np.polyfit(x, y, 1)
    p = np.poly1d(z)

    ## name the axis
    plt.xlabel('Episode')
    plt.ylabel('Reward')

    plt.plot(x,y)
    plt.plot(x, p(x), "r--", label="Regression line")
    plt.show()

## main function
def main():
    content = load_file('data4.txt')
    data = get_content(content)
    plt.title('Reward data')
    plot_data(data)

    arrayEpsilonValues = []
```

```
    epsilonValue = 0.99
    for i in range(10000):
        arrayEpsilonValues.append(epsilonValue)
        if epsilonValue > 0.05:
            epsilonValue = epsilonValue * 0.997

    x = np.array(arrayEpsilonValues)
    plt.title('Epsilon Decay')
    plt.plot(x)
    plt.show()



## call main function
if __name__ == '__main__':
    main()
```

Bibliografía

[1] Open Robotics, “Diagrama general funcionamiento ros,” 2022. https://docs.ros.org/en/foxy/_images/Nodes-TopicandService.gif, Consultado el 03/06/2022.

[2] Open Robotics, “Funcionamiento topics ros,” 2022. https://docs.ros.org/en/foxy/_images/Topic-MultiplePublisherandMultipleSubscriber.gif, Consultado el 03/06/2022.

[3] clearpathrobotics, “Turtlebot 2,” 2022. https://clearpathrobotics.com/turtlebot-2-open-source-robot/, Consultado el 06/07/2022.

[4] J. M. Karlsson, *A Decade of Robotics; Analysis of the Diffusion of Industrial Robots in the 1980s by Countries, Application Areas, Industrial Branches and Types of Robots*. Mekanförbundets Förlag, Stockholm, Sweden, 1991.

[5] J. Wallén, *The history of the industrial robot*. Linköping University Electronic Press, 2008.

[6] E. Lytle, “Ai winter,”

[7] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[8] T. Okuyama, T. Gonsalves, and J. Upadhay, “Autonomous driving system based on deep q learnig,” in *2018 International Conference on Intelligent Autonomous Systems (ICoIAS)*, pp. 201–205, IEEE, 2018.

[9] C. Chen, H.-X. Li, and D. Dong, “Hybrid control for robot navigation-a hierarchical q-learning algorithm,” *IEEE Robotics & Automation Magazine*, vol. 15, no. 2, pp. 37–47, 2008.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[11] D. Singh, E. Trivedi, Y. Sharma, and V.Ñiranjan, “Turtlebot: Design and hardware component selection,” in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, pp. 805–809, IEEE, 2018.

[12] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3, pp. 2149–2154, IEEE, 2004.

[13] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo," *arXiv preprint arXiv:1608.05742*, 2016.

[14] Open Robotics, "Understanding ROS 2 Nodes," 2022. https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html, Consultado el 03/06/2022.

[15] Open Robotics, "Understanding ROS 2 Topics," 2022. https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html, Consultado el 03/06/2022.

[16] Open Robotics, "Understanding ROS 2 Services," 2022. https://docs.ros.org/en/foxy/Tutorials/Services/Understanding-ROS2-Services.html, Consultado el 03/06/2022.

[17] C.Ñock, O. Taugourdeau, S. Delagrange, and C. Messier, "Assessing the potential of low-cost 3d cameras for the rapid measurement of plant woody structure," *Sensors (Basel, Switzerland)*, vol. 13, pp. 16216–33, 12 2013.

[18] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus time-of-flight kinect," *Comput. Vis. Image Underst.*, vol. 139, pp. 1–20, 2015.

[19] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[20] Wikipedia contributors, "Q-learning — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=1095183627, 2022. [Online; accessed 6-July-2022].

[21] OpenAI, "Gym api," 2022. https://www.gymlibrary.ml/, Consultado el 06/07/2022.

[22] "Epsilon-greedy q-learning," 2015. https://www.baeldung.com/cs/epsilon-greedy-q-learning, Consultado el 06/07/2022.

[23] "Exploración de los servicios de azure compute." https://docs.microsoft.com/es-es/learn/modules/azure-compute-fundamentals/, Consultado el 06/07/2022.

[24] G. Tesauro and J. O. Kephart, "Pricing in agent economies using multi-agent q-learning," *Autonomous agents and multi-agent systems*, vol. 5, no. 3, pp. 289–304, 2002.

[25] M. Kaisers and K. Tuyls, "Frequency adjusted multi-agent q-learning," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pp. 309–316, 2010.

[26] V. M. Vilches, "Basic reinforcement learning." https://github.com/vmayoral/basic_reinforcement_learning, 2018.

[27] "Unity ml agents," 2022. https://unity.com/es/products/machine-learning-agents, Consultado el 12/07/2022.