UNIVERSIDAD
POLITÉCNICA
DE MADRID

Práctica 1: Wall Following

Asignatura de Robótica

Práctica desarrollada por:

- Eder Tarifa Fernández 21C011

-Mario García Berenguer 21C025

GRADO EN CIENCIA DE DATOS E INTELIGENCIA ARTIFICIAL - CURSO 23/24

1. Introducción a la Práctica de Wall Following.

En esta práctica se nos pedía diseñar un robot modelo Pioneer P3DX que fuera capaz de, a través de sus sensores de ultrasonido, seguir una pared sin chocarse con ella. Nuestra solución propuesta surge de aplicar el algoritmo de Reinforcement Learning Deep Q-Learning para conseguir este resultado. Durante el desarrollo de la práctica hemos sacado un par de conclusiones en relación con el método de resolución usado que ya comentaremos más adelante.
Para el desarrollo de esta práctica se ha usado el software CoppeliaSim y un Jupyter Notebook con código escrito en Python. La escena utilizada en CoppeliaSim ha sido diseñada por nosotros, y también la adjuntaremos junto con distintos vídeos del funcionamiento del robot.
La estructura de la memoria será, en primer lugar, el código diseñado y sus principales puntos a comentar, en segundo lugar la fase de entrenamiento del robot, y por último la fase de test.

2. Arquitectura de Control y Comportamientos.

La arquitectura que hemos diseñado para esta práctica se trata de una arquitectura de enfoque tradicional jerárquica, ya que estamos constantemente en un bucle que va pasando por los niveles definidos. En este caso, en nuestra arquitectura tendríamos 3 principales niveles:

- Nivel de percepción, en el que el robot percibe a través de sus sensores las paredes y obstáculos a su alrededor.
- Nivel de aprendizaje y elección, en el que elegimos la mejor acción para realizar en función de la información sensorial.
- Nivel de acción, en el que indicamos que acción física debe realizar nuestro robot, en nuestro caso la velocidad de cada motor.

La razón de esta arquitectura es que es la más sencilla de implementar al aplicar el algoritmo de Deep Q-Learning, ya que esta sigue un orden al entrenar y realizar las acciones.
Dentro de los niveles de nuestra arquitectura podemos vislumbrar una subdivisión que se podría dar en forma de comportamientos, debido a que cada uno de estos funciona en conjunto para permitir el correcto funcionamiento de todos los niveles, a pesar de que el sistema no esté diseñado como una arquitectura de Brooks al compartir todos los niveles el ciclo de Percepción, Planificación y Acción .
En el nivel de percepción los comportamientos definidos son:

- Percepción sensorial: Obtención de datos de distancias a través de sensores. En el código se representa con la función `get_sonar()`
- Tratamiento de información sensorial: Tratamiento de la información obtenida por los sensores para que sea más útil y manejable por nuestro robot. En el código se representa con la clase `state()` y la funciones `get_metrics()` y `set_state()`

En el nivel de aprendizaje y elección tenemos dos principales comportamientos:

- Aprendizaje a partir de acciones y estados.

- Selección de acción: Elección de la acción que maximice la recompensa conseguida.

A nivel de acción, solo encontramos un comportamiento:

- Realización de la acción a través de los motores.

3. Código de la Implementación.

Como ya hemos comentado, todo el código ha sido desarrollado en Python y a través de un Jupyter Notebook.
En primer lugar hemos diseñado las clases que se van a usar para el algoritmo de Deep Q-Learning, las cuales ya hemos probado previamente en la asignatura de Aprendizaje Automático II.

```python
class ReplayMemory:

    def __init__(self,number_of_observations):
        # Create replay memory
        self.states = np.zeros((MEMORY_SIZE, number_of_observations))
        self.states_next = np.zeros((MEMORY_SIZE, number_of_observations))
        self.actions = np.zeros(MEMORY_SIZE, dtype=np.int32)
        self.rewards = np.zeros(MEMORY_SIZE)
        self.terminal_states = np.zeros(MEMORY_SIZE, dtype=bool)
        self.current_size=0

    def store_transition(self, state, action, reward, state_next, terminal_state):
        # Store a transition (s,a,r,s') in the replay memory
        i = self.current_size % MEMORY_SIZE
        self.states[i] = state
        self.states_next[i] = state_next
        self.actions[i] = action
        self.rewards[i] = reward
        self.terminal_states[i] = terminal_state
        self.current_size = i + 1

    def sample_memory(self, batch_size):
        # Generate a sample of transitions from the replay memory
        batch = np.random.choice(self.current_size, batch_size)
        states = self.states[batch]
        states_next = self.states_next[batch]
        rewards = self.rewards[batch]
        actions = self.actions[batch]
        terminal_states = self.terminal_states[batch]
        return states, actions, rewards, states_next, terminal_states
```

Figura 1. Código correspondiente a la memoria diseñada para almacenar tuplas de acciones, estados y recompensas.

En la Figura 1 vemos el código que crea la memoria donde vamos a estar almacenando todas las acciones, estados y recompensas por las que vayamos pasando en el entrenamiento para poder entrenar nuestra red neuronal con esto. El tamaño de la memoria se decide con un hiperparámetro.

```
class DQN:

    def __init__(self, number_of_observations, number_of_actions):
        # Initialize variables and create neural model
        self.number_of_actions = number_of_actions
        self.number_of_observations = number_of_observations
        self.scores = []
        self.memory = ReplayMemory(number_of_observations)

        # Neural model
        self.model = keras.models.Sequential()
        self.model.add(keras.layers.Dense(20, input_shape=(number_of_observations,), \
                              activation="relu",kernel_initializer="he_normal"))
        self.model.add(keras.layers.Dense(90, activation="relu",kernel_initializer="he_normal"))
        self.model.add(keras.layers.Dense(90, activation="relu",kernel_initializer="he_normal"))
        self.model.add(keras.layers.Dense(number_of_actions, activation="linear"))
        self.model.compile(loss="mse",
optimizer=keras.optimizers.Adam(learning_rate=LEARNING_RATE))

        # Neural model target
        self.model_target = keras.models.Sequential()
        self.model_target.add(keras.layers.Dense(20, input_shape=(number_of_observations,), \
                              activation="relu",kernel_initializer="he_normal"))
        self.model_target.add(keras.layers.Dense(90,
activation="relu",kernel_initializer="he_normal"))
        self.model_target.add(keras.layers.Dense(90,
activation="relu",kernel_initializer="he_normal"))
        self.model_target.add(keras.layers.Dense(number_of_actions, activation="linear"))
        self.model_target.compile(loss="mse",
optimizer=keras.optimizers.Adam(learning_rate=LEARNING_RATE))

    def remember(self, state, action, reward, next_state, terminal_state):
        # Store a tuple (s, a, r, s') for experience replay
        self.memory.store_transition(state, action, reward, next_state, terminal_state)

    def select(self, state, exploration_rate):
        # Generate an action for a given state using epsilon-greedy policy
        if np.random.rand() < exploration_rate:
            return random.randrange(self.number_of_actions)
        else:
            q_values = self.model.predict(state)
            return np.argmax(q_values[0])

    def select_greedy_policy(self, state):
        # Generate an action for a given state using greedy policy
        q_values = self.model.predict(state)
        return np.argmax(q_values[0])

    def learn(self, learn : bool):
        # Learn the value Q using a sample of examples from the replay memory
        if self.memory.current_size < BATCH_SIZE: return

        states, actions, rewards, next_states, terminal_states =
self.memory.sample_memory(BATCH_SIZE)

        q_targets = self.model_target(states).numpy()
        q_next_states = self.model_target(next_states).numpy()

        for i in range(BATCH_SIZE):
            if (terminal_states[i]):
                q_targets[i][actions[i]] = rewards[i]
            else:
                q_targets[i][actions[i]] = rewards[i] + GAMMA * np.max(q_next_states[i])

        self.model.train_on_batch(states, q_targets)
```

```
        if learn:
            weights = self.model.get_weights()
            self.model_target.set_weights(weights)
```

Figura 2. Código correspondiente a la clase diseñada para implementar el algoritmo de Deep Q-Learning.

En la Figura 2 se muestra el código de la clase DQN, con la que definimos todas las funciones que necesitamos para implementar el Deep Q-Learning, como el almacenamiento de tuplas en memoria, el aprendizaje a través de las dos redes neuronales o la selección de la mejor acción.

A continuación se muestran las clases con las que hemos creado tanto el Robot como el Entorno.

```
class Coppelia():

    def __init__(self):
        client = RemoteAPIClient()
        self.sim = client.getObject('sim')

    def start_simulation(self):
        self.default_idle_fps = self.sim.getInt32Param(self.sim.intparam_idle_fps)
        self.sim.setInt32Param(self.sim.intparam_idle_fps, 0)
        self.sim.startSimulation()

    def stop_simulation(self):
        self.sim.stopSimulation()
        while self.sim.getSimulationState() != self.sim.simulation_stopped:
            time.sleep(0.1)
        self.sim.setInt32Param(self.sim.intparam_idle_fps, self.default_idle_fps)

    def is_running(self):
        return self.sim.getSimulationState() != self.sim.simulation_stopped
```

Figura 3. Código de implementación de la clase Coppelia para poder conectarnos con el software CoppeliaSim.

```
class State():
    def __init__(self):
        self.left_wall_metric = 0
        self.right_metric = 0
        self.left_metric = 0
        self.forward_metric = 0
        self.backward_metric = 0
        self.wandering = 0
        self.left_wall = 0
        self.front_wall = 0

    def list_state(self):
        return np.array([[self.left_wall_metric, self.right_metric, self.left_metric,
                          self.forward_metric, self.wandering, self.left_wall, self.front_wall]] )
```

Figura 4. Código de implementación del formato de los states.

Este trozo de código se encarga de especificar el formato que van a tener los estados del robot, además de implementar una función que convierte todo en un array de numpy para permitir el aprendizaje.

```
class P3DX():
    num_sonar = 16
    sonar_max = 1.0

    def __init__(self, sim, robot_id) -> None:
        self.sim = sim
        self.left_motor = self.sim.getObject(f'/{robot_id}/leftMotor')
        self.right_motor = self.sim.getObject(f'/{robot_id}/rightMotor')
        self.sonar = [self.sim.getObject(f'/{robot_id}/visible/ultrasonicSensor[{idx}]') for idx
in range(self.num_sonar)]
        self.learning_network = DQN(7, 5)
        self.state = None
        self.readings = self.get_sonar()

    def get_sonar(self):
        readings = []
        for i in range(self.num_sonar):
            res, dist, _, _, _ = self.sim.readProximitySensor(self.sonar[i])
            readings.append(dist if res == 1 else self.sonar_max)
        return readings

    def get_action(self, exploration_rate, test: bool = False):
        if not test:
            return self.learning_network.select(self.state.list_state(), exploration_rate)
        else:
            return self.learning_network.select_greedy_policy(self.state.list_state())

    def set_speed(self, action):
        if action == 0:
            ispeed, rspeed = self.turn_right(0)
        if action == 1:
            ispeed, rspeed = self.turn_right(0.25)
        if action == 2:
            ispeed, rspeed = self.move_forward()
        if action == 3:
            ispeed, rspeed = self.turn_left(0.25)
        if action == 4:
            ispeed, rspeed = self.turn_left(0)
        self.sim.setJointTargetVelocity(self.left_motor, ispeed)
        self.sim.setJointTargetVelocity(self.right_motor, rspeed)

    def move_forward(self):
        ispeed, rspeed = 1, 1
        return ispeed, rspeed

    def move_backward(self):
        ispeed, rspeed = -1, -1
        return ispeed, rspeed

    def turn_right(self, grade):
        ispeed, rspeed = 1, grade
        return ispeed, rspeed

    def turn_left(self, grade):
        ispeed, rspeed = grade, 1
```

```
        return ispeed, rspeed

    def get_metrics(self, readings):
        metrics = {}
        metrics["forward_metrics"] = readings[3] * readings[4]
        metrics["left_metrics"] = readings[1] * readings[2]
        metrics["right_metrics"] = readings[5] * readings[6]
        metrics["left_wall_metrics"] = readings[0]
        metrics["readings"] = readings
        return metrics

    def set_state(self):
        readings = self.get_sonar()
        metrics = self.get_metrics(readings)
        self.state.right_metric = metrics["right_metrics"]
        self.state.left_metric = metrics["left_metrics"]
        self.state.forward_metric = metrics["forward_metrics"]
        self.state.left_wall_metric = metrics["left_wall_metrics"]
        if metrics["forward_metrics"] < 0.75:
            self.state.wandering = 0
            self.state.front_wall = 1
        if metrics["left_wall_metrics"] < 0.75:
            self.state.wandering = 0
            self.state.left_wall = 1
        if self.state.left_wall == 0 and self.state.front_wall == 0:
            self.state.wandering = 1
        else:
            self.state.wandering += 0.25
            self.state.left_wall -= 0.25
            self.state.front_wall -= 0.25
        return self.state
```

Figura 5. Código que implementa el funcionamiento del robot y de sus sensores.
Este trozo de código crea la clase P3DX para el robot y define una serie de funciones como pueden ser la decisión de qué hacer según la salida de la red neuronal, la actualización del state en el que se encuentra el robot, la obtención de las métricas o de las medidas de los sensores, etc. Por tanto, este trozo de código se refiere tanto al comportamiento encargado de la percepción, del tratamiento sensorial, de la selección y de la acción.

```
class environment():
    def __init__(self, robot_id, sim):
        self.actions = [0, 1, 2, 3, 4] # 0: left, 1: fordward-left, 2:fordward , 3: fordward-right, 4:
right
        self.number_of_observations = 7 # fordward-metrics, left-metrics, right-metrics,
right-wall-metrics, left-wall-metrics, wandering, front-wall, right-wall, left-wall
        self.number_of_actions = 5
        self.robot = P3DX(sim, robot_id)

    def reward(self, multiplier, action):
        reward = 0
        # turn right
        if action == 0:
            if self.robot.state.wandering > 0.75:
                reward = -7.5 * multiplier
            else:
                if self.robot.state.front_wall > 0:
                    reward = 25
                elif self.robot.state.left_wall > 0:
                    reward = -50
```

```
            else:
                reward = 1
        # fordward-turn right
        if action == 1:
            if self.robot.state.wandering > 0.75:
                reward =  -5 * multiplier
            else:
                if self.robot.state.front_wall > 0:
                    reward = -10
                elif self.robot.state.left_wall > 0:
                    if self.robot.state.left_metric < 0.4:
                        reward = 15
                    else:
                        reward = -5
                else:
                    reward = -5
        # fordward
        if action == 2:
            if self.robot.state.wandering > 0.75:
                reward =  -3 * multiplier
            else:
                if self.robot.state.front_wall > 0:
                    reward = -100
                elif self.robot.state.left_wall > 0:
                    if self.robot.state.left_metric < 0.2:
                        reward = -5
                    elif self.robot.state.left_metric < 0.6 and self.robot.state.left_metric > 0.4:
                        reward = 30
                    else:
                        reward = -15
                else:
                    reward = -1
        # fordward-turn left
        if action == 3:
            if self.robot.state.wandering > 0.75:
                reward =  -5 * multiplier
            else:
                if self.robot.state.front_wall > 0:
                    reward = -50
                elif self.robot.state.right_metric < 0.2:
                    reward = 20
                elif self.robot.state.left_metric > 0.6 and self.robot.state.left_metric < 0.9:
                        reward = 20
                elif self.robot.state.left_wall_metric > 0.6 and self.robot.state.left_wall_metric < 0.9:
                        reward = 20
                else:
                    reward = -5
        # turn left
        if action == 4:
            if self.robot.state.wandering > 0.75:
                reward = -7.5 * multiplier
            else:
                if self.robot.state.front_wall > 0:
                    reward = -50
                else:
                    if self.robot.state.left_wall < 0.75 and self.robot.state.left_wall > 0:
                        reward = 25
                    else:
                        reward = -5
        return reward
```

Figura 6. Código que implementa las interacciones entre el entorno y el robot.

Esta parte del código básicamente se encarga de definir las recompensas que va a obtener nuestro robot en función del state del mismo y de la acción que haya realizado, de forma que vamos a intentar siempre mantener una pared a su izquierda.
Las recompensas han sido diseñadas de la siguiente forma:

- La acción de girar a la derecha estará bien recompensada en caso de que haya una pared en frente, si no estamos siguiendo ninguna pared o hay alguna a la izquierda será mal recompensada.
- La acción de girar ligeramente a la derecha estará bien recompensada en caso de que nos estemos pegando mucho a la pared de la izquierda, en otro caso estará mal recompensada.
- La acción de seguir de frente estará bien recompensada en caso de que estemos siguiendo y manteniendo una distancia prudente con la pared, en otro caso estará mal recompensada también.
- La acción de girar ligeramente a la izquierda estará bien recompensada en caso de que nos estemos alejando de la pared de la izquierda o en caso de que nos peguemos mucho a un obstáculo de la derecha. En otro caso la recompensa será negativa.
- La acción de girar a la izquierda estará bien recompensada en caso de que hayamos detectado que estábamos siguiendo una pared a la izquierda y esta ha desaparecido. En otro caso se recompensará negativamente.

Sabemos que también es posible establecer las recompensas en función de las métricas, por ejemplo, dar más recompensas cuanto más tiempo nos mantengamos paralelos a la pared, reducir la recompensa cuanto más nos peguemos a una pared, pero nos pareció algo más difícil de implementar que la idea de asignar una recompensa en función del estado y la acción del robot.

```
coppelia.start_simulation()

episode = 0
# start_time = time.perf_counter()
total_steps = 1
exploration_rate = EXPLORATION_MAX
goal_reached = False
learn_batch = 0
while (episode < MAX_NUMBER_OF_EPISODES_FOR_TRAINING) and coppelia.is_running():
    episode += 1
    steps = 1
    score = 0
    state = robot.set_state().list_state()
    count_action = 1
    last_action = None
    end_episode = False
    while steps < 6000:
        learn_batch +=1
        # Select an action for the current state
        action = robot.get_action(exploration_rate, test=False)
        if action == last_action:
            count_action += 0.25
        else:
            count_action = 1
       # Execute the action on the environment
        reward = env.reward(count_action, action)
```

```
        print(reward, state, action)
        # if state[0, 5] == 1 and action == 2:
        #     reward += 250
        robot.set_speed(action)
        state_next = robot.set_state().list_state()
        print(state, steps, action)

        # Store in memory the transition (s,a,r,s')
        robot.learning_network.remember(state, action, reward, state_next, False)

        score += reward

        # Learn using a batch of experience stored in memory
        if learn_batch == LEARN_BATCH:
            robot.learning_network.learn(True)
            learn_batch = 0
        else:
            robot.learning_network.learn(False)

        state = state_next
        last_action = action
        total_steps += 1
        steps += 1

        writte_log(state, action, reward, score, episode, steps)

    # Decrease exploration rate
        exploration_rate *= EXPLORATION_DECAY
        exploration_rate = max(EXPLORATION_MIN, exploration_rate)

coppelia.stop_simulation()
```

Figura 7. Código que inicia la simulación y el algoritmo de Deep Q-Learning.

```
coppelia = Coppelia()
env = environment(robot_id= "PioneerP3DX", sim=coppelia.sim)
robot = env.robot
robot.state = State()
coppelia.start_simulation()

score = 0
episode = 0
# start_time = time.perf_counter()
total_steps = 1
goal_reached = False
while (episode < MAX_NUMBER_OF_EPISODES_FOR_TRAINING) and coppelia.is_running():
    episode += 1
    steps = 1
    score = 0
    state = robot.set_state().list_state()
    count_action = 1
    last_action = None
    end_episode = False
    while steps < 2000:
        learn_batch +=1
        # Select an action for the current state
        action = robot.get_action(exploration_rate, test=True)
        if action == last_action:
            count_action += 0.25
        else:
            count_action = 1
```

```
        # Execute the action on the environment
        reward = env.reward(count_action, action)
        print(reward, state, action)

        robot.set_speed(action)
        state_next = robot.set_state().list_state()
        print(state, steps, action)

        score += reward

        state = state_next
        last_action = action
        total_steps += 1
        steps += 1

        writte_log(state, action, reward, score, episode, steps)

coppelia.stop_simulation()
```

Figura 8. Código que inicia la simulación.

Este último trozo de código es el usado para realizar la parte de Test en nuestra práctica.

Una vez mostrado el código de nuestra implementación, toca llevar a nuestro modelo a la fase de entrenamiento.

4. Entrenamiento del Modelo.

Para entrenar el modelo hemos realizado decenas de ejecuciones cambiando los hiperparámetros del mismo, tales como la tasa de aprendizaje, el número de neuronas, etc. Sin embargo, después de mucha experimentación nos hemos dado cuenta que el parámetro más importante con el que jugar son el número de neuronas y capas ocultas. Esto es porque, si el número de neuronas es menor que un umbral, en este caso parece ser 170 neuronas en total más o menos, el robot parece tener cierta tendencia a girar y no seguir las paredes. Parece adquirir una estrategia que consiste en estar algo de tiempo girando en una de las esquinas para conseguir puntos para después salir hacia cualquier dirección y volver a llegar a una esquina y seguir girando. Este comportamiento se puede ver aquí. Para este modelo es muy difícil seguir las paredes y las esquinas, pero no se suelen chocar ya que normalmente se dedican a girar cerca de una pared pero sin ir recto hacia ellas.

Si por el contrario usamos un número mayor de neuronas nos encontramos que la táctica que va a seguir el robot para evitar chocarse con las paredes y aún así seguirlas es realizar un zigzag, como podemos ver en los siguientes videos: video 1, video 2. Para este modelo seguir las paredes sin chocarse es algo muy sencillo, sin embargo con las esquinas de 270º o de 360º puede llegar a tener problemas.

Sin embargo, con algo de suerte y acertando en el número de neuronas podemos obtener un modelo que no realiza zigzag, pero es capaz de seguir las paredes de forma recta y girar

la porción justa. En el siguiente vídeo se puede ver esta evolución y como pasa de no poder realizar correctamente los giros y seguir recto a seguir perfectamente la pared.

Otras pruebas que hemos realizado nos han dado lugar a resultados muy poco entrenados que no eran capaces de realizar bien algunas curvas, como este ejemplo.

El entrenamiento se realiza en un recinto cerrado y bastante simple, en el que pretendemos que el robot aprenda a seguir las paredes y a realizar giros de 90º y 180º. El escenario es el siguiente:

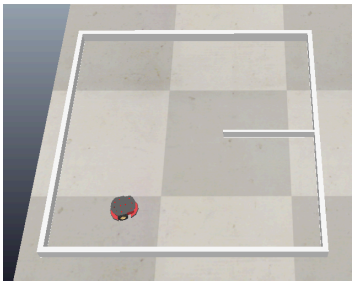

Figura 9. Zona de entrenamiento básica.

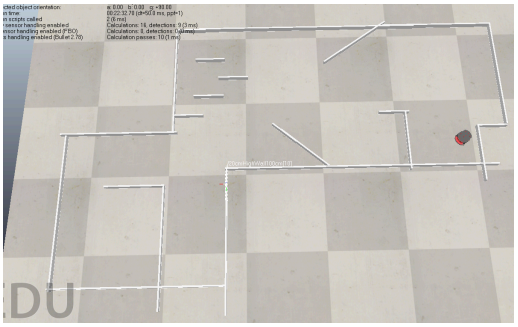

Figura 10. Zona de entrenamiento compleja

Después de realizar alrededor de 4000 steps en este escenario, lo cambiamos a una zona donde más tarde se van a realizar el test para que también aprenda sobre este recinto y así obtener mejores resultados.

Durante el entrenamiento hemos obtenido las recompensas acumuladas de nuestro robot. El mejor resultado para estas ha sido con un modelo de zigzag, como podemos ver en la Figura 11.

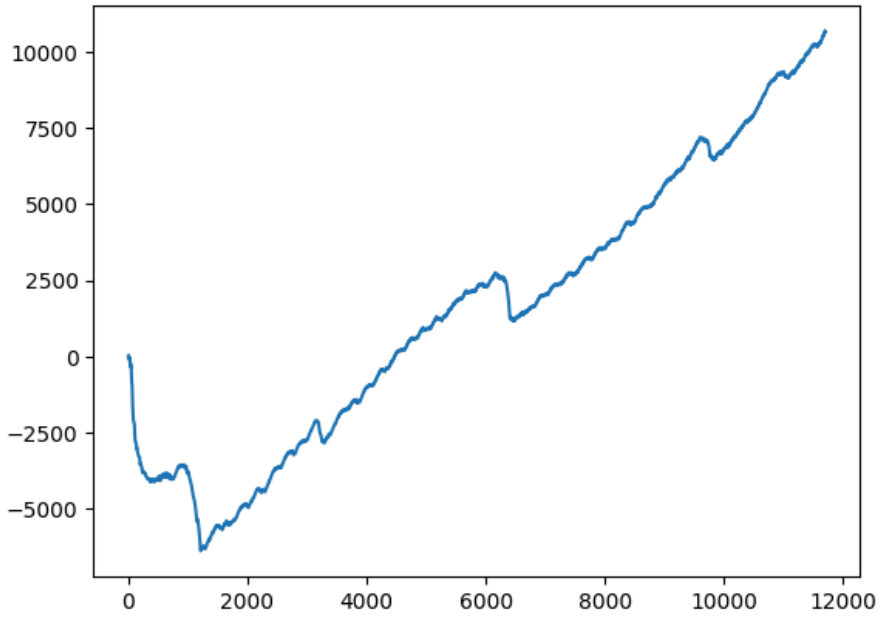

Figura 11. Mejor puntuación conseguida por un modelo en entrenamiento.

5. Test del Modelo.

El test de nuestro robot también se realiza en el entorno de la Figura 10, ya que consideramos que es un entorno bastante completo y que tiene una gran variedad de obstáculos que puede superar nuestro robot.

El mejor modelo que hemos obtenido, como ya hemos comentado, conseguía ir recto durante los tests y realizaba perfectamente las curvas. Sin embargo, no hemos podido volver a entrenar un modelo tan bueno, y principalmente han llegado a la fase de test modelos de zigzag. Aquí podemos ver un vídeo del nuestro mejor modelo, que podía realizar el circuito de forma casi perfecta, aunque en algún momento fallaba las curvas de 270º y 360º. No obstante, este seguía las paredes de forma recta, consiguiendo mucha soltura. Durante este test también le fuimos cambiando el escenario para ver que tal se adaptaba a él.

Por otro lado tenemos el test de uno de los modelos de zigzag obtenidos, que al igual que el anterior, fallaba en alguna curva de 270º o 360º. Sin embargo, es una alternativa muy buena también y que nunca pierde el contacto con la pared, como podemos ver en este test.

De la misma forma que hemos obtenido la puntuación en entrenamiento, también la tenemos en los test. Esta vez la mejor puntuación fue obtenida por el modelo que conseguía seguir la pared sin hacer zigzag. Esto se puede ver en la Figura 12.

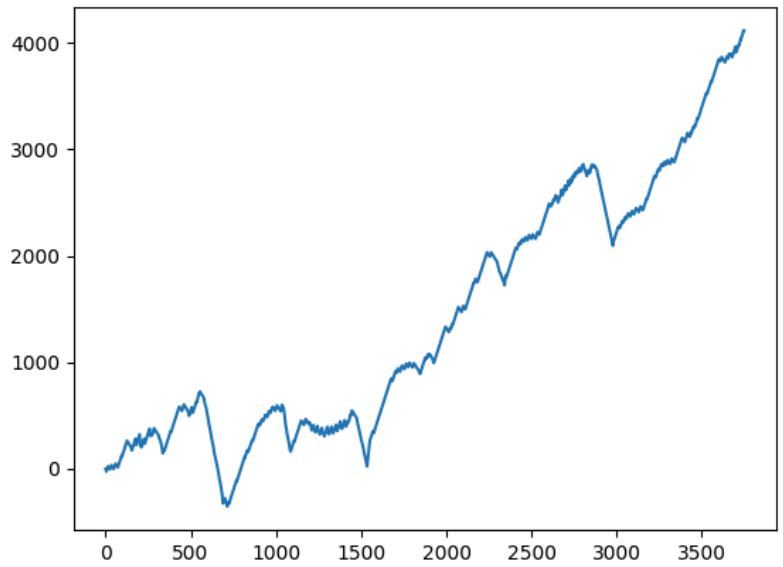

Figura 12. Mejor puntuación obtenida por un modelo en test.

Esas caídas de puntuación tan grandes corresponden con los momentos en los que cambiamos el escenario, ya que no lo hacemos correctamente al estar el robot demasiado cerca e influimos negativamente en los resultados. Sin embargo, ese robot es sin duda el mejor modelo en test que hemos conseguido desarrollar.

6. Conclusiones y Posibles Mejoras.

Las principales conclusiones que sacamos de esta práctica es que el comportamiento que debe seguir el robot es demasiado simple, y un algoritmo de Deep Q Learning nos parece que vuelve demasiado complejo el problema. En nuestro caso también realizamos pruebas con un algoritmo heurístico antes de plantear el Deep Q Learning, y la mejora realmente no es tan grande. Otro punto negativo del Deep Q Learning es que no somos capaces de entender realmente porque el robot decide realizar una acción o otra, por lo que solo podemos obtener suposiciones.
Otra conclusión que hemos sacado es que, como ya hemos comentado, el número de neuronas determinaba el comportamiento del robot, si decidía seguir una estrategia de seguir recto las paredes o por otro lado seguirlas en zigzag. Parece ser que cuantas más neuronas añadías al modelo, más buscaba la estrategia de zigzag. Nuestra reflexión sobre esto es que, dado que la penalización por dejar de seguir la pared es bastante, el robot decide seguir la estrategia de zigzag no por gusto, sino porque resulta más fácil para él seguir la pared de esta forma, ya que esto le ayuda a no chocar con la pared pero tampoco irse muy lejos de ella. La principal pega de esta estrategia es que los giros hacia la izquierda no siempre los realiza correctamente, ya que depende de en qué punto del zigzagueo se encuentre, podría incluso chocar con la pared. Sin embargo, esta estrategia sí que resulta óptima para los giros hacia la derecha y para mantenerse cerca de los muros.
Por otro lado, si decidimos añadir menos neuronas, nos encontramos con un robot que gira demasiado en las esquinas. No es capaz de seguir las paredes como debería y se dedica a vagar por el escenario llegando a las esquinas y realizando giros, los cuales les da una buena recompensa. Hay casos en los que este comportamiento ha convergido a un resultado más satisfactorio, que es capaz de girar la porción justa al realizar los giros como para seguir la pared, sin embargo no es para nada habitual y apenas hemos conseguido modelos de este estilo. En conclusión, parece ser que el robot la mejor estrategia que plantea es el zigzag.
Por último, también comentar que la definición de las recompensas es una parte muy clave para el algoritmo, ya que estas van a dictar qué acción es correcta realizar en cada momento, por lo que acaban definiendo el comportamiento del robot.

Algunas de las mejoras que hemos pensado para esta práctica son por un lado cambiar la forma de abordarla, y usar otro método más simple que el Deep Q Learning y que no tenga tanta complejidad y sea más explicable. También podríamos cambiar la forma que tenemos para definir los estados del robot, y crear o modificar las métricas usadas para aprender para que se adapten mejor al desarrollo de la práctica. Por último, creemos que otra mejora podría ser las recompensas, que podrían cambiar con más tiempo de experimentación para adaptarlas completamente al objetivo de nuestro robot y no dejar esa libertad al modelo de que pueda elegir entre varias tácticas.

7. Bibliografía y Recursos.

Jupyter Notebook con Código del Desarrollo:
https://drive.google.com/file/d/1arfgslIkCqwDj4FzHhesj052uQh6B9ed/view?usp=sharing

Escenario de Entrenamiento Y Test:
https://drive.google.com/file/d/1Wqlvj3la998EdhlK8JNV0AhDbUFaxqAm/view?usp=sharing
Diapositivas sobre Reinforcement Learning, Martín Molina.
Diapositivas sobre Deep Q Learning, Martín Molina.
Diapositivas sobre Arquitecturas de Control, Javier de Lope Asiaín.